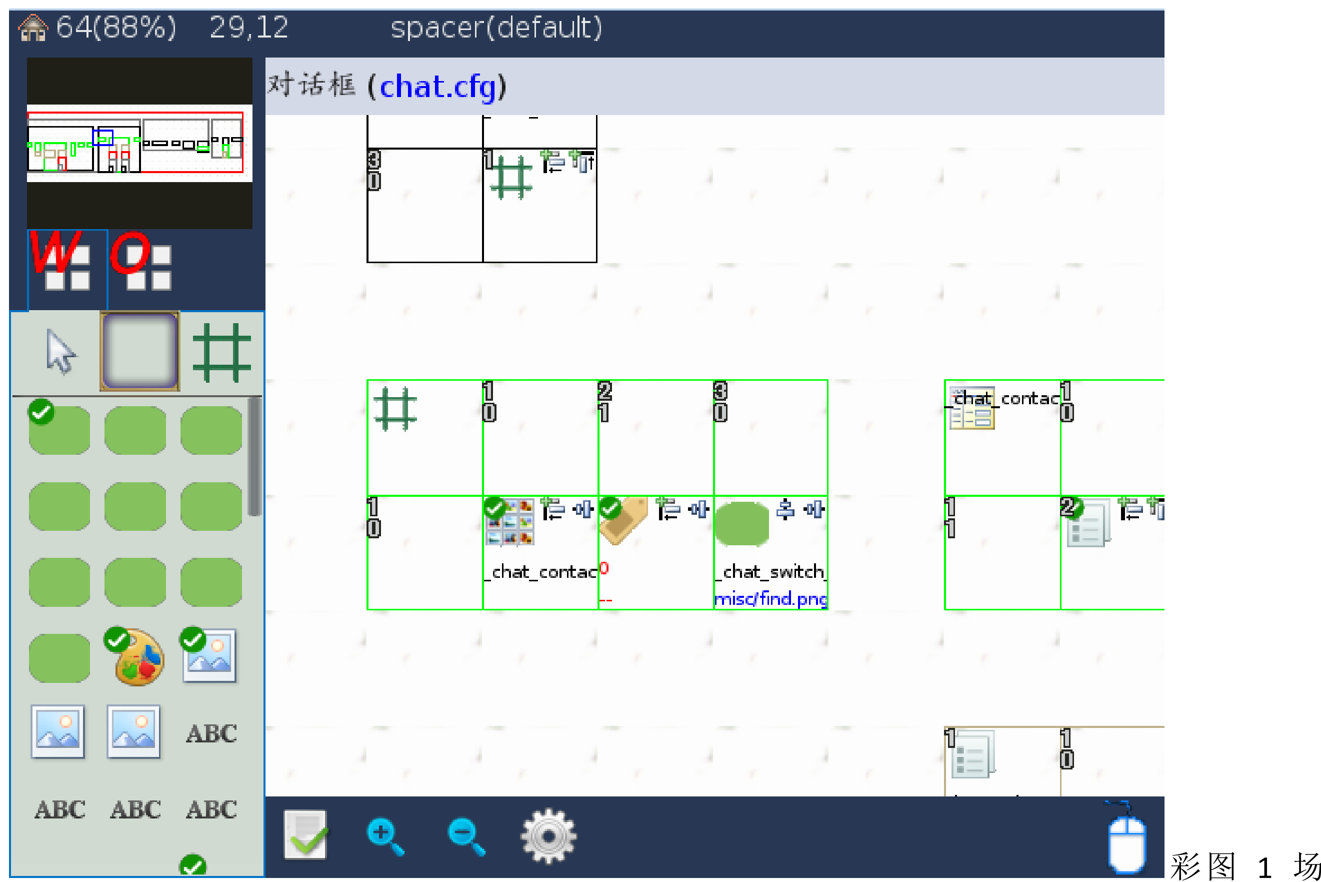
《Rose 编程指南》

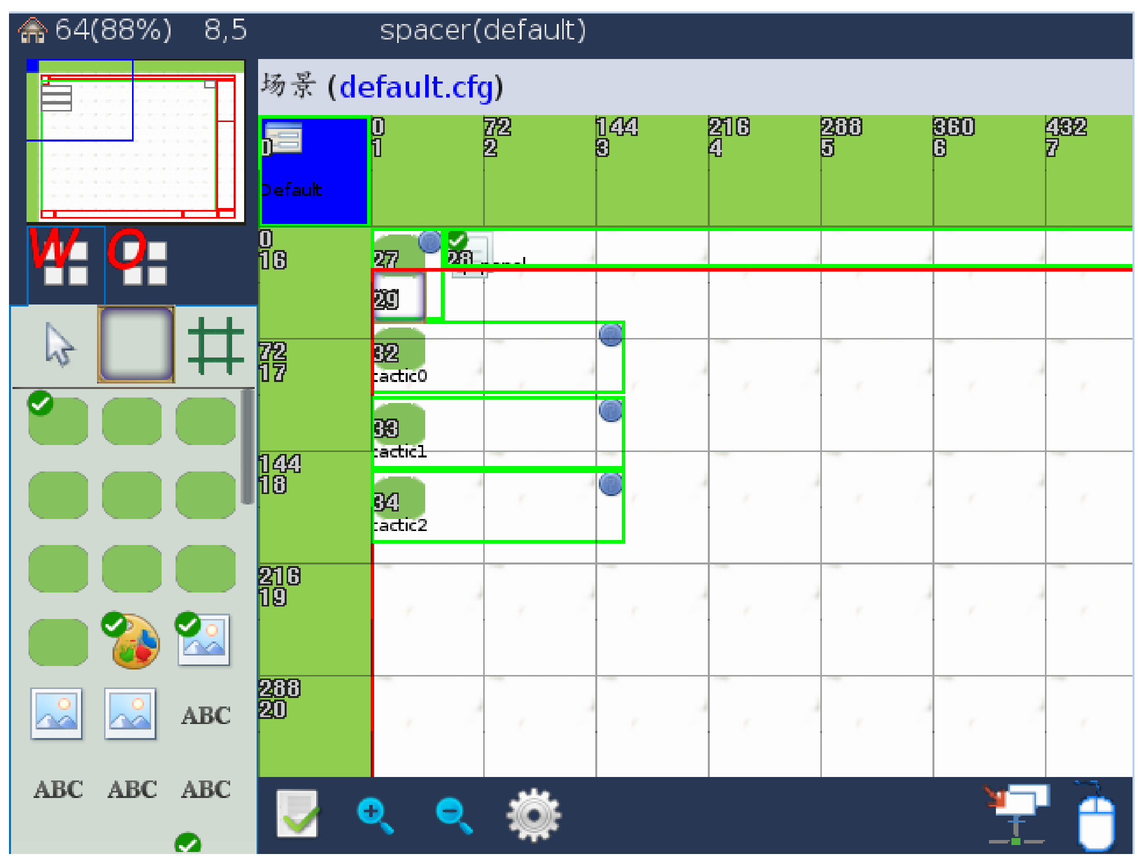
用于开发人工智能 App 的跨平台 SDK

日期：2019/12/9

中文 SDL：http://www.libsdl.cn。
Rose Github：https://github.com/freeors/Rose。
Leagor SDL Github：https://github.com/freeors/SDL。
邮箱：ancientcc@leagor.com

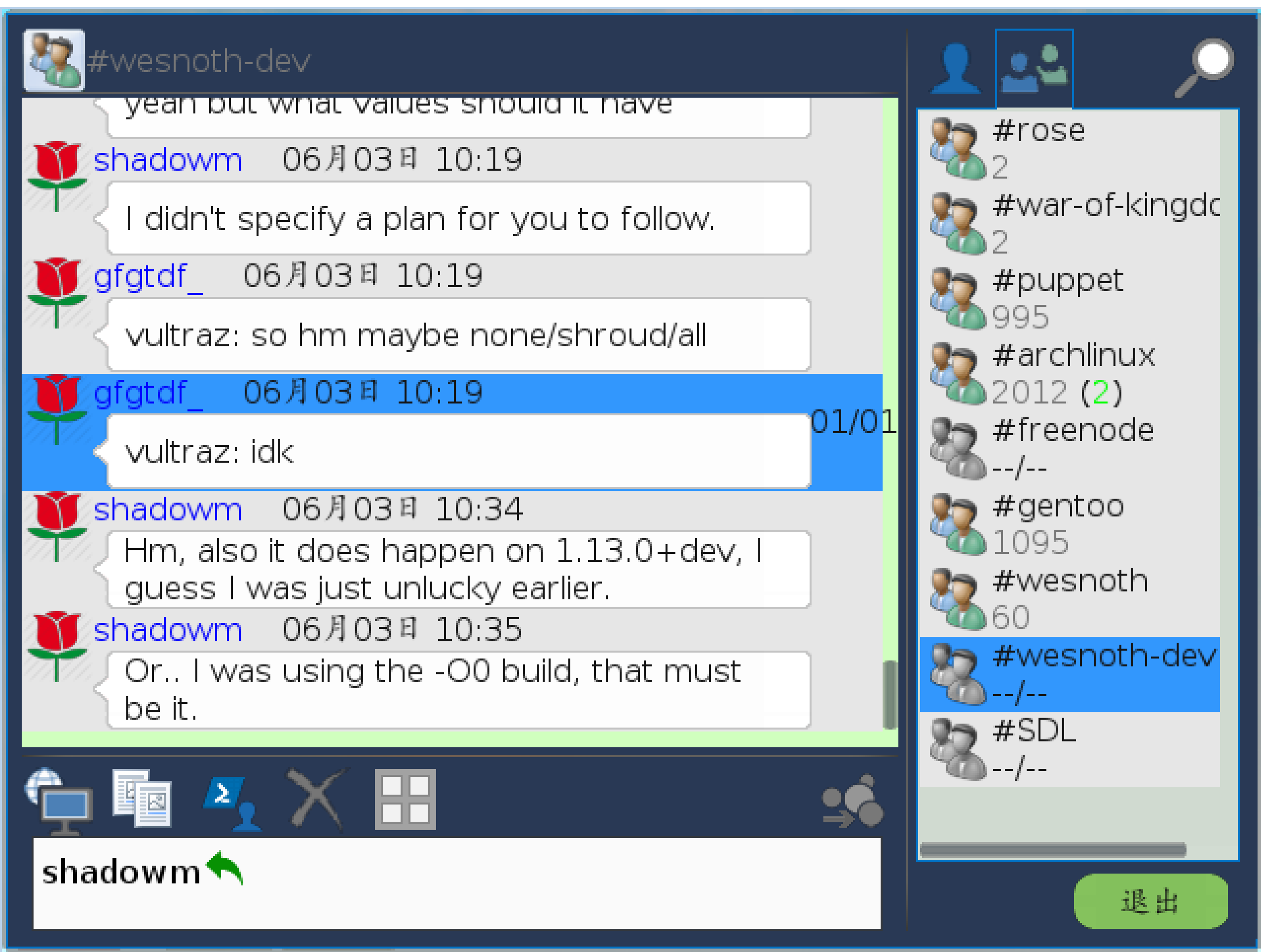


彩图 1 场景。Studio 中编辑对话框。地图栅格是正方形，栅格坐标系值等于单元坐标系。水平方向未填满整行，采用边界设置的“background_image”的图像去填塞。场景中存在三个报表控件。

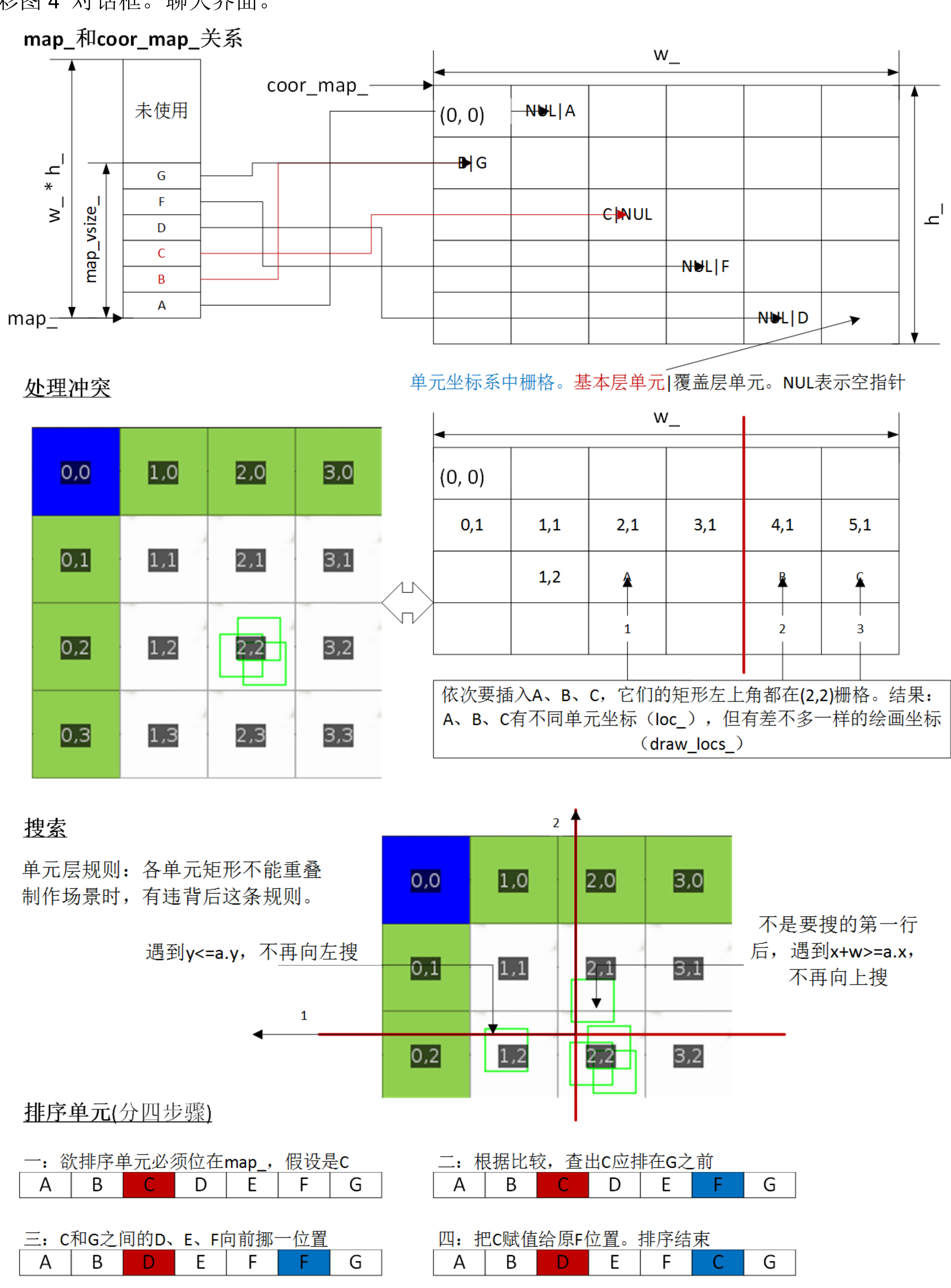


彩图 2 场景。Studio 中编辑场景。地图栅格是正方形，栅格坐标系值不等于单元坐标系。

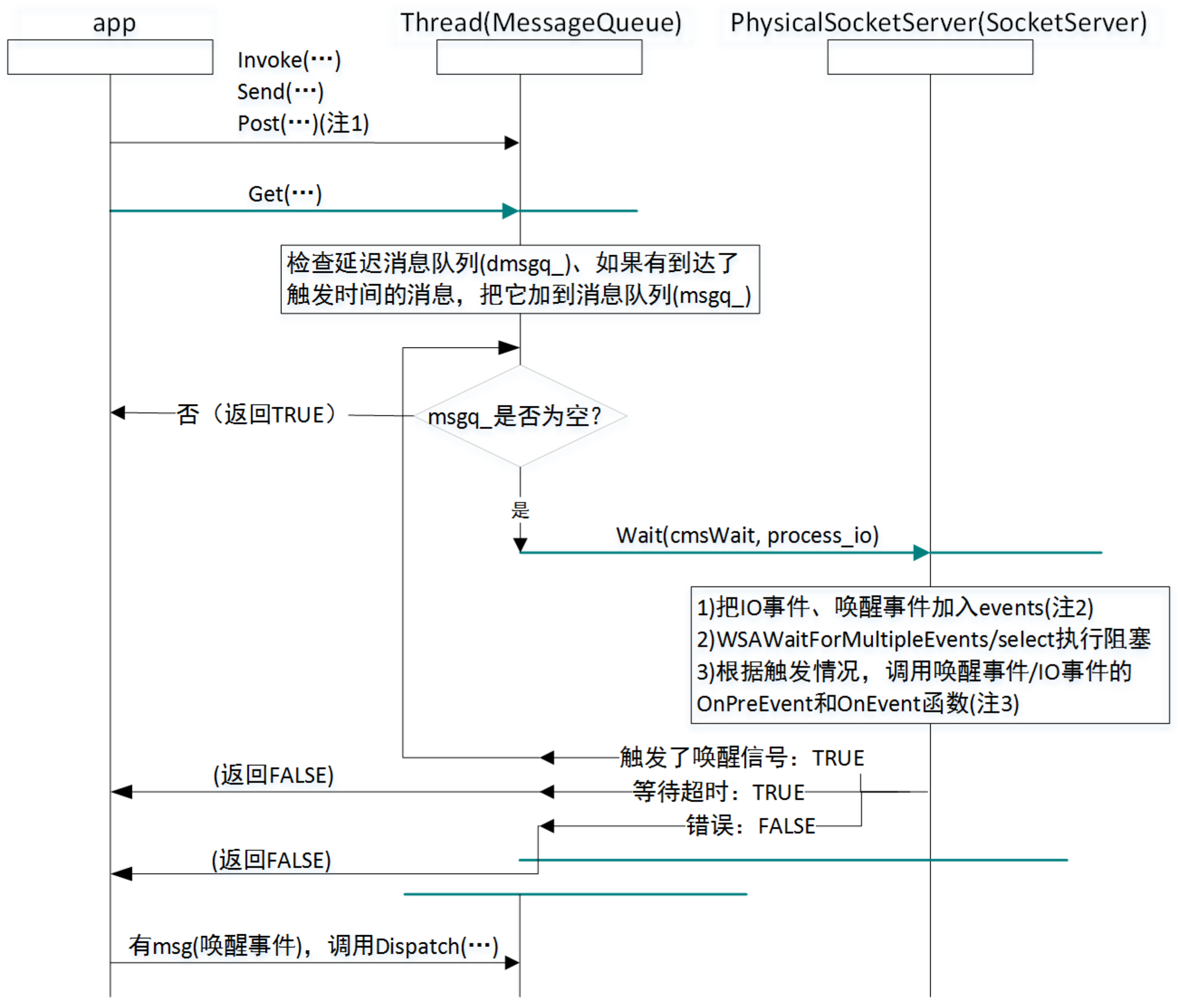
彩图 3 场景。《王国战争》关卡。栅格是正六边形，栅格坐标系值等于单元坐标系。



彩图 4 对话框。聊天界面。

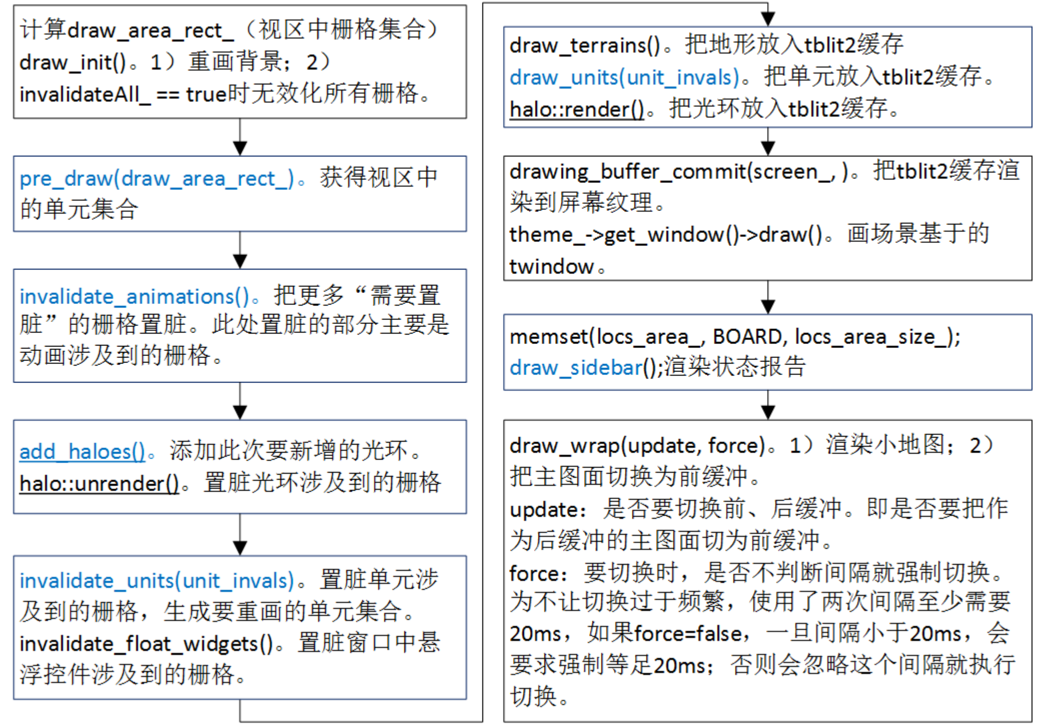


彩图 5 单元层。

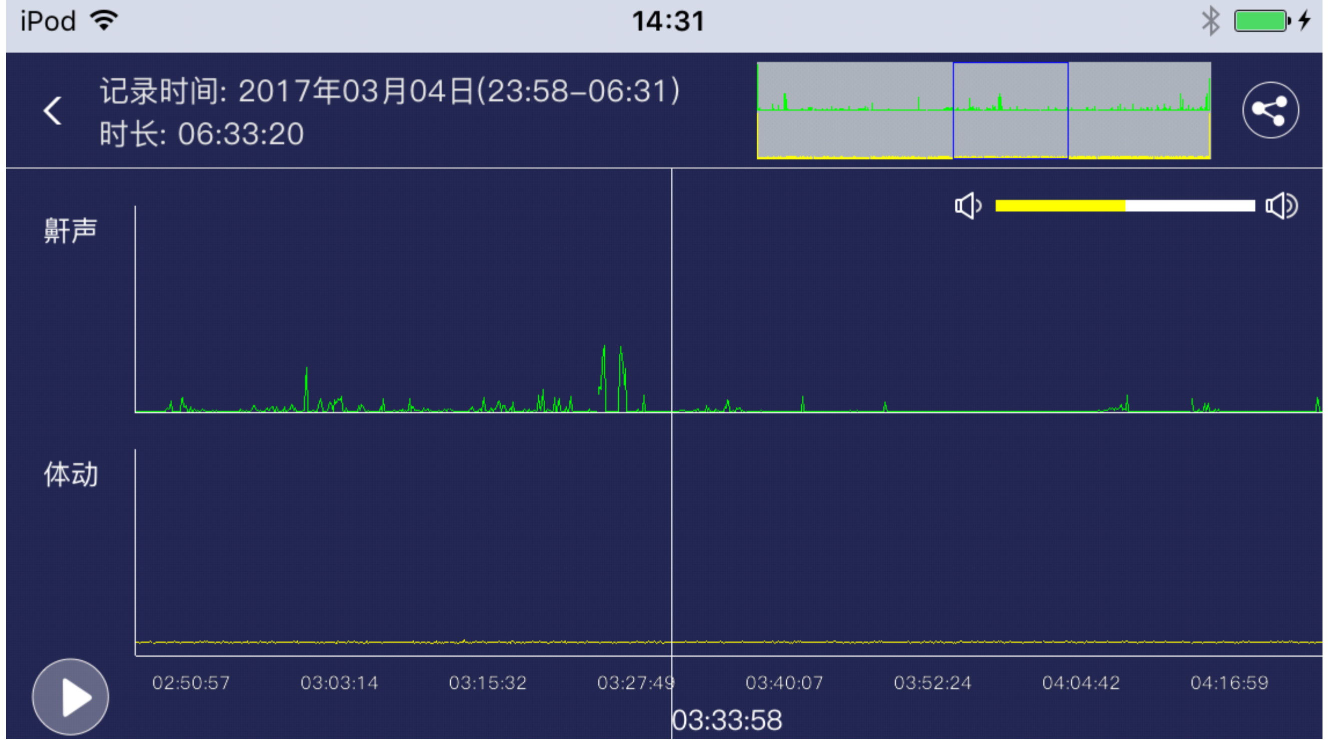


(注1)不是只有app才会调用Invoke等函数，webrtc内部就经常调用它们，但不论哪种情况，都是在Thread的上下文执行。
(注2)events[0]是IO事件，每次要阻塞前会调用events.push_back(socket_ev_)去占掉位置0
(注3)对唤醒事件，包括Invoke、Send、Post和工作者线程，还不清楚OnPreEvent、OnEvent要执行什么，处理它们主要是通过后面的Dispatch，这类消息会附带个Message结构，然后由当中的MessageHandler去处理。对IO事件，像socket的读，它是通过OnPreEvent、OnEvent去执行，也就是这两函数执行了具体操作，也意味着处理IO事件的线程就是此个ss_所在的Thread。

彩图 7 Webrtc 的线程模型/多路事件分离器



彩图 8 display::draw



彩图 9 场景。地图栅格是正方形，栅格坐标系值等于单元坐标系。图中两条横线、中间竖线、竖线旁边的“03:33:58”都是光环。

前言

让 PC、手机、Pad、家用机器人用同一个 SDK。

一、为什么要开发 Rose

随着硬件触摸技术、各种 Sensor 的成熟以及整体微处理器能力的提升和小型化，使 2007 年末开始移动时代。现在距 2007 过去十多年，很多迹象指向同一个结论：移动 app 的高速增长已经饱和。可技术不会停滞不前，用户对 app 的功能需求不会有尽头，下一阶段 app 增长点会是在哪里？——人工智能。近几年以深度学习为主力掀起了一波人工智能浪潮，不得不承认，这波浪潮在很多地方被吹嘘过头了。但是，此波 DNN 技术的确给机器视觉领域带来变革，让识别可靠性大幅提升，如果把这个提升应用到移动 app，势将在不少行业触发新增长点。

除进入传统 app，人工智能又一领域是机器人，而且会有人认为机器人才是人工智能应该待着的地方。新事物要能成功，是循序渐进，是水道渠成，一旦用的新技术太多，往往导致的是产品功能达不预期，公司进退两难。硬件上，像工艺、成本；软件上，像算法、多模块融合，一个“完整”机器人涉及到太多技术。可以肯定机器人将来会是主流，但对于市场，可能只是在已有成熟技术基础上加些许创新技术的产品会成功。保守观点，人工智能技术依旧须要时间发展自已，不要夸大它能力，相应地要谨慎评估机器人市场。

不论怎么说，传统 app 嵌入人工智能是趋势，机器人 app 门槛只会不断被突破。那如何让开发者编写这些 app？时间已证明，不论 PC 还是手机，一个好的 SDK 会深刻影响 app 开发的普及度。但相比传统功能，人工智能编程须要的已不是一、两个 SDK，而是多个 SDK 形成的工具链。另外，操作系统极可能不自带这些 SDK，或由于操作系统过多，开源 SDK 功能又和操作系统自带的不相上下，种种都导致开发者会选择开源 SDK。就近几年来说，TensorFlow 加 OpenCV 将会是 DNN 的主流解决方案。

一个工具链如果实现以下功能会是怎样?1）同时支持开发传统 app（包括游戏和非游戏）、机器人 app。2）跨平台，同时能跨 PC（Windows、Linux、Mac os X）、移动（iOS、Andoird）、以及机器人。3）针对布局窗口，不论算法、代码编写都有着媲美 Web 的方案。4）可无缝融合并内置开源社区中项目，像 Webrtc、OpenCV、TensorFlow Lite、Chromium。——市面上没这么个工具链，可没有不代表不能实现，于是希望 Rose 能成为这样的一个工具链。

二、Rose 架构

Rose 是用于开发人工智能 app 的跨平台工具链，初看起来会非常复杂，但结合开源社区，会发现真正要写的模块就是 gui、场景和动画，至于其它的可内置开源项目。

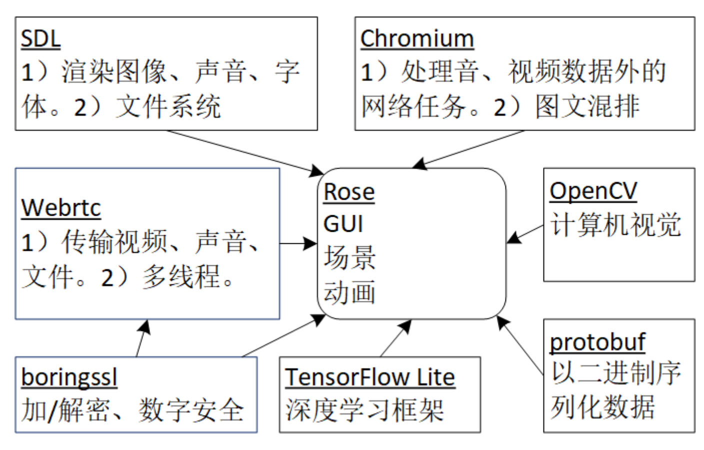


图 1 Rose 架构

Rose 有狭义、广义之分。狭义 Rose 指图 1 那个位于中心的模块，它实现窗口系统、场景和

动画，以及使用其它开源库的融合剂代码。广义 Rose 则包括图中的其它项目。从 Github 下载的是广义 Rose，也就是说，只要下载它就已包含 Webrtc、OpenCV、TensorFlow Lite、Chromium 等源码，不必再去官网下载了。

gui 和场景应该深度融合。有些开发者认为 gui 和场景（游戏引擎）应该分为两个 SDK，这里不得不要求他们打破这个陈旧观念，只有深度融合才能降低 app 开发难度，以及获得更好用户体验。

三、设计理念和三个阶段

上面已说了 Rose 的一个设计理念：同时支持开发传统 app（包括游戏和非游戏）、机器人 app。除了它，还要坚持 1）不用高深语法，提倡“C 加 class”。2）提供清晰的开发包目录结构，简化升级、维护以及同时开发多个 app。3）一个编程方向。4）高度模块化，能无缝融合开源社区中项目。5）一种 C/C++和脚本语言互补的解决框架。前两项不再补充，这里说下后三项。

什么是一个编程方向？在编程时，开发者会遇到这样事，要解决一个问题，A 这个 SDK 提供了 N 种方法，B 这个 SDK 只提供一种方法，按功能说，A 要更强，可实际 B 却更好。好的 SDK 永远是给开发者只提供一个编程方向，当中所有类也好，函数也罢，它们都是为这个方向服务，而开发者顺着这个方向都能编写出自个希望的 app。如果一个 SDK 只是函数的堆积、类的堆叠，不管具体类、具体函数功能有多强，那这个 SDK 还是会失败。有人会反驳，设计者怎么可能预知开发者会面对什么情况，为全面，自然是要提供多种选择。——如果坚持多方向论，那是设计者偷懒。探索并抽像出方向是 SDK 计设者的主要工作，时间是让这个方向更准确，而不只是堆积更多函数、更多类，后者带来的不是质的提升，而是臃肿。

用 C/C++编程，已具备融合第三方库的先决条件，但无缝融合同时，如果还能或多或少简化第三方库工作量，那自然会更好。要实现这目标，一个重要的任务是 Rose 要提供用 C/C++写的、平台无关的 gui。市面流行操作系统没几个，可实现 gui 时差异化非常严重，这就造成一个问题，如果第三方库和界面相关，如何简化它的 api？为每个操作系统编写界面，这对绝大多数开发者来说太苛刻，于是不写界面，提供文档告知 app 如何一步步调用 api。提供文档是最终能解决问题，可这一来须要 app 开发者有相关技术，二来意味着要投入更多时间和精力，面对当今这个以应用为导向的开发氛围，这越来越难能可贵了。不愿深入开发文档，这不能全怪 app 开发者，是环境变了。此时跨平台库若能提供一个独立于平台的 gui，而且还是 C/C++写，那对编写第三方库来说自然是个大好消息。

对 C/C++和脚本语言互补的重要性，有句简单但不失深度的话：C 写系统，C++写框架，脚本语言写业务。如何做到互补，首先脚本语言要能和 C/C++几近零成本互调。在这里举近些年流行的 Web 解决方案例子，它们都可认为基于 javascript，而开源社区中项目，但凡涉及到多些数据处理、数值计算，逃不开用 C/C++。可由于 javascript 和 C/C++间的语言障碍，不得不导致了 C/C++写的库须要很多处理才能融入 javascript 框架，这个处理不仅仅是时间，可能还存在难以解决的技术问题。

在互补框架中，有的 api 是 C/C++，有的是脚本语言，根据 api 中 C/C++、脚本语言比重，可把 Rose 分为三个阶段。第一阶段，api 全是 C/C++。第二阶段，C/C++占绝大多数。脚本语言提供插件式服务。什么是插件式服务？举个四对四团队对战的例子，C/C++写出对战逻辑，脚本去写各种地图。第三阶段，脚本语言不仅提供插件式服务，还不断封装本应是 C/C++接口的 api。一旦到了这个阶段，app 绝大部分都将是脚本代码，只有些非常独特功能才需要写 C/C++。

选择哪种脚本语言？除要能和 C/C++几近零成本互调，还要具备 2）既是 C/C++的“扩展语言”，又是“可扩展语言”。第一种形式中，C/C++拥有控制权，脚本是一个库。在第二种形式中，脚本拥有控制权，C/C++是一个库。3）只需少量 C/C++代码就能解释脚本语言。4）语法要类似 C/C++。如果只是针对 1、2、3，Lua 是种较好方案，但对它的语法，至少我实在是有点那个。要求语法类似 C/C++，个人较喜欢 php。

目前处在第一阶段。显而易见，不论哪个阶段，C/C++框架都是根基，都是要不断完善的，

“根基不牢、地动山摇”。另外，设计再好的脚本语言 SDK 都不能写出百分百性能 app，因而即使到了第三阶段，也须要向开发者同时提供 C/C++写的框架代码，至于开发者要不要用是另一回事。为此 Rose 会始终坚持下载开发包就同时下载了 C/C++框架源码。

四、3D

硬件发展会促进软件技术的革新。上世纪 90 年代，调色板表示图像中像素，现在 4 分量、32 位 ARGB 已是标配，调色板技术淡出了历史。一直到 2010，移动设备很少能支持视频聊天，现在已不再满足一对一，要设计 3 人、4 人会议了。同样，硬件性能提升也使 app 广泛使用 3D 成为可能。

app 如何使用 3D，很多人第一想到 3D 游戏，AR、VR、MR 种种和“现实”相关的使用场景。相比这些具体应用，3D 更大影响是会从根基上改变 app，这个根基就是图形用户界面。从 Google 推出“Material Design”，到微软的“Fluent Design System”，人机交互将慢慢从 2D 发展到更接近自然的 3D。Light（光照）、Depth（深度）、Motion（动态）、Material（材质）、Scale（伸缩性），随着时间发展，它们会变得更具体，更实用。

Rose 目前没支持 3D，但已在大范围使用 3D 的基础，硬件渲染技术。在实现架构上，SDL 处理各种技术，像 DirectX、OpenGL、OpenGL Es，以及可能流行的 Vulkan，它们被以着统一的 C API 提供给上层。Rose 使用 SDL 提供的渲染 API，或直接使用或进行一定的 C/C++封装，然后去实现希望的操作。或公司或个人开发，已有不少的 3D 库，但由于这个那个原因难以推广，个人建议遵循“C 写系统，C++写框架”，把底层 C 的部分融入 SDL。

五、项目进度和建议

在 Windows、iOS、Android，用 Rose 开发非 3D App 已没啥问题了。至于其它平台，跨平台基于 SDL，SDL 已能很好支持 Linux、Mac OS X，Rose 没支持主要是要保持一个月更新一版 SDK，个人毕竟精力有限，只能先做用户相对较多的。

设计理念中有一条是“不用高深语法，提倡‘C 加 class’”。不过一些 Boost 库中组件和新语法（C++ 11）的确在简化编程同时又没增加多少复杂度。Boost 以源码形式被融入 Rose，考虑到它的大量文件会增大容量，Rose 只包括了本 SDK 须要部分，自个喜欢组件没有被包括，把它们加入到相应位置即可。对 Boost，Rose 比较明确会用到三个组件。1）boost::bind 和 boost::function，让实现变参回调。2）boost::iostreams::gzip_compressor 等，实现数据压缩、解压缩。3）boost::regex，处理正则表达式。对新语法，被广泛使用一个是 C++11 的 std::unique_ptr。

Rose 真正做到写一次代码、跨平台运行。这使得代码只要在一个平台通过，基本可认为其它平台也没问题了。一些平台会碰到调试难问题，像 Android，由于这个特色，其它功能强大的平台已完成代码调试，到它基本就是编译下而已。Visual Studio 无疑有着最强大的 C/C++调试器，顺理成章成为主推的开发工具。假设一个月要发布一新版本，基本可做到 25 天都在 Windows 开发，2 天花在其它平台（就是编译），剩下 3 天做发布前测试。

看书顺序可以是第一章、第三章，看完第三章后就按自个重点选择看特定章节。

第一章 建立调试环境

本章目标

- Windows 系统，用 Visual Studio 编译 SDL、studio.exe。
- Mac OS X/iOS，用 XCode 编译 SDL、studio.app。
- Android，用 Android Studio + NDK 编译 SDL、studio.apk。

本书注重实践，每讲到相关部分都需要读者以代码去验证，进而理解，为此必须要先建立可调试环境。读者不能跳过这一章，看完本章后应该已可以修改代码，并运行自己修改过的程序。本章会介绍四个操作系统（Windows、Mac OS X、iOS、Android）下调试环境，读者可以选择略过后三个，但不能略过第一个，本书其它章节都可说是基于 Visual Studio。

为方便使用 Rose 开发 app，提供了一个叫 Studio 的应用，app 标识“studio”。Studio 基于 Rose SDK，它的代码对如何使用 Rose 有着极好参考价值，编译 Stduio 需要两个步骤，第一步编译 SDL；第二步编译 app。可以这么说，只要是基于 Rose 写的 app，它们第一步都是相同的，不同的只是第二步。

1.1 使用开发包

1.1.1 基本开发流程

步骤一：下载 Rose、SDL

Rose：https://github.com/freeors/Rose。

SDL：https://github.com/freeors/SDL。下载 Rose 时已包括 SDL 相关库的生成文件，如果你不想深入这些库，像要在哪库中设断点，可以不下载它。

放置时请注意 Rose 和 SDL 的位置，参考图 1-1。

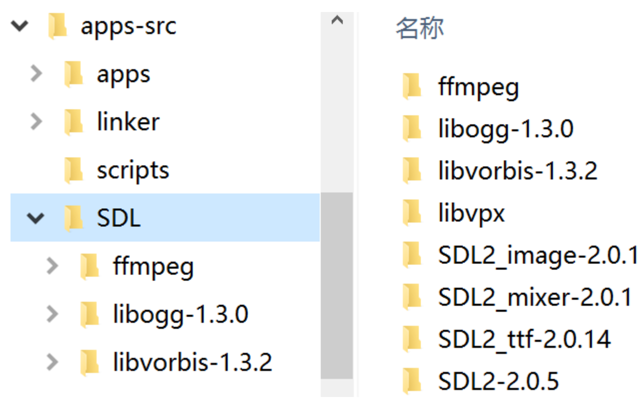


图 1-1 Rose 和 SDL 的位置

步骤二：编译 Studio

下载 Rose 后，默认就有 Studio 这个 app。请先编译 Studio。如何在各平台编译见“1.2、1.4、1.5”小节。

步骤三：在 Windows 系统，向工作包新建 app

下载 Rose 会在本地生成 apps-res、apps-src，它们合称工作包。接下在 Windows 运行 Studio 新建 app，新建操作是在左侧工作包的顶结点按鼠标右键，弹出菜单中选“新建 app ...”，以下假设新建的 app 是“hello”。

编译该 app 的 Windows 版本，让能成功运行。

步骤四：开发 app 的 iOS、Android 版本

运行 Studio，导出“hello”包，成功后会在工作包的同级目录生成“hello-res”、“hello-src”。
依照 1.3 节，编译 iOS 上的 hello.app。
依照 1.4 节，编译 Android 上的 hello.apk。

步骤五：app 私有资源

“hello”逻辑很简单，不需要私有资源，但一般 app 都会需要。新建 app 时已创建了存放私有资源的四个目录。私有资源更多细节参考“3.4 资源包”。

1.1.2 升级 Rose

下载新版 Rose，编译出“Studio”。运行“Studio”，在左侧工作包的顶结点按鼠标右键，弹出菜单中选“导入 app…”，弹出窗口选择旧版本的资源包目录，执行“导入”。导入结束，升级也就完成了。

升级不会改变 app 私有数据、源码，因而如果新版 Rose 对 API 接口有变动，app 又恰好用了那 API，app 需要手动改相关代码。

1.1.3 资源包和源码包

开发包包括资源包和源码包，它们分别对应一个目录。在命名上，资源包格式是<appid>-res，源码包是<appid>-src，当中的“appid”是 app 标识。

资源包是 app 运行时需要的资源，包括图像文件、声音文件、字体文件，各语言 pot、po 文件，及各样配置文件，像地形、兵种、战役、窗口。

源码包是源代码，包括 C/C++的*.c/*.cpp/*.h/*.hpp，及各平台下工程文件。

除非很熟悉 Rose 包结构，请把资源包和源码包放在同级目录。原因是 Studio 在处理开发包时默认认为它们在同级目录。

要在 Visual Studio 直接运行 app 就需要知道资源包存放地址，因此资源包放在哪里会影响如何设置 Visual Studio 中 app 工程的命令行参数。具体就是要设置正确的“Debugging”——“Command Arguments”，关于设置命令行参数参考“**1.2.4 运行**”。

1.1.4 工作包和 app 包

Rose 支持同时开发多个 app，按处在开发中的哪个阶段，开发包分为工作包（Work Kit）和 app 包（app Package）。工作包包含同时在开发的多个 app，它在开发、调试时使用。工作包中所有的 app 共享一个 Rose 库，当然，你也可以把自个的共享库放在 external，以方便修改、维护代码。app 包则只有一个 app，当 app 已基本稳定可以发布时，用 Studio 从工作包导出该包，它将用于发布。具体操作是在左侧树形控件的“apps-res”单击右键，弹出菜单中选择要导出的 app，如图 1-2 所示。

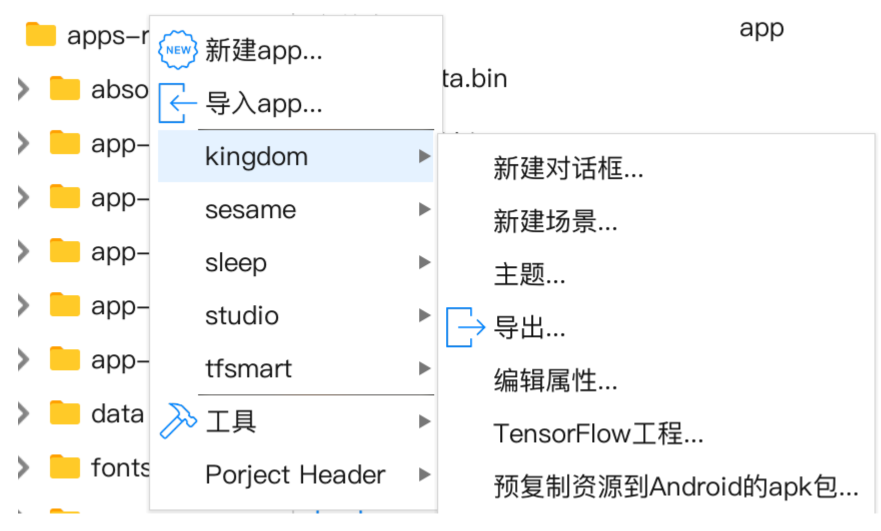


图 1-2 Studo 提供的包处理命令

为方便工作包、app 包之间的导出、导入等操作，Studio 提供了一系列命令。

“导入 app”。它有两个作用，一是升级 Rose，具体见“**1.1.2 升级 Rose**”。二是从他人提供的 app 包中导入 app。

“新建对话框（app 相关）”。它会向资源包、源码包增加该对话框需要的相关文件。

“新建场景（app 相关）”。它会向资源包、源码包增加该场景需要的相关文件。

“主题（app 相关）”。方便 app 增加、删除、修改主题。

“导出 app 包（app 相关）”。从工作包导出该 app 包。注:这命令会在 <app>/src/main/assets/res 目录下生成“__cookie.cki”哑文件。

“编辑属性（app 相关）”。修改 app 属性，这些属性会影响 app 的工程文件。举个例子，一旦打开蓝牙外设，生成的 iOS 工程文件会包含蓝牙权限，同样的，也修改 Andorid 工程。

“导出 kOS 安装包（app 相关）”。从工作包导出该 app 用于安装 kOS 的.ksa 文件。

“创建 iOS 工作包（工具）”。虽然内容一样，但 iOS 工作包的目录结构和 Windows 稍有不同。执行这命令后，生成的 iOS 工作包放在同级的 ios_kit 目录。

1.1.5 app 私有源码、include 目录

以 aismart 为例，源码包中的 aismart 目录存放着 app 所有私有源码，在它之下默认会有 gui/dialogs 目录存放着处理窗口的代码。除了它们，有的 app 须要新建目录，像 aismart 通过内置 easypr 完成车牌识别，于是把整个 easypr 源码包复制到 aismart。一旦私有源码出现除 aismart、gui/dialogs 外的目录时，要规范 Visutal Studio 书写 include 目录的格式。

这么做的原因是为了生成非 Windows 系统的工程文件时能自动增加这些额外的 include 目录。具作是在向“Additional Include Directories”添加时，确保让它们的前缀是../../aismart。根据这规则，仍以 aismart 为例，对 easypr 要求的两个 include 目录写成下面格式。

```
../../aismart/easypr
../../aismart/easypr/include
```

新建了目录，底下会有 cpp 文件。studio 会搜索那些目录，然后认为找到的*.c、*.cc、*.cpp 都是要编译的。

建议 app 不要有额外的编译、链接选项，像自个的预定义宏。因为这意味着你得在 iOS、Android 工程文件手动添加它们。

1.1.6 editor.exe

Studio 用于构造各个*.bin，而 Studio 整个运行都依赖这些*.bin，一旦出现构造*.bin 的过程中出现意外，像强制退出了，就会出现破坏了的*.bin，结果就是 Studio 也运行不了。为解决这问题 Rose 提供了 editor.exe。

editor.exe 只能运行在 Windows，它运行不依赖*.bin。你可用它去构造*.bin，自然包括修复那些破坏的*.bin。

1.2 Visual Studio 2017（Windows）

必须 Visual Studio 2017 或以上。配置管理器中的 Configuration 选择 Release，Platform 选 Win32。可以换到 Debug、x64，但那需要做非常多的设置，除了搜索路径、预定义宏等等，还要为一些文件单独设置编译选项，像*.asm。虽然是 Release，但修改了标准 Release 选项，使产生调试符号文件以支持断点。

Rose 库有汇编文件，编译它们须安装 yasm，yasm 官网：http://yasm.tortall.net/。一种安装方法是下载 vsyasm.exe（针对 Visutal Studio 的包中， yasm.exe 文件名是 vsyasm.exe），然后把 vsyasm.exe 复制 Visual Studio 的 bin 路径，类似 C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise\VC\Tools\MSVC\14.12.25827\bin\Hostx64\x86。

x86(win32)的*.exe 不能使用 x64(win64)编译出的*.lib，x64(win64)的*.exe 不能使用 x86(win32)

编译出的*.lib。以上 exe 选择了用 win32，意味着编译相关 dll、lib 都要选 win32。和 exe 不同，dll、lib 都支持 Debug 、Release，不过它们切到 Debug 的方法分为两类。

（1）opencv、protobuf、tensorflowlite、chromium，Configuration 还是 Release，修改配置，1）C/C++——General。Debug Information Format 改为“Program Database(/Zi)”，2）C/C++——Optimization。Optimization 改为“Disabled(/Od)”。

（2）以上之外的库，直接在 Configuration 切到 Debug。

1.2.1 目录结构

除非已经了解为什么要这么设置目录，否则不要去修改要求的目录结构。这个要求包括之后要介绍的 Mac OS X、iOS、Android 平台。

下面以编译 studio 为例描述编译过程。

名称	修改日期	类型
apps	2016/6/24 21:18	文件夹
linker	2016/6/27 16:52	文件夹
scripts	2016/6/26 11:33	文件夹
SDL	2016/2/28 10:41	文件夹

图 1-3 顶层目录结构

- apps：源码目录，以及编译、链接时生成的文件。Studio 是个有点特殊的 app，它要和其它 app 并存，因而用了工作目录的目录名 apps，其它的 app 一般是 app id，像 tfsmart。
- SDL：多媒体开发库 sdl，从 https://github.com/freeors/SDL 下载。
- linker：编译、链接时须要的头文件、库文件。它包括两部分，一是 SDL，二是除 SDL 外的第三方开发包，像微信、QQ。
- scripts：为方便编译过程而写的脚本文件。Windows 当前没有脚本，因而可以不须要这目录，放在这里是想说，如果你有脚本建议放在这里。

总的来说，开发包须要单独编译的分两个部分：sdl 和 app。以下就按这个分类分别描述如何编译。

1.2.2 SDL

下载 Rose 后，已把须要的 SDL 开发库放在<linker>/windows，如果你不想重编译它们，可以略过这部分。注：默认是 Release 版，无法单步调试，要调试需编译 Debug 版。

名称	修改日期	类型
libogg-1.3.0	2015/8/5 15:23	文件夹
libvorbis-1.3.2	2015/3/1 21:06	文件夹
SDL2_image-2.0.1	2016/3/27 22:19	文件夹
SDL2_mixer-2.0.1	2016/3/27 22:21	文件夹
SDL2_net-2.0.1	2016/3/27 22:21	文件夹
SDL2_ttf-2.0.14	2016/3/27 22:21	文件夹
SDL2-2.0.4	2016/6/26 16:32	文件夹

图 1-4 SDL 目录结构

SDL 又分数个库：

- SDL2-2.0.5：SDL 主库 sdl2.dll。
- SDL2_image-2.0.1：SDL 图像辅助库 sdl2_image.dll。
- SDL2_mixer-2.0.1：SDL 音频辅助库 sdl2_mixer.dll。
- SDL2_ttf_2.0.14：SDL 字符串渲染辅助库 sdl2_ttf.dll。
- libvpx：不属于 SDL。https://github.com/webmproject/libvpx。

- libogg-1.3.0：Vistual Studio 时忽略。
- libvorbis-1.3.2：Vistual Studio 时忽略。

SDL 是个开源库，源代码、lib、dll 都可从 sdl 官方网站上下载。但这里还是选择了重新编译，这是考虑到几个原因。

1. 要调试。直接下载的库没法设置断点，即没法进行源码级跟踪。
2. Release 版时，编译选项 Runtime Library 要改成 Multi-threaded(/MT)。官网下载的一般是 Multi-threaded DLL(/MD)。
3. 进行源码级调试可以更好地理解 SDL 内部是如何工作。
4. 对官方 SDL 库进行过修改。有兴趣的可以对比这里下载的和官方版本，改了几处。

编译 **SDL2.dll**。打开并编译<SDL>\SDL2-2.0.5\VisualC\SDL.sln。要确认编译是否成功，可以察看<linker>\windows\dll 下是否生成了一个新的 SDL2.dll 文件。

编译 **SDL2_image.dll**。打开并编译<SDL>\SDL2_image-2.0.1\VisualC\SDL_image_VS2010.sln。要确认编译是否成功，察看<linker>\windows\dll 下是否生成了一个新的 SDL2_image.dll 文件。

注意：SDL_image 工程除了 SDL_image 项目外还有 showimage，编译时不必管这个项目，只要 SDL_image 这个项目成功就行。

编译 **SDL2_mixer.dll**。打开并编译<SDL>\SDL2_mixer-2.0.1\Visual\SDL_mixer.sln。要确认编译是否成功，察看<linker>\windows\dll 下是否生成了一个新的 SDL2_mixer.dll 文件。

注意：SDL_mixer 工程除了 SDL_mixer 项目外还有 native_midi、playmus、playwave、timidity，编译时不必管那些个项目，只要 SDL_mixer 这个项目成功就行。

编译 **SDL2_ttf.dll**。打开并编译<SDL>\SDL_ttf-2.0.14\VisualC\SDL_ttf_VS2010.sln。要确认编译是否成功，察看<linker>\windows\dll 下是否生成了一个新的 SDL2_ttf.dll 文件。

注意：SDL_ttf 工程除了 SDL_ttf 项目外还有 glfont、showfont，编译时不必管那些个项目，只要 SDL_ttf 这个项目成功就行。

编译 **ffmpeg.dll**。打开并编译<SDL>\ffmpeg\projectfiles\vc\ffmpeg.sln。要确认编译是否成功，察看<linker>\windows\dll 下是否生成了一个新的 ffmpeg.dll 文件。

编译 **opencv.dll**。打开并编译<SDL>\opencv\projectfiles\vc\opencv.sln。要确认编译是否成功，察看<linker>\windows\dll 下是否生成了一个新的 opencv.dll 文件。

编译 **openh264.dll**。打开并编译<SDL>\openh264\projectfiles\vc\openh264.sln。要确认编译是否成功，察看<linker>\windows\dll 下是否生成了一个新的 openh264.dll 文件。

编译 **libprotobuf.lib**。打开并编译<SDL>\libprotobuf\projectfiles\vc\libprotobuf.sln。要确认编译是否成功，察看<linker>\windows\lib 下是否生成了一个新的 libprotobuf.lib 文件。注：libprotobuf 是以静态库方式链入最终 app。

编译 **libvpx.lib**。打开并编译<SDL>\libvpx\projectfiles\vc\libvpx.sln。要确认编译是否成功，察看<linker>\windows\lib 下是否生成了一个新的 libvpx.lib 文件。注：libvpx 是以静态库方式链入最终 app。

编译 **tensorflowlite.lib**。打开<SDL>\tensorflowlite\projectifiles\vc\tensorflowlite.sln，“Configuration”选“Release”，“Platform”选“Win32”。编译，要确认编译是否成功，可以察看<linker>\windows\lib 下是否生成了一个新的 tensorflowlite.lib。注：libtensorflowlit 是以静态库方式链入最终 app。

编译后形成的*.lib 和*.dll 都被统一存放在<linker>\windows 目录。app 通过<apps>和<linker>目录关系找到 liniker，进而链接时链接到相应的*.lib，链接后自动把相应 dll 复制到可执行程序 studio.exe 所在目录下。

1.2.3 studio.exe

名称	修改日期	类型
external	2016/6/26 22:09	文件夹
librose	2016/6/26 22:09	文件夹
projectfiles	2016/6/26 22:09	文件夹
studio	2016/6/26 22:09	文件夹

图 1-6 studio 目录结构

- external：rose 库源码分为两部分，它是当中一个。存放要没意外不要去改的源码，一般是比较稳定了的开源包，像 boost、bzip2、zlib、gettext。它被内嵌的方式链入 app。
- librose：rose 库源码分为两部分，它是当中一个。它被内嵌的方式链入 app。
- projectfiles：子目录 android 对应 Android 时工程文件，及编译后形成的*.obj、studio.apk。studio：studio 这个 app 的私有源码。

编译

编译 **chromium.lib**。打开<SDL>\chromium\projectifiles\vc\chromium.sln，“Configuration”选“Release”，“Platform”选“Win32”。编译，要确认编译是否成功，可以察看<linker>\windows\lib 下是否生成了一个新的 chromium.lib。

编译 **studio.exe**。在 Visual Studio 打开 apps.sln，“Build”——“Build Solution”。

1.2.4 运行

一旦以上全部编译成功，<projectfiles>\vc\Release 目录会出现以下文件：

- studio.exe。
- ffmpeg.dll、SDL2.dll、SDL2_ttf.dll、SDL2_mixer.dll、SDL2_image.dll。

除了以上还须要复制其它开源库，可从相应官网上下载或取自资源包（<apps-res>）根下，然后复制到 Release 目录。

libjpeg-9.dll、libfreetype-6.dll、libogg-0.dll、libpng16-16.dll、libtiff-5.dll、libvorbis-0.dll、libvorbisfile-3.dll、libwebp-4.dll、mikmod.dll、smpeg.dll、zlib1.dll。

运行 studio.exe

运行前要先设置 studio 项目命令行参数：“Configuration Properties”——“Debugging”——“Command Arguments”。命令行参数须正确指示资源包存放在哪里。

Local Windows Debugger

Command	$(TargetPath)
Command Arguments	**--res-dir "..\..\..\..\apps-res"**
Working Directory	$(ProjectDir)
Attach	No
Debugger Type	Auto

图 1-7 设置 studio.exe 命令行参数

```
资源包和源码包同级放置时命令行参数
--res-dir "..\..\..\..\apps-res"

资源包放在“c:\apps-res”时命令行参数
--res-dir "c:\apps-res"
```

- 把 studio 设置为“Set as Startup Project”。
- “Debug”——“Start Debuging”或“Start Widthout Debuging”，即可运行 app。

注：

命令行参数写在<projectfiles>\vc\studio.vcxproj.user，但该文件同时包含了调试信息，像工程在硬盘上位置。位置等信息要是不能反映真实情况，例如项目被挪到其它目录，会使得“Debug”——“Attach to Process”到“它”时变得很慢，可能由 2 或 3 秒扩大为十多秒。一旦项目换了位置时，最好删掉*.user，以便让 Visual Studio 重新生成。

1.2.5 向工作包新建 app

运行 Studio，在左侧工作包的顶结点按鼠标右键，弹出菜单中选“新建 app ...”。

1.3 XCode 4.2（Mac OS X）（已过时）

1.3.1 目录结构

- 除非已经了解为什么要这么设置目录，否则不要去修改要求的目录结构。
- 发布时所有工程的 Active Configuration 选择 Release。调试时可按需把 Active Configuration 置为 Debug。macosx_2_kingdom.sh 支持的是 Release 版本。

《王国战争》项目包括数个工程，为方便整个项目上的编译、链接，在整体上编为数个目录。

名称	修改日期	大小	种类
gettext	今天, 下午12:09	--	文件夹
kingdom	今天, 上午11:39	--	文件夹
postbuild	今天, 下午12:28	--	文件夹
SDL	昨天, 上午7:09	--	文件夹

图 1-8 顶层目录结构

- gettext：国际化支持库 gettext。
- kingdom：主应用程序 kingdom.app。
- SDL：多媒体开发库 sdl。
- postbuild：macosx_2_kingdom.sh 用于把编译生成的各个 framework 复制到主项目 kingdom 指定的位置。

总的来说，《王国战争》源码分三个部分：两个支持库，sdl 和 gettext；主应用程序 kingdom.app。以下就以这些分别说下如何建立开发环境。

1.3.2 SDL

名称	修改日期	大小	种类
freetype-2.3.9	12-1-23	--	文件夹
libogg-1.3.0	12-1-23	--	文件夹
libvorbis-1.3.2	11-8-30	--	文件夹
SDL_image-1.2.12	今天, 上午10:56	--	文件夹
SDL_mixer-1.2.12	12-1-28	--	文件夹
SDL_net-1.2.8	12-1-28	--	文件夹
SDL_ttf-2.0.11	12-1-28	--	文件夹
SDL-1.3.0-6217	12-2-24	--	文件夹
SDL.framework	12-2-24	--	文件夹
SDL.framework.iPhoneOS	12-1-16	--	文件夹
smpeg	11-8-30	--	文件夹

图 1-9 SDL 目录结构

SDL 分数个库：

- SDL-1.3.0-6217：SDL 主库 SDL.framework。
- SDL_image-1.2.12：SDL 图像辅助库 SDL_image..framework。
- SDL_mixer-1.2.12：SDL 音频辅助库 SDL_mixer..framework。
- SDL_ttf_2.0.11：SDL 字符串渲染辅助库 SDL_ttf.framework。
- smpeg：解码 mp2/mp3 音频格式要用到的库，是 SDL_mixer.framework 辅助库。
- ____
- freetype-2.3.9：freetype 是个渲染字体库，但在此种它没作为一个独立库，源文件被汇到入 SDL_ttf 编译成 SDL_ttf.framework。
- ____
- SDL.framework：它就是 SDL 主库生成的 SDL.framework，复制出一份是为了编译辅助库时能找到 SDL 头文件。
- ____
- jpeg-8d：Mac OS X 时可忽略。
- lpng157：Mac OS X 时可忽略。
- libogg-1.3.0：Mac OS X 时可忽略。
- libvorbis-1.3.2：Mac OS X 时可忽略。。
- SDL.framework.iPhoneOS：Mac OS X 时忽略。

SDL 是个开源库，源代码、lib、dll 都可从 SDL 官方网站上下载。但这里还是选择了重新编译，这是考虑到几个原因。

1. 对官方 SDL 库进行过修改。有兴趣的可以对比这里下载的和官方版本，改了几处。
2. 要调试。直接下载的库没法设置断点，即没法进行源码级跟踪。
3. 进行源码级调试可以更好地理解 SDL 内部是如何工作。

编译 SDL.framework。打开并编译<SDL>/SDL-1.3.0-6217/Xcode/SDL/SDL.xcodeproj。要确认编译是否成功，可以察看<SDL>/SDL-1.3.0-6217/Xcode/SDL/build/Release 下是否生成了一个新的 SDL.framework。

把生成的 SDL.framework 复制到<SDL>/。

编译 SDL_image.framework。打开并编译<SDL>/SDL_image-1.2.12/Xcode/SDL_image.xcodeproj。要确认编译是否成功，可以察看<SDL>/SDL_image-1.2.12/Xcode/build 下是否生成了一个新的 SDL_image.framework。

编译 SDL_mixer.framework。打开并编译<SDL>/SDL_mixer-1.2.12/Xcode/SDL_mixer.xcodeproj。要确认编译是否成功，可以察看<SDL>/SDL_mixer-1.2.12/Xcode/build 下是否生成了一个新的 SDL_mixer.framework。

编译 SDL_ttf.framework。打开并编译<SDL>/SDL_ttf-2.0.11/Xcode/SDL_ttf.xcodeproj。要确认编译是否成功，可以察看<SDL>/SDL_ttf-2.0.11/Xcode/build 下是否生成了一个新的 libSDL_ttf.framework。

编译 smpeg.framework。打开并编译<SDL>/smpeg/Xcode/smpeg.xcodeproj。要确认编译是否成功，可以察看<SDL>/smpeg/Xcode/build 下是否生成了一个新的 smpeg.framework。

注：虽然 smpeg.frame 是作为动态链接库运行时被 SDL_mixer.framework 加载，但要编译 SDL_mixer.framework 可以不先编译 smpeg.frame。因为 smpeg.frame 也是动态链接库，发布时需要把 smpeg.framework 和 SDL_mixer.framework 一样放在资源包中被发布的。

编译后形成的*.framework 被放在各自目录，而要被主应用程序 kingdom.app 调用需复制到它能够找到的指定目录下，后绪会执行的 macosx_2_kingdom.sh 完成这些个复制。

1.3.3 gettext

名称	修改日期	大小	种类
gettext	今天, 下午12:10	--	文件夹
libiconv	2011年7月7日 上午10:57	--	文件夹
Xcode	2012年1月12日 下午4:39	--	文件夹
Xcode-iPhoneOS	2011年8月31日 下午2:23	--	文件夹

图 1-10 gettext 目录结构

gettext 工程形成 libintl.framework。GNU 是把 gettext 分两个项目：iconv 和 gettext。iconv 是个静态库 iconv.a，gettext 则是基于 iconv.a，不过我把它统一编译合为一个.framework 了。如果不改名，gettext 形成的二地制目标是 gettext.framework，不过使用时一般都把它改名为 libintl.framework，intl 是 international 缩写，指示它的功能是支持国际化。

- gettext：gettext 项目源代码。
- libiconv：iconv 项目源代码。
- Xcode：gettext 在 Mac OS X 下的工程文件。
- ______________________________
- Xcode-iPhoneOS：Mac OS X 时忽略。

编译。打开并编译<gettext>/Xcode/gettext.xcodeproj。要确认编译是否成功，可以察看<gettext>/Xcode /build/Release 下是否生成了一个新的 libintl.framework 文件。

1.3.4 kingdom.app

名称	修改日期	大小	种类
data	2012年2月17日 下午5:40	--	文件夹
external	2012年1月15日 下午3:57	--	文件夹
fonts	2011年6月12日 下午3:52	--	文件夹
images	2012年2月17日 下午5:40	--	文件夹
projectfiles	昨天, 上午7:17	--	文件夹
sounds	2011年11月26日 下午5:32	--	文件夹
src	今天, 上午10:42	--	文件夹
translations	2012年2月17日 下午5:40	--	文件夹
xwml	2012年2月17日 下午5:40	--	文件夹

图 1-11 kingdom 目录结构

- external：要被嵌入到工程的 Boost 库*.hpp/*.cpp 源码。
- projectfiles：存放 kingdom.xcodeproj 工程文件，及编译后形成的*.obj、kingdom.app 包。
- src：*.hpp/*.cpp 源码。
- ______________________________
- data、fonts、images、sounds、translations、xwml：来自发布时资源包，如何生成见“1.1.3 处理开发包”中的用编辑器生成发布时资源包。iOS 形成 kingdom.app 时要把这些内容打包。

编译

1. 打开<kingdom>/postbuild/macosx_2_kingdom.sh，针对你 PC 路径设置好正确$POSTBUILD 变量。

```
POSTBUILD=/Users/ancientcc/ddksample/postbuild
SDL_sdl=$POSTBUILD/../SDL/SDL-1.3.0-6217/Xcode/SDL/build/Release
SDL_image=$POSTBUILD/../SDL/SDL_image-1.2.12/Xcode/build/Release
.....
```

2. 运行终端，切换到<kingdom>/postbuild 目录，运行 macosx_2_kingdom.sh，该脚本会把以上编译生成的 framework 复制到 kingdom 指定位置。

```
cd ddksample/postbuild
./macosx_2_kingdom.sh
```

3. 打开<kingdom>/projectfiles/Xcode/kingdom.xcodeproj，编译。

1.3.5 运行

一旦以上全部编译成功，<kingdom>/projectfiles/Xcode/Release 目录下会出现 kingdom.app。在 Xcode IDE 中按多个 Run 命令中一个就可运行游戏；或进入资源管理器，双击 kingdom.app 就可运行游戏。

1.3.6 编译、链接选项，framework

Base SDK 需至少设置为 Mac OS X 10.6。SDL 库调用了 10.6 有而 10.5 没有的 SDK API。

framework

要被构造的各个库都是以着 framework 形式被发布。framework 封装开发工具包，以 Windows 观点来说，它是一个包括头文件、lib 和 dll 的目录。包内模块被应用程序调用的形式是动态链接库。

framework 类似 Windows 上 dll，像开发时需要显示给出头文件目录，发布时需要把相关二进制模块文件合进 release 包，但也存在和 Windows 很不一样地方。

编译时链接

以要编译 kingdom 为例，kingdom 要调用 SDL_mixer.framework 中函数，那程序在链接时怎么找到正确的 SDL_mixer.framework？对此可以有两种方法。

- “Build”的“Linking”——“Other Linker Flags”中添加“-frame SDL_mixer”。此种方法同时需要在“Search Paths”——“Framework Search Paths”中指定 SDL_mixer.framework 的存放路径。
- 在浏览器 Targets 的“Link Binary With Libraries”下指定 SDL_mixer.framework。为了能在 Targets 中出现，在源中要加入 SDL_mixer.framework（往往会创建个专门存放 framework 的组，像 Frameworks）。“Link Binary With Libraries”指定的是编译时需要的 framework，而不是最后打包时要被打进包的 framework，后者是由“Copy Files”实现。此种方法加入时已隐式指定 framework 路径，因此“Build”的“Framework Search Paths”选项就不必设置了。

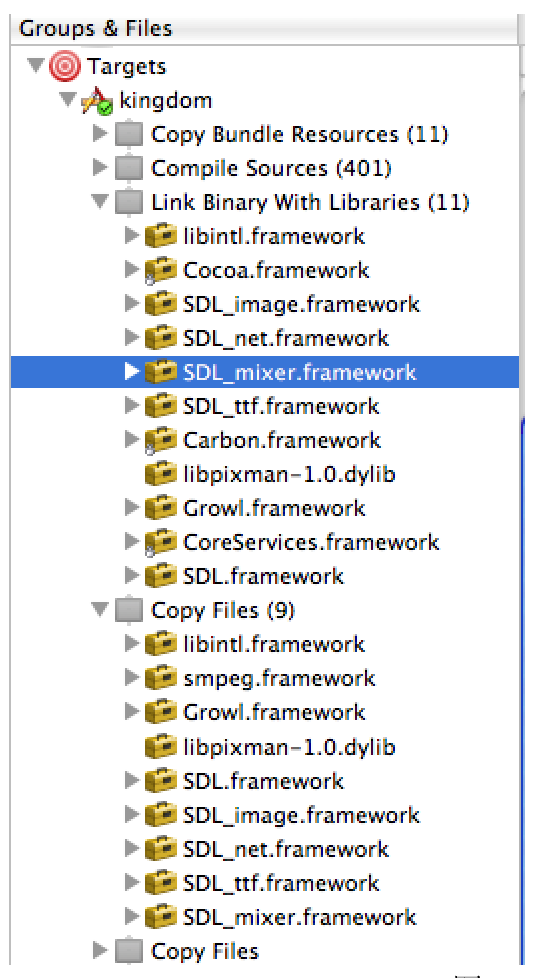


图 1-12 设定 framework 库链接

由于第二种较为直观，《王国战争》使用第二种方案。

运行时定位

以要动态链接 SDL_mixer.dll 为例，应用程序在运行时发现需要调用 SDL_mixer.dll，它是知道文件名叫 SDL_mixer.dll，但它怎么知道 SDL_mixer.dll 在文件系统哪个位置？这就是动态链接库定位问题。对于 Windows 来说，它是以一种硬性规则来执行(代码是非 LoadLibrary 的显式链接情况)，首先从包含 EXE 映象文件的目录下找，如果没有就从进程的当前目录中找，再没就从 Windows 系统目录中，再没就从……，framework 则把定位写在包中，“Deployment”——“Installation Directory”设置这个目录。

因为这个定位一般和可执行文件所在目录相关，为方便书写，Xcode 用@executable_path 这个变量指示可执行目录，在设置“Installation Directory”时可使用该变量，以下就是把包定位在可执行文件上级目录的 Frameworks 目录下。

Deployment	
Additional Strip Flags	
Alternate Install Group	staff
Alternate Install Owner	ancientcc
Alternate Install Permissions	u+w,go-w,a+rX
Alternate Permissions Files	
Deployment Location	☐
Deployment Postprocessing	☐
Install Group	staff
Install Owner	ancientcc
Install Permissions	u+w,go-w,a+rX
Installation Build Products Location	/tmp/SDL_mixer.dst
Installation Directory	@executable_path/../Frameworks
Mac OS X Deployment Target	Mac OS X 10.5
Skip Install	☐
Strip Debug Symbols During Copy	☑
Strip Linked Product	☐
Strip Style	Debugging Symbols
Use Separate Strip	☐

图 1-13 定位 framework 安装位置

1.4 XCode 9.3.1（iOS）

1.4.1 目录结构

Rose 包含不少和处理器相关汇编代码，编译出的 app 只支持真机，不支持模拟器。

虽然内容一样，但 iOS 工作包的目录结构和 Windows 稍有不同，为方便，Studio 有提供命令帮快速创建这目录，见“1.1.3 工作包和 app 包”中的“创建 iOS 工作包”。图 1-14 是一个目录示例，在同时开发 5 个 app：blesmart、kingdom、sesame、sleep、studio。

名称	修改日期	大小	种类
blesmart	2016年3月18日 下午10:51	--	文件夹
external	2016年4月11日 下午8:56	--	文件夹
kingdom	2016年2月6日 下午4:04	--	文件夹
librose	前天 下午3:44	--	文件夹
linker	2016年2月24日 下午9:09	--	文件夹
scripts	前天 上午11:33	--	文件夹
SDL	2016年4月12日 下午4:22	--	文件夹
sesame	2016年6月25日 上午10:37	--	文件夹
sleep	2016年5月29日 下午10:31	--	文件夹
studio	前天 下午7:39	--	文件夹

图 1-14 顶层目录结构

- blesmart、kingdom、sesame、sleep、studio：在开发的 5 个 app 的私有源码目录，它里面的内容见“**1.4.3 studio.app**”。
- external：rose 库源码分为两部分，它是当中一个。存放要没意外不要去改的源码，一般是比较稳定了的开源包，像 boost、bzip2、zlib、gettext。它被内嵌的方式链入 app。

- librose：rose 库源码分为两部分，它是当中一个。它被内嵌的方式链入 app。
- SDL：多媒体开发库 sdl。它会生成多个库，以库的方式汇入 app.
- linker：编译、链接时须要的头文件、库文件。它包括两部分，一是 SDL，二是除 SDL 外的第三方开发包，像微信、QQ。
- scripts：为方便编译过程而写的脚本文件。iOS 当前没有脚本，因而可以不须要这目录，放在这里是想说，如果你有脚本建议放在这里。

总的来说，开发包须要单独编译的分两个部分：sdl 和 app。以下就按这个分类分别描述如何编译。

1.4.2 SDL

下载 apps 包后，已把须要的 SDL 开发库放在<linker>/ios，如果你不想重编译它们，可以先略过这部分。注：默认是 Release 版，无法单步调试，要调试需编译 Debug 版。

名称	修改日期	大小	种类
libogg-1.3.0	15/10/5	--	文件夹
libvorbis-1.3.2	15/10/5	--	文件夹
SDL2_image-2.0.1	16/2/5	--	文件夹
SDL2_mixer-2.0.1	16/4/29	--	文件夹
SDL2_net-2.0.1	16/2/5	--	文件夹
SDL2_ttf-2.0.14	16/2/5	--	文件夹
SDL2-2.0.4	16/6/26	--	文件夹

图 1-15 SDL 目录结构

SDL 又分数个库：

- SDL2-2.0.5：SDL 主库 libsdl2.a。
- SDL2_image-2.0.1：SDL 图像辅助库 libsdl2_image.a。
- SDL2_mixer-2.0.1：SDL 音频辅助库 libsdl2_mixer.a。
- SDL2_ttf_2.0.14：SDL 字符串渲染辅助库 libsdl2_ttf.a。
- libvpx：不属于 SDL。https://github.com/webmproject/libvpx。

- libogg-1.3.0：解码 ogg 音频要用到的库，是 SDL2_mixer.a 辅助库。libogg.a。
- libvorbis-1.3.2：解码 ogg 音频要用到的库，是 SDL2_mixer.a 辅助库。libvorbis.a、libvorbisfile.a。

SDL 是个开源库，源代码、lib、dll 都可从 sdl 官方网站上下载。但这里还是选择了重新编译，这是考虑到几个原因。

1. 对官方 SDL 库进行过修改。有兴趣的可以对比这里下载的和官方版本，改了几处。
2. 要调试。直接下载的库没法设置断点，即没法进行源码级跟踪。
3. 进行源码级调试可以更好地理解 SDL 内部是如何工作。

libSDL2.a。打开并编译<SDL>/SDL2-2.0.4/Xcode-iOS/SDL/SDLiPhoneOS.xcodeproj。要确认编译是否成功，可以察看<SDL>/SDL2-2.0.4/Xcode-iOS/SDL/build 下是否生成了一个新的 libSDL2.a 文件。

libSDL2_image.a。打开<SDL>/SDL2_image-2.0.1/Xcode-iOS/SDL_image.xcodeproj，编译。要确认编译是否成功，可以察看<SDL>/SDL2_image-2.0.1/Xcode-iOS 下是否生成了一个新的 libSDL2_image.a 文件。

libSDL2_mixer.a。要编译 SDL2_mixer.a 需先编译 libogg.a、libvorbis.a、libvorbisfile.a。

1. libogg.a。打开并编译<SDL>/libogg-1.3.0/Xcode-iPhoneOS/Ogg.xcodeproj。把生成的 libogg.a

复制到<SDL>/SDL_mixer-1.2.12/Xcode-iOS/lib/-iphoneos（模拟器-iphonesimulator）。

2. libvorbis.a、libvorbisfile.a。打开<SDL>/libvorbis-1.3.2/Xcode-iPhoneOS/Vorbis.xcodeproj，编译。它一次编译只能编译两个中一个，要编译另外一个需在 Target 中进行切换。把生成的 libvorbis.a、libvorbisfile.a 复制到<SDL>/SDL_mixer-1.2.12/Xcode-iOS/lib/-iphoneos（模拟器-iphonesimulator）。
3. libSDL_mixer.a。打开并编译<SDL>/SDL2_mixer-2.0.1/Xcode-iOS/SDL_mixer.xcodeproj。要确认编译是否成功，可以察看<SDL>/SDL2_mixer-2.0.1/Xcode-iOS 下是否生成了一个新的 libSDL_mixer.a 文件。

libSDL2_ttf.a。打开并编译<SDL>/SDL2_ttf-2.0.12/Xcode-iOS/SDL2_ttf.xcodeproj。要确认编译是否成功，可以察看<SDL>/SDL2_ttf-2.0.12/Xcode-iOS 下是否生成了一个新的 libSDL2_ttf.a 文件。

libvpx.a。打开系统自带的“终端”app，运行下面命令。执行 build-ios.sh 后就会在 build-leagor 下生成 libvpx.a。

```
cd <libvpx>/build-leagor    #进入 build-leagor 目录
./build-ios.sh
```

打开并编译<SDL>/SDL2_ttf-2.0.12/Xcode-iOS/SDL2_ttf.xcodeproj。要确认编译是否成功，可以察看<SDL>/SDL2_ttf-2.0.12/Xcode-iOS 下是否生成了一个新的 libSDL2_ttf.a 文件。编译后形成的*.a 没有被放在统一目录下。接下要编译 app 时需把这些*.a 复制到 app 能找到位置，即<linker>/ios/lib/-iphoneos。至于 libogg.a、libvorbis.a、libvorbisfile.a 已静态链接在 libSDL2_mixer.a，不必把它们复到 linker 目录。

1.4.3 studio.app

工作目录中存在数个 app，编译它们的步骤基本一样，下面以 studio.app 为例。

名称	修改日期	大小	种类
app-studio	16/6/26 下午4:25	--	文件夹
data	16/6/26 下午4:25	--	文件夹
fonts	15/10/2 上午11:23	--	文件夹
projectfiles	16/6/26 下午10:57	--	文件夹
studio	16/6/26 下午4:28	--	文件夹
translations	16/6/26 下午4:25	--	文件夹
xwml	16/6/26 下午4:25	--	文件夹

图 1-17 studio 目录

- studio：studio 这个 app 的私有源码。
- projectfiles：存放 studio.xcodeproj 工程文件，及编译后形成的*.obj、studio.app 包。
- app-studio、data、fonts、translations、xwml：来自 app 包中的资源包，如何生成见“1.1.3 处理开发包”中的用编辑器生成发布时资源包。iOS 形成 studio.app 时要把这些内容打包。

编译

1. 把编译 SDL 生成的 5 个*.a 复制到<linker>/ios/lib。
2. 打开<studio>/projectfiles/Xcode-iOS/studio.xcodeproj。
3. 如果 studio.xcodeproj 是 Studio 自动生成，此时它的“src”组中内容是空的，你须要把 app 私有源文件加入到该组。具体步骤：1）在“src”组按右键弹出菜单，选择“Add Files to studio…”，

2）在新弹出的窗口单击“Option”按钮，确保已选择“Create groups”，执行“Add”。
4. 编译。

1.4.4 运行

一旦以上全部编译成功，<studio>/projectfiles/Xcode-iOS/目录会出现 studio.app。然后由它制做成 iPhone/iTouch/iPad 认识的 ipa 文件。

1.4.5 新建基于 Rose SDK 的应用

运行 Studio 导出 app 时会自动生成该 app 在 iOS 下的工程文件。要注意，自动生成的“src”组中内容是空的，须要把 app 私有源文件加入到该组。如何操作见“1.4.3 编译的步骤 3”。

1.5 Android Studio 3.0.1 + NDK-r16b（Android）

File——Settings——Build, Execution, Deployment——Instance Run，建议关闭“Enable Instant Run to hot swap code/resource changes on deploy (default enabled)”，否则 1）无法调试，像小米手机须要“设置”——“开发者选项”——关闭 MIUI 优化。2）生成的 apk 在调试机能正常运行，但无法安装在其它手机。

Rose 包含不少和处理器相关汇编代码，编译出的 app 只支持真机，不支持模拟器。

开发平台用 Windows，要安装的除了 Android Studio、NDK（如果是新手，请使用 Android Studio 内置的入口安装 NDK，“File”-“Project Structure…”-“SDK Location”），不必再安装其它工具，像类 Unit 命令行 cygwin，这里可直接使用系统自带的 cmd.exe。

到现在还没法自定义打包进 apk 资源目录的路径，这意味着要想打包进 apk，必须把资源放在<app>/src/main/assets/res（res 是 Rose 要求的路径）。而它们基本就是“<app>-res”中资源包的原样复制，为省容量，项目上传到 github 时往往就不复制了。为方便，Studio 有提供命令帮快速复制这部分资源，见“1.1.3 工作包和 app 包”中的“预复制资源到 Andorid 的 apk 包”。

C:\ddksample\sleep-src\sleep\projectfiles\android\app\src\main\assets\res

名称	修改日期	类型
app-sleep	2016/6/26 10:14	文件夹
data	2016/6/26 10:14	文件夹
fonts	2016/6/26 10:14	文件夹
translations	2016/6/26 10:14	文件夹
xwml	2016/6/26 10:14	文件夹

图 1-19 打包进 apk 的资源（包括 res 这个目录名）

下面以编译 studio 为例描述编译过程。

1.5.1 目录结构

除非已经了解为什么要这么设置目录，否则不要去修改要求的目录结构。

名称	修改日期	类型
apps	2016/6/24 21:18	文件夹
linker	2016/6/27 16:52	文件夹
scripts	2016/6/26 11:33	文件夹
SDL	2016/2/28 10:41	文件夹

图 1-19 顶层目录结构

- apps：源码目录，以及编译、链接时生成的文件。Studio 是个有点特殊的工程，它要和其它

app 并存，因而用了工作目录的目录名 apps，其它的 app 一般是 app id，像 blesmart。

- SDL：多媒体开发库 sdl，从 https://github.com/freeors/SDL 下载。
- linker：编译、链接时须要的头文件、库文件。它包括两部分，一是 SDL，二是除 SDL 外的第三方开发包，像微信、QQ。
- scripts：为方便编译过程而写的脚本文件。android_2_ndk.bat/android_2_app.bat 用于把编译生成的各个库复制到指定的位置。

总的来说，开发包须要编译的分两个部分：sdl 和 app。不论 sdl 还是 app 都要用 NDK，在编译它们之前首先要设对环境变量。用文本编辑器打开<scripts>/android_set_variable.bat，这文件由 Studio 自动生成，请根据你的电脑设对以下 NDK 等路径。

```
set NDK=c:\Users\ancientcc\AppData\Local\Android\sdk\ndk-bundle

set _APP_SRC=c:\apps-src
set SCRITPS=C:\apps-src\scripts

set SDL_sdl=c:\apps-src\SDL\SDL2-2.0.7\android
set SDL_image=c:\apps-src\SDL\SDL2_image-2.0.1\android
set SDL_mixer=c:\apps-src\SDL\SDL2_mixer-2.0.1\android
set SDL_ttf=c:\apps-src\SDL\SDL2_ttf-2.0.14\android
set libvpx=c:\apps-src\SDL\libvpx\projectfiles\android
set openv=c:\apps-src\SDL\opencv\projectfiles\android
set protobuf=c:\apps-src\SDL\protobuf\projectfiles\android

set studio=c:\apps-src\apps\projectfiles\android
```

运行 DOS 命令行后，须先运行这个 bat 以让环境中存在以上定义的变量。

```
cd c:\apps\apps-src\scripts
android_set_variable.bat
```

注：查看以上变量是否被设置，可在命令行运行“set”。

```
Set
```

1.5.2 SDL

下载 apps 包后，已把须要的 SDL 相关库文件放在<linker>/android，如果你不想重编译它们，可以略过这部分。

名称	修改日期	类型
libogg-1.3.0	2015/8/5 15:23	文件夹
libvorbis-1.3.2	2015/3/1 21:06	文件夹
SDL2_image-2.0.1	2016/3/27 22:19	文件夹
SDL2_mixer-2.0.1	2016/3/27 22:21	文件夹
SDL2_net-2.0.1	2016/3/27 22:21	文件夹
SDL2_ttf-2.0.14	2016/3/27 22:21	文件夹
SDL2-2.0.4	2016/6/26 16:32	文件夹

图 1-20 SDL 目录结构

SDL 又分数个库：

1. SDL2-2.0.5：SDL 主库 libsdl2.so。
2. SDL2_image-2.0.1：SDL 图像辅助库 libsdl2_image.so。
3. SDL2_mixer-2.0.1：SDL 音频辅助库 libsdl2_mixer.so。
4. SDL2_ttf_2.0.14：SDL 字符串渲染辅助库 libsdl2_ttf.so。
5. libvpx：不属于 SDL。https://github.com/webmproject/libvpx。

SDL 是个开源库，源代码、so 都可从 sdl 官方网站上下载。但这里还是选择了重新编译，这

是考虑到几个原因。

1. 对官方 SDL 库进行过修改。有兴趣的可以对比这里下载的和官方版本，改了几处。

编译 libSDL2.so

```
cd %SDL_sdl%
%NDK%/ndk-build
```

要确认编译是否成功，可以察看<SDL>/SDL2-2.0.4/android/libs/armeabi 下是否生成了一个新的 libSDL2.so 文件。

到现在要编译的库都是不依赖其它库的库，接下要编译依赖其它库的库，像 SDL2_ttf 要依赖以上编译出的 libSDL2.so，为接下编译时不报错，须复制编译出的*.so 到 ndk 库目录，从而让编译接下库时的链接阶段可成功找到这些库。

```
cd %SCRIPTS%
android_2_ndk.bat
```

注：如果报出不能复制 libSDL2_image.so，libSDL2_mixer.so，libSDL2_ttf.so，可忽略这几个错误。

编译 libSDL2_image.so

```
cd %SDL_image%
%NDK%/ndk-build
```

要确认编译是否成功，可以察看<SDL>/SDL2_image-2.0.1/android/libs/armeabi-v7a 下是否生成了一个新的 libSDL2_image.so 文件。

编译 libSDL2_mixer.so

```
cd %SDL_mixer%
%NDK%/ndk-build
```

要确认编译是否成功，可以察看<SDL>/SDL2_mixer-2.0.1/android/libs/armeabi-v7a 下是否生成了一个新的 libSDL2_mixer.so 文件。

编译 libSDL2_ttf.so

```
cd %SDL_ttf%
%NDK%/ndk-build
```

要确认编译是否成功，可以察看<SDL>/SDL_ttf-2.0.14/android/libs/armeabi-v7a 下是否生成了一个新的 libSDL2_ttf.so 文件。

编译 libvpx.so

```
cd %libvpx%
%NDK%/ndk-build
```

要确认编译是否成功，可以察看<SDL>/libvpx/projectfiles/android/libs/armeabi-v7a 下是否生成了一个新的 libvpx.so 文件。

编译 opencv.so

```
cd %opencv%
%NDK%/ndk-build
```

要确认编译是否成功，可以察看<SDL>/opencv/projectfiles/android/libs/armeabi-v7a 下是否生成了一个新的 opencv.so 文件。

编译 protobuf.so

```
cd %libvpx%
%NDK%/ndk-build
```

要确认编译是否成功，可以察看<SDL>/protobuf/projectfiles/android/libs/armeabi-v7a 下是否生成了一个新的 protobuf.so 文件。

1.5.3 libmain.so

名称	修改日期	类型
external	2016/6/26 22:09	文件夹
librose	2016/6/26 22:09	文件夹
projectfiles	2016/6/26 22:09	文件夹
studio	2016/6/26 22:09	文件夹

图 1-22 apps 目录结构

- external：rose 库源码分为两部分，它是当中一个。存放要没意外不要去改的源码，一般是比较稳定了的开源包，像 boost、bzip2、zlib、gettext。它被内嵌的方式链入 app。
- librose：rose 库源码分为两部分，它是当中一个。它被内嵌的方式链入 app。
- projectfiles：子目录 android 对应 Android 时工程文件，及编译后形成的*.obj、studio.apk。
- studio：studio 这个 app 的私有源码。

为接下编译 libmain.so 不报错，复制编译出的*.so 到 ndk 库目录

```
cd %SCRIPTS%
android_2_ndk.bat
```

注：此时应该不会报错。

编译 libmain.so

```
cd %studio%
%NDK%/ndk-build
```

要确认编译是否成功，察看<projectfiles>/android/app/libs/armeabi-v7a 下是否生成了一个新的 libmain.so 文件。

1.5.4 studio.apk 及运行

一旦以上全部编译成功，DOS 命令行中执行以下命令。

```
cd %SCRIPTS%
android_2_app.bat %studio%
```

以上命令把相关 so 复制到<projectfiles>/android/app/libs/armeabi 目录。下一步生成 studio.apk。

1. 运行 Android Studio。
2. “Welcome to Android Studio”选择“Open an existing Android Studio project”。
3. “Open File or Project”中定位到<projectfiles>/android，打开。
4. 连上 Android 设备，执行菜单命令“Run”——“Run ‘app’”。就会在<projectfiles>/android/app/build/outputs/apk 生成 studio.apk，并可调试该 apk。

1.5.5 编译、链接选项

Application.mk

Android Studio 的 JNI（NDK）开发主要有两种情况：一种是使用已经编译好的.so 动态库（用自写的 Android.mk）；一种是使用 c/c++源代码开发（由 gradle 自动生成 Android.mk），Rose 现阶段使用第一种方法。主要原因是第二种生成的“LOCAL_C_INCLUDES”会使编译变复杂并可能失败。

- APP_ABI := armeabi-v7a

  指示要生成的 CPU 架构。当前 Android 支持的 CPU 架构包括 armeabi、armeabi-v7a、arm64-v8a、x86、x86_64、mips、mips64。针对手机、平板市场，绝大多数基于 arm，即前三种。armeabi 针对早期 arm v5 cpu,市场上基本已绝迹。arm64-v8a 针对 arm v8 cpu，但 NDK 到 APP_PLATFORM=android-21 才开始支持 64 位。考虑到每一种架构会生成一组 so，为减少安装包容量只设置了 armeabi-v7a，等将来哪天 APP_FLATFORM 设到 android-21 时再考虑是否增加 arm64-v8a。

生成的 apk 缺少架构可能严重影响效率，主要原因是缺少相应架构后只能采用低效渲染方式。举个例子，手机用的 CPU 是 ARMv7-A Cortex，如果只编译 armeabi，那它只会用软件渲染，渲染一窗口可能要花 1.8 秒，一旦增加 armeabi-v7a，那运行时会选择到硬件泻染，时间会降低到 0.8 秒。

如果主 so 编译出了 armeabi-v8a 架构，那依赖的第三方库需要有 armeabi-v8a 相关的 so 文件，不然就会报错。

- APP_PLATFORM := android-21

  android-21 换成通俗版本号就是 Android-5.0。指示要基于运行的 Android 版本。

  针对每一 APP_FLATFORM，NDK 提供了和之对应的 API，它们是头文件形式的函数声明，类似 Windows 上的 dll 头文件部分。具体运行则要到真机，真机版本决定了它的最后执行效率。比如 strchr，如果真机是 android-21，那它最终运行的是 android-21 实现的 strchr，换到 android-23 的真机，那就变成 23 版本。

  一般来说，APP_PLATFORM 值越大，会向外提供更多 API。举个例子，一直到 android-21 都未提供 stpcpy，但 android-23 有了。编译 app 用 APP_PLATFORM := android-23，意味着由 android 系统提供 stpcpy 实现，可如果这 app 换在 android-21 运行，会由于该系统没有 stpcpy 导致 loadLibrary 加载该 so 时失败。从这个说，APP_PLATFORM 值越小兼容性更好，但这个值越低意味着不能调用新版才有的函数。

  综上所述，设置这值要考虑兼容性（要小）和功能（要大），当前选择 android-21。考虑到三个原因，一是 webrtc 要求至少 android-16，二是 android-18 开始支持蓝牙 4.x，三是 TensorFlow Lite 要求至少 android-21。
- APP_STL := c++_shared

  LLVM libc++共享运行库。rose 需要 c++11、异常处理，RTTI，此库完整支持了这些特色。
- APP_CPPFLAGS += -fexceptions

  默认情况下不使能 exceptions，需手动打开。
- APP_CPPFLAGS += -frtti

  默认情况下不使能 rtti，需手动打开。
- APP_CFLAGS += -Wno-error=format-security

  把类似以下的警告不视做错误。

```
Jni../../../src/ai/default/ai.cpp:1496:5: warning: format not a string literal and no format arguments [-Wformat-security]
```

1.5.6 权限策略

基本原则：app 必须拥有需要权限，否则不让继续执行。

对 Android 6.0 或以上版本，支持运行时申请权限。在这些系统，app 一开始会检查权限，是“询问”状态的让用户选择。一旦用户有选择了“拒绝”，弹出“缺少限权”警告框，给出两个出口，或跳到“设置”让手动选择为“允许”，或退出 app。

app 如何收集需要的限权？mPessionList 是 SDLActivity 的成员变量，存储着需要的权限，它来自两部分。一是 librose 需要的权限，它们在 onCreate 时加入 mPessionList。二是 app 私有，建议在 app 的构造函数中加入 mPermissionList。举个例子，app 需要发送短信，那在 <java>/com/leagor/smsrobot/app.java 加入以下代码。

```
public class app extends SDLActivity {
    public app() {
        mPermissionList.add(Manifest.permission.SEND_SMS);
    }
}
```

系统先执行构造 app，然后 onCreate，是第二部分先于第一部分加入 mPessionList。

1.6 NDK-r16b（kOS）

开发平台用 Windows，要安装的除了 NDK（和 Android 是同一个），不必再安装其它工具，像类 Unit 命令行 cygwin，这里可直接使用系统自带的 cmd.exe。

为减少尺寸，kOS app 共用了 Android app 时生成的资源包。

下面以编译 studio 为例描述编译过程。

1.5.1 目录结构

除非已经了解为什么要这么设置目录，否则不要去修改要求的目录结构。

名称	修改日期	类型
apps	2016/6/24 21:18	文件夹
linker	2016/6/27 16:52	文件夹
scripts	2016/6/26 11:33	文件夹
SDL	2016/2/28 10:41	文件夹

图 1-19 顶层目录结构

- apps：源码目录，以及编译、链接时生成的文件。Studio 是个有点特殊的工程，它要和其它 app 并存，因而用了工作目录的目录名 apps，其它的 app 一般是 app id，像 blesmart。
- SDL：多媒体开发库 sdl，从 https://github.com/freeors/SDL 下载。
- linker：编译、链接时须要的头文件、库文件。它包括两部分，一是 SDL，二是除 SDL 外的第三方开发包，像 opencv、protobuf、vpx。
- scripts：为方便编译过程而写的脚本文件。android_2_ndk.bat/android_2_app.bat 用于把编译生成的各个库复制到指定的位置。

总的来说，开发包须要编译的分两个部分：sdl 和 app。不论 sdl 还是 app 都要用 NDK，在编译它们之前首先要设对环境变量。用文本编辑器打开<scripts>/kos_set_variable.bat（不是 Android 的 android_set_variable.bat），这文件由 Studio 自动生成，请根据你的电脑设对以下 NDK 等路径。

```
set NDK=c:\Users\ancientcc\AppData\Local\Android\sdk\ndk-bundle

set _APP_SRC=c:\apps-src
set SCRITPS=C:\apps-src\scripts

set SDL_sdl=c:\apps-src\SDL\SDL2-2.0.8\android
set SDL_sdl_so_path=c:\ddksample\apps-src\linker\kos\lib\armeabi-v7a
set SDL_image=c:\apps-src\SDL\SDL2_image-2.0.3\android
set SDL_mixer=c:\apps-src\SDL\SDL2_mixer-2.0.1\android
set SDL_ttf=c:\apps-src\SDL\SDL2_ttf-2.0.14\android
set libvpx=c:\apps-src\SDL\libvpx\projectfiles\android
set openv=c:\apps-src\SDL\opencv\projectfiles\android
set protobuf=c:\apps-src\SDL\protobuf\projectfiles\android

set studio=c:\apps-src\apps\projectfiles\android
```

运行 DOS 命令行后，须先运行这个 bat 以让环境中存在以上定义的变量。

```
cd c:\apps\apps-src\scripts
kos_set_variable.bat
```

注：查看以上变量是否被设置，可在命令行运行“set”。

```
Set
```

除 libmain.so 外，只有 libSDL.so 在 Android 和 kOS 存在差别。Android 时，编译出的 libSDL.so 放在<apps-src>/linker/android/lib，kOS 时放在<apps-src>/linker/kos/lib。bat 中的环境变量 SDL_sdl_so_path 指示着这个路径。

1.6.2 SDL

所有*.so 编译方法和 Android 一样。只不过查看编译 libSDL2.so 结果的方法换在

<SDL>/SDL2-2.0.8/kos/libs/armeabi，而不是<SDL>/SDL2-2.0.8/android/libs/armeabi。

1.6.3 libmain.so

名称	修改日期	类型
external	2016/6/26 22:09	文件夹
librose	2016/6/26 22:09	文件夹
projectfiles	2016/6/26 22:09	文件夹
studio	2016/6/26 22:09	文件夹

图 1-22 apps 目录结构

- external：rose 库源码分为两部分，它是当中一个。存放要没意外不要去改的源码，一般是比较稳定了的开源包，像 boost、bzip2、zlib、gettext。它被内嵌的方式链入 app。
- librose：rose 库源码分为两部分，它是当中一个。它被内嵌的方式链入 app。
- projectfiles：子目录 android 对应 Android 时工程文件，及编译后形成的*.obj、studio.apk。
- studio：studio 这个 app 的私有源码。

为接下编译 libmain.so 不报错，复制编译出的*.so 到 ndk 库目录

```
cd %SCRIPTS%
android_2_ndk.bat
```

注：此时应该不会报错。

编译 libmain.so

```
cd %studio%
%NDK%/ndk-build
```

要确认编译是否成功，察看<projectfiles>/kos/app/libs/armeabi-v7a 下是否生成了一个新的 libmain.so 文件。

1.6.4 studio.ksa 及运行

一旦以上全部编译成功，DOS 命令行中执行以下命令。

```
cd %SCRIPTS%
android_2_app.bat %studio%
```

以上命令把相关 so 复制到<projectfiles>/kos/app/libs/armeabi 目录。下一步生成 studio.ksa。1）运行 Rose Studio。2）在左侧树形控件的“apps-res”单击右键，弹出一级菜单移到要导出的 app：studio，二级菜单选“导出 kOS 安装包…”。3）命令结束后，就会在<projectfiles>/kos 下生成 com.leagor.studio.ksa。

“导出 kOS 安装包…”需要数据有两个，一是<projectfiles>/android/studio/src/main/assets/res 下的资源，二是<projectfiles>/kos/studio/libs/armeabi-v7a 下的各个*.so。

1.6.5 编译、链接选项

Application.mk

Android Studio 的 JNI（NDK）开发主要有两种情况：一种是使用已经编译好的.so 动态库（用自写的 Android.mk）；一种是使用 c/c++源代码开发（由 gradle 自动生成 Android.mk），Rose 现阶段使用第一种方法。主要原因是第二种生成的“LOCAL_C_INCLUDES”会使编译变复杂并可能失败。

- APP_ABI := armeabi-v7a

  指示要生成的 CPU 架构。当前 Android 支持的 CPU 架构包括 armeabi、armeabi-v7a、arm64-v8a、x86、x86_64、mips、mips64。针对手机、平板市场，绝大多数基于 arm，即前三种。armeabi 针对早期 arm v5 cpu,市场上基本已绝迹。arm64-v8a 针对 arm v8 cpu，但 NDK 到 APP_PLATFORM=android-21 才开始支持 64 位。考虑到每一种架构会生成一组 so，为减少

安装包容量只设置了 armeabi-v7a，等将来哪天 APP_FLATFORM 设到 android-21 时再考虑是否增加 arm64-v8a。

生成的 apk 缺少架构可能严重影响效率，主要原因是缺少相应架构后只能采用低效渲染方式。举个例子，手机用的 CPU 是 ARMv7-A Cortex，如果只编译 armeabi，那它只会用软件渲染，渲染一窗口可能要花 1.8 秒，一旦增加 armeabi-v7a，那运行时会选择到硬件泻染，时间会降低到 0.8 秒。

如果主 so 编译出了 armeabi-v8a 架构，那依赖的第三方库需要有 armeabi-v8a 相关的 so 文件，不然就会报错。

- APP_PLATFORM := android-21

  android-21 换成通俗版本号就是 Android-5.0。指示要基于运行的 Android 版本。

  针对每一 APP_FLATFORM，NDK 提供了和之对应的 API，它们是头文件形式的函数声明，类似 Windows 上的 dll 头文件部分。具体运行则要到真机，真机版本决定了它的最后执行效率。比如 strchr，如果真机是 android-21，那它最终运行的是 android-21 实现的 strchr，换到 android-23 的真机，那就变成 23 版本。

  一般来说，APP_PLATFORM 值越大，会向外提供更多 API。举个例子，一直到 android-21 都未提供 stpcpy，但 android-23 有了。编译 app 用 APP_PLATFORM := android-23，意味着由 android 系统提供 stpcpy 实现，可如果这 app 换在 android-21 运行，会由于该系统没有 stpcpy 导致 loadLibrary 加载该 so 时失败。从这个说，APP_PLATFORM 值越小兼容性更好，但这个值越低意味着不能调用新版才有的函数。

  综上所述，设置这值要考虑兼容性（要小）和功能（要大），当前选择 android-21。考虑到三个原因，一是 webrtc 要求至少 android-16，二是 android-18 开始支持蓝牙 4.x，三是 TensorFlow Lite 要求至少 android-21。

- APP_STL := c++_shared

  LLVM libc++共享运行库。rose 需要 c++11、异常处理，RTTI，此库完整支持了这些特色。

- APP_CPPFLAGS += -fexceptions

  默认情况下不使能 exceptions，需手动打开。

- APP_CPPFLAGS += -frtti

  默认情况下不使能 rtti，需手动打开。

- APP_CFLAGS += -Wno-error=format-security

  把类似以下的警告不视做错误。

```
Jni../../../src/ai/default/ai.cpp:1496:5: warning: format not a string literal and no format arguments [-Wformat-security]
```

1.7 操作系统宏

对这些宏除了要知道宏名还需要知道出处。它们用于判断是哪个操作系统，以及是 32 还是 64 位。判断 32 还是 64 位要分两个步骤，1）这是哪个操作系统，2）是 32 还是 64 位。

Microsoft Windows

- _WIN32：Microsoft 32 位/64 位 C/C++编译器内置定义。

  编译器内置了该宏定义，使用时不必包含任何必须的头文件。

经常能看到 WIN32、_WINDOWS，这些是 Visual Studio 在创建工程后给的默认定义，不能作为 Windows 的标准定义。

如何判断是 32 位还是 64 位？看是否定义了_WIN64。编译器内置定义了该宏，使用时不必包含任何必须的头文件。

Apple Mac OS X

- __APPLE__：Apple C/C++（LLVM）编译器内置定义。

如何判断是 32 位还是 64 位？看是否定义了__LP64__。编译器内置定义了该宏，使用时不必包含任何必须的头文件。

Apple iOS

- __APPLE__：Apple C/C++（LLVM）编译器内置定义。
- TARGET_OS_IPHONE：TargetConditionals.h 内定义。iOS（包括 iPad/iPhone/iTouch）真机及模拟器。
- TARGET_IPHONE_SIMULATOR：TargetConditionals.h 内定义。只针对 iOS 模拟器。

注：TARGET_OS_IPHONE、TARGET_IPHONE_SIMULATOR 是宏，但在 Mac OS X 这两个宏也是存在的，只不过值是 0！

Apple C/C++只内置__APPLE__宏定义，但 Mac OS X 和 iOS 都定义了这宏，那要如何区分是 OS X 还是 iOS？——**使用 TARGET_OS_IPHONE 宏**。程序要满足“#if defined(__APPLE__) && TARGET_OS_IPHONE”时认为是 iOS 系统。

但 TARGET_OS_IPHONE 是在 TargetConditionals.h 内定义的，要使用该宏则必须先#include <TargetConditionals.h>，一般代码得有以下顺序。

```
#if defined(__APPLE__)
    #include <TargetConditionals.h>
    #if TARGET_OS_IPHONE
        放置 iOS 代码
    #else
        放置 OS X 代码
    #endif
#endif
```

TargetConditionals.h 路径，以 Xcode4.2 下的 iPhoneOS SDK 为例。

```
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS4.2.sdk/
usr/include/TargetConditionals.h
```

如何判断是 32 位还是 64 位？看是否定义了__LP64__。编译器内置定义了该宏，使用时不必包含任何必须的头文件。

Google Android

- ANDROID：Android GCC/LLVM 编译器内置定义。
  编译器内置了此个宏定义，使用时不必包含任何必须的头文件。

kOS

- ANDROID：Android GCC/LLVM 编译器内置定义。
  编译器内置了此个宏定义，使用时不必包含任何必须的头文件。
- _KOS：app 或*.so 的 Android.mk 内定义。

1.8 检测内存泄漏（Memory Leak Detection）

能按需分配/释放内存是 C/C++强大功能的体现，但正所谓功能越强隐患越大，要是不及时或忘记释放分配的内存，它会造成内存泄漏，泄漏累积到一定程度会影响程序性能，甚至让程序因缺少内存而崩溃。但泄漏不属语法问题，编译、链接过程检测不出它的，要等到真正运行了才能知道有没有存在泄漏。

内存泄漏在代码中表现是调用了分配函数后忘记调用相应的释放函数。依函数来源可把 C/C++内存分配归为两类：C 运行库和平台专用 API。运行库常见的有“malloc”、“new”，API 则是操作系统相关，像 Windows 上有 VirtualAlloc。以运行库观点描述，泄漏问题就是程序调用“malloc/new”后没有调用相应的“free/delete”。

这里介绍种利用 C 运行库加 Visual Studio 调试器检测内存泄漏方法，该方法基于 C 运行库，

虽然 C 运行库是跨平台标准库，但此中方法实现的 C 宏、头文件并不是跨平台的。虽然方法不是跨平台，但对程序来说，各平台下是同一份代码，因而一旦在 Visual Studio 平台验证了该代码不存在内存泄漏，那意味着这些代码在其它平台也已通过验证。

注：该方法只能检测运行库函数倒致的泄漏，无法查出平台专用 API，像 VirtualAlloc。

为直观察看这种检测方法的大概情况，让做以下操作。

1. 打开 kingdom 工程，“Build”——“Configuration Manager...”，“Active solution configuration:”中选择“Debug”，设当前配置为“调试”。
2. “Solution Explorer”窗口的“editor”上按鼠标右键，弹出菜单中选“Set as Startup Project”，把“编辑器”设为启动项目。
3. “Debug”——“Start Debugging”，让以调试模式运行编辑器。
4. 退出编辑器，打开“Output”察看输出信息。（真正稳定程序是不应该看到泄漏信息的。）

```
......
   e:\ddksample\kingdom-src\kingdom\src\editor2\main.cpp(1165)  :  {204}  normal
block at 0x007D6510, 41 bytes long.
   Data: <          > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
   {197} normal block at 0x007D6218, 52 bytes long.
   Data: < P} p\} P}    > 00 50 7D 00 70 5C 7D 00 00 50 7D 00 00 00 00 00
   {193} normal block at 0x007D5D50, 60 bytes long.
   Data: < A ]}       > CC C3 41 01 C8 5D 7D 00 00 00 00 00 00 00 00 00
   {191} normal block at 0x007D5C70, 52 bytes long.
......
```

根据第一条泄漏信息，让定位到信息指示的“main.cpp”文件第 1165 行，能看到一条“malloc”语句，而该语句没有相应的“free”函数，致使泄漏 41 字节。

```
char* ptr = (char*)malloc(41);
```

1.8.1 C 运行库

深入内存泄漏让从叙述 C 行运库开始。早期 Visual Studio 中运行库分还单线程、多线程版本，时至今日单线程程序已没什么意义，于是只剩多线程。运行库根据调试/发布、静态/动态链接方式分为四种版本。

C 运行库	编译选项	预定义宏	
libcmt.lib	Multi-threaded(/MT)	_MT	多线程、静态链接
msvcrt.lib	Multi-threaded DLL(/MD)	_MT，_DLL	多线程、动态链接
libcmtd.lib	Multi-threaded Debug(/MTd)	_DEBUG，_MT	多线程、静态链接（调试）
msvcrtd.lib	Multi-threaded Debug DLL(/MDd)	_DEBUG，_MT，_DLL	多线程、动态链接（调试）

MSDN 等资料都有对运行库说明，这里只说下几个结论。

- 同个二进制目标文件中项目必须设置同种运行库选项。一个二进制目标文件指 exe 或 dll，它们一个共同特点是运行时会被操作系统以单独模块方式被加载，相应地，二进制目标文件中项目包括生成 exe/dll 项目、导入该 exe/dll 的静态库项目。举个例子，kingdom.exe 要使用 kingdom-core 项目生成的 kingdom-core.lib，lualib 生成的 lualib.lib，这时 kingdom.exe、kingdom-core.lib 和 lualib.lib 必须使用同种运行库选项。需再次强调下，这个相同只是该 exe/dll 本身和链接入该 exe/dll 的静态库，不包括该 exe/dll 要使用的动态链接库，像 intl.dll、SDL.dll、SDL_image.dll。一旦出现多个项目不是同一种运行库设置，意味生成该二进制目标文件时同一个运行库函数会有不同版本实现，链接时会报类似以下的错误。

```
   LIBCMTD.lib(invarg.obj) : error LNK2005: __invalid_parameter already defined
in MSVCRTD.lib(MSVCR90D.dll)
   LIBCMTD.lib(invarg.obj) : error LNK2005: __invoke_watson already defined in
MSVCRTD.lib(MSVCR90D.dll)
   LIBCMTD.lib(mlock.obj)  :  error  LNK2005:  __lock  already  defined  in
MSVCRTD.lib(MSVCR90D.dll)
```

```
    LIBCMTD.lib(mlock.obj)  :  error  LNK2005:  __unlock  already  defined  in
MSVCRTD.lib(MSVCR90D.dll)
    LIBCMTD.lib(sprintf.obj)  :  error  LNK2005:  _sprintf  already  defined  in
MSVCRTD.lib(MSVCR90D.dll)
    LIBCMTD.lib(sprintf.obj)  :  error  LNK2005:  _sprintf_s  already  defined  in
MSVCRTD.lib(MSVCR90D.dll)
```

- 采用静态链接时（libcmt.lib、libcmtd.lib），编译器链接过程已把“运行库代码”填充入二进制目标文件，因而运行时在链接的动态库中看不到这些库。因为运行时已不需要这些库，目标机上也就不必存在编译时链接向的那版本 libcmt.dll/libcmt.dll
- 采用动态链接时（msvcrt.lib、msvcrtd.lib），编译器链接过程没有把“运行库代码”填充入二进制目标文件，而是要等到运行时动态填充，因而运行时在链接的动态库中会看到这些库。因为要运行时链接，目标机上必须存在编译时链接向的那版本 msvcrt.dll/msvcrtd.dll。
- 只有调试版本才支持检测内存泄漏，那如何决定是选择静态还是动态链接。相比于动态，静态优势在于它不须要目标机存在编译时链接向的动态库；相比于静态，动态优势在于它最后生成的二进制目标文件尺寸更小。具体到调试这种应用，调试机上肯定已安装 Visual Studio，也就是说目标机上肯定存在编译时链接向的动态库，因此为减少目标文件尺寸优先选择动态链接。因为这个理由，生成调试版本时 kingdom.exe、gettext、SDL 中各工程采用的都是动态链接，即“C/C++”、“Code Generation”中运行库选项都是“Multi-threaded Debug DLL(/MDd)”。

1.8.2 为检测内存泄漏，代码上做的支持

支持检测内存泄漏代码分两部分。

代码一：存在分配函数的源文件要包含以下代码

```
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
```

三条语句要注意次序，_CRTDBG_MAP_ALLOC 须首定被定义，crtdbg.h 须放在最后。在书写支持泄漏代码会经常用到该三条语句，把它们叫做“支持泄漏标准头文件代码”。

代码二：在程序执行入口处执行_CrtSetDbgFlag

```
_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
```

注释

1. 定义_CRTDBG_MAP_ALLOC 宏是为输出详细格式的泄漏时信息，以更快排除错误；
2. #include <crtdbg.h>，crtdbg.h 声明了_CrtSetDbgFlag，以及定义该函数的标志参数。此外也声明定义_CRTDBG_MAP_ALLOC 宏后的分配/释放函数。
3. 调用_CrtSetDbgFlag 是为了让程序退出前能命令 Visual Studio 调试器自动调用_CrtDumpMemoryLeaks 以显示内存检测结果，_CrtDumpMemoryLeaks 作用是显示整个运行过程中内存泄漏状况。正所谓入口处只一个，出口处可能多个，调用_CrtSetDbgFlag 往往也就被放在 main/WinMian。注：参数要用 _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF。
4. 只要把运行库设为调试版本，运行库就已自动检测内存泄漏。定义_CRTDBG_MAP_ALLOC 是要能让输出详细格式的泄漏信息，调用_CrtSetDbgFlag 则是让运行结束后可以让在“Output”显示泄漏信息。
5. 泄漏信息有两种格式：简化格式、详细格式。

简化格式

new 导致的泄漏只会输出简化格式。没有定义“_CRTDBG_MAP_ALLOC”时 malloc 也只会输出简化格式。

```
{71960} normal block at 0x020328F0, 4080 bytes long.
    Data: <        > BD 01 00 00 C1 F5 B1 B8 D6 AE B2 CE C4 B1 A1 A3
{70482} normal block at 0x020318C0, 4080 bytes long.
```

```
 Data: <        > 05 00 00 00 BA C2 C3 C8 00 CD CD CD 03 00 00 00
```

详细格式

相比简化格式，详细格式多了分配函数所在文件/行。定义了“_CRTDBG_MAP_ALLOC”后，malloc 会输出详细格式。

```
    e:\ddksample\kingdom\gettext\gettext\gettext-
runtime\intl\dcigettext.c(1309) : {71960} normal block at 0x018328F0, 4080 bytes
long.
    Data: <        > BD 01 00 00 C1 F5 B1 B8 D6 AE B2 CE C4 B1 A1 A3
    e:\ddksample\kingdom\gettext\gettext\gettext-
runtime\intl\dcigettext.c(1309) : {70482} normal block at 0x018318C0, 4080 bytes
long.
 Data: <        > 05 00 00 00 BA C2 C3 C8 00 CD CD CD 03 00 00 00
```

1.7.3 检测内存泄漏原理

调试器实现检测内存泄漏基本原理是把通常下的分配/释放函数原型导向专门的、带调试功能的函数实现，在那些实现中执行对分配/释放完整性检查。像 malloc，当使能检测泄漏时，它真正执行的是“_malloc_dbg”。

```
摘自<crtdbg.h>。

#ifdef _CRTDBG_MAP_ALLOC
#define malloc(s)    _malloc_dbg(s, _NORMAL_BLOCK, __FILE__, __LINE__)
#define free(s)     _free_dbg(p, NORMAL_BLOCK)
……
#endif
......
_CRTIMP void * __cdecl _malloc_dbg(size_t, int, const char *, int);
_CRTIMP void * __cdecl _free_dbg(void *, int);
......
```

1.8.4 同时检测 exe、dll 中泄漏

代码量越来越大时，基于模块化管理等原因会把代码分割为一个 exe 和多个 dll，这里让说下一次运行时要能在这些 exe、dll 同时检测出泄漏，程序上要做何样支持。

- 相关 exe、dll 运行库设置必须是 DEBUG 版。
- 代码中只须在一处调用“_CrtSetDbgFlag”。这个调用可以放在 exe，也可以放在 dll，只要这点在运行时能够被执行到。
- 为输出详细格式泄漏信息，所有调用“malloc/new”的 c/cpp 源文件必须包含“支持泄漏标准头文件代码”。这个包含对于各个 exe、dll 是独立的，即工程中有几个 exe、dll 那就须要多少份“支持泄漏标准头文件代码”。

第二章 SDL

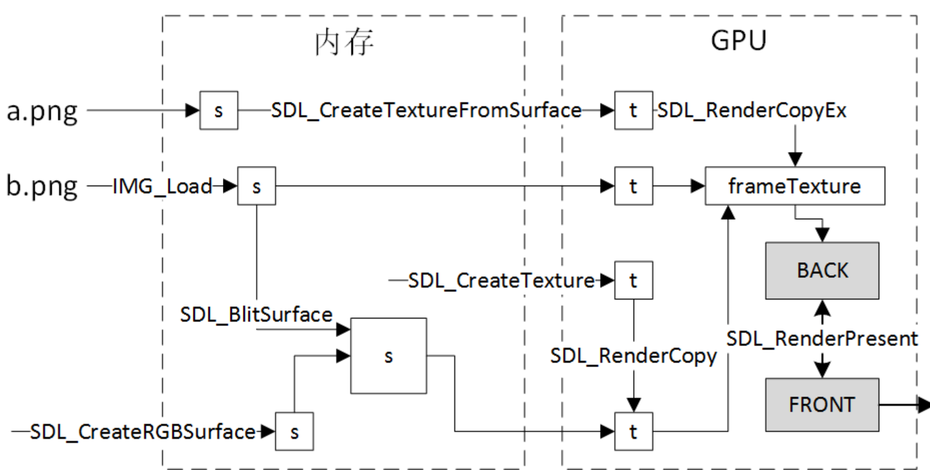
本章目标

- 理解 SDL_Surface 中 pixels 数据块格式。
- SDL_PixelFormat 结构的分配和释放过程。
- 知道点对点块移、RLE 块移优点和缺陷。
- 哈希表、surface 表、LRU 链表协同管理图面。

SDL（Simple DirectMedia）是一个开源的跨平台多媒体开发包。它有广义和狭义之分，狭义 SDL 就指 SDL 库，广义还包括 SDL_mixer、SDL_image、SDL_ttf 和 SDL_net。Rose 依赖广义 SDL，但网络模块基于 Webrtc，因而不需要 SDL_net。

2.1 主流程

2.1.1 概述



让以加载 a.png 看大致流程。IMG_Load 是 SDL 提供的加载图像 API，它根据扩展名判断出图像类型，然后分析文件中数据，在内存形成 SDL_Surface。SDL_CreateTextureFromSurface 用于从 SDL_Surface 生成 SDL_Texture。这个 SDL_Texture 被用 SDL_RenderCopy 或 SDL_RenderCopyEx 渲染到叫 frameTexture 的 SDL_Texture。相关纹理都渲染好后，frameTexture 把自个内容渲染到显示器的后台缓冲区，最后调用 SDL_RenderPresent 实现双缓冲机制（Double Buffering）的前后台切换，app 希望的内容就被显示到屏幕。

内存中的 SDL_Surface 并不一定要来自硬盘文件，也可调用 SDL_CreateRGBSurface 生成一个空的。多个 SDL_Surface 可用 SDL_BlitSurface 混合成一个更复杂 SDL_Surface，混合过程叫块移（Blit）。

GPU 中的 SDL_Texture 并不一定要来自 SDL_Surface，也可调用 SDL_CreateTexture 生成一个空的。多个 SDL_Texture 可用 SDL_RenderCopy 或 SDL_RenderCopyEx 混合成一个更复杂 SDL_Texture。

frameTexture 用 SDL_CreateTexture 创建，本质上和其它 SDL_Texture 一模一样，只是位置上，让它成为所有内容的汇集点，方便一次性渲染到显示器的后台缓冲区。对 app 来说，它要做的是把来自文件的、自画的、块移出的、混合出的，等等，在 frameTexture“画”出自个想要的结果。

2.1.2 框架代码

```
SDL_Window* window = SDL_CreateWindow(title, x, y, w, h, flags);
SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, 0);
```

```
SDL_Texture* frameTexture = SDL_CreateTexture(rendere, SDL_PIXELFORMAT_ARGB8888,
          SDL_TEXTUREACCESS_TARGET, w, h);
.....
.....
SDL_SetRenderTarget(renderer, NULL);
SDL_RenderCopy(renderer, frameTexture, NULL, NULL);
SDL_RenderPresent(renderer);
SDL_SetRenderTarget(renderer, frameTexture);
```

首先要创建一个 SDL_Windows（主窗口），然后用它创建 SDL_Renderer（渲染器）。x、y、w、h 分别是窗口的左上角、宽度、高度。flags 是个用“|”合成的标志，对要支持 HDPI 的 app，必须包含 SDL_WINDOW_ALLOW_HIGHDPI。有了渲染器就可创建 frameTexture 了，在 Rose，不论 surface 还是 texture，中间格式都是 SDL_PIXELFORMAT_ARGB8888。

省略号部分假设是执行向 frameTexture 执行复杂混合。SDL_SetRenderTarget 把接下要渲染到的目的地换为显示器的后台缓冲区。SDL_RenderCopy 执行把 frameTexture 渲染到到后缓冲区， SDL_RenderPresent 执行切换前、后缓冲。在这之后，SDL_SetRenderTarget 重新把渲染目的地换回 frameTexture。

2.2 图像

处理图像是个耗 CPU 的操作，为提高效率，必须大范围使用硬件渲染。

2.1.1 OpenGL 渲染 2D 图像

本小节概述 SDL 如何使用 OpenGL 渲染 2D 图像。你不懂 OpenGL，可跳过它，但要知道不仅仅 3D 用硬件渲染，2D 也是。

你想学习 OpenGL，建议购买《OpenGL ES 3.0 编程指南(原书第 2 版)》。该书提供配套源码，一边看书一边调试源码能较好加深理解。对在 Windows 构建工程，书中用的是 CMake，如果不习惯用 CMake 的（就像我），可直接到这里下载 sln。

OpenGL 是当今各主流平台（包括 Windows）都支持的渲染技术，不仅用于渲染 3D 图形，也用在 2D 图像。OpenGL ES 是以手持和嵌入式设备为目标的 OpenGL。这里以渲染一个图像为例子，让遍历 OpenGL ES 的渲染管线。图 2-2 是例子执行情况。

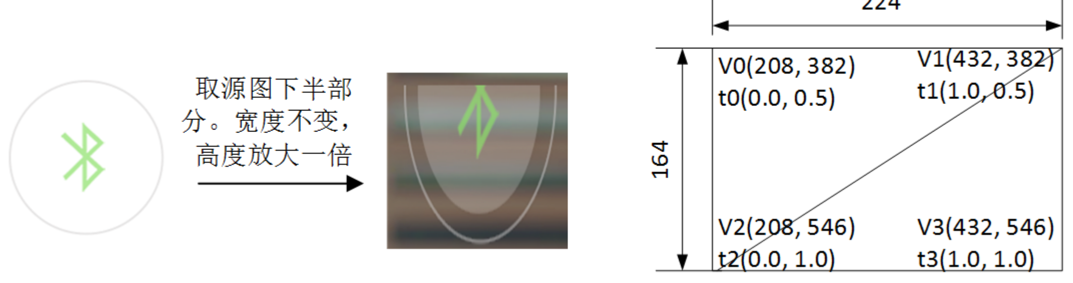


图 2-2 示例

实际编程时，app 直接调用 OpenGL API 还是太烦了，一般会基于某个封装了 OpenGL 的 SDK，像 SDL。

```
SDL_Surface* surf = IMG_Load("ble.png");
SDL_Rect src = ::create_rect(0, surf->h / 2, surf->w, surf->h / 2);
SDL_Rect dst = ::create_rect(208, 382, surf->w, surf->h);
SDL_Texture* tex = SDL_CreateTextureFromSurface(renderer, surf);
SDL_RenderCopy(renderer, tex, &src, &dst);
```

上面是用 SDL 写的把磁盘上的“ble.png”渲染到另一个纹理的帧缓冲区。让就以这代码开始看整个过程。

一、把磁盘文件加载到内存

IMG_Load 是 SDL 提供的加载图像 API，它根据扩展名判断出图像类型，然后分析文件中数据，并放入内存。内存的像素数据须要确定格式，称中间格式，假设中间格式是 ARGB8888，即 4 个分量，依次是 A、R、G、B，每个分量 8 位。注意，ARGB8888 是以小端先存的 uint32 来说的，把它换到 OpengGL 的字节序则是 BGRA。

二、从 SDL_Surface 生成 SDL_Texture

SDL_CreateTextureFromSurface 用于从 SDL_Surface 生成 SDL_Texture。创建过程主要是调用 SDL_CreateTexture。

```
texture = SDL_CreateTexture(renderer, format, SDL_TEXTUREACCESS_STATIC,
surface->w, surface->h);
```

access 参数固定为 SDL_TEXTUREACCESS_STATIC，表示不会有其它纹理叠加到它这里，不用为它创建帧缓冲对象。SDL_CreateTexture 会调用一系列 OpenGL API。

```
int GLES2_CreateTexture(SDL_Renderer *renderer, SDL_Texture *texture) {
    ...
    data = (GLES2_TextureData *)SDL_calloc(1, sizeof(GLES2_TextureData));
    glGenTextures(1, &data->texture);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(data->texture_type, data.texture);
    glTexParameteri(data->texture_type, GL_TEXTURE_MIN_FILTER, scaleMode);
    glTexParameteri(data->texture_type, GL_TEXTURE_MAG_FILTER, scaleMode);
    glTexParameteri(data->texture_type, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(data->texture_type, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexImage2D(data->texture_type, 0, format, texture->w, texture->h, 0, format,
type, NULL);
}
```

激活 GL_TEXTURE0 纹理单元，依次执行生成一个纹理对象，设置缩小、放大模式，当纹理坐标超出[0.0, 1.0]时，x、y 都限定读取边沿，用 glTexImage2D 向该纹理加载图像。由于 pixels 参数值是 NULL，此时的加载只是设置像素格式、图像尺寸，但是，SDL_CreateTexture 很快会用 glTexSubImage2D 上传真正数据。

SDL 的放大、缩小模式，即代码中的 scalMode 变量只可能是 GL_NEAREST 或 GL_LINEAR。当只有这两种时，可以不需要为纹理指定完整的 mip 贴图链，而 SDL 的确也没有使用 mip 贴图。

虽然调用了若干 API，但依旧没开始 OpenGL 管线。当然，已经准备好了可用于片断着色器的一个纹理单元：GL_TEXTURE0。

三、SDL_RenderCopy 把贴图渲染到帧缓冲区

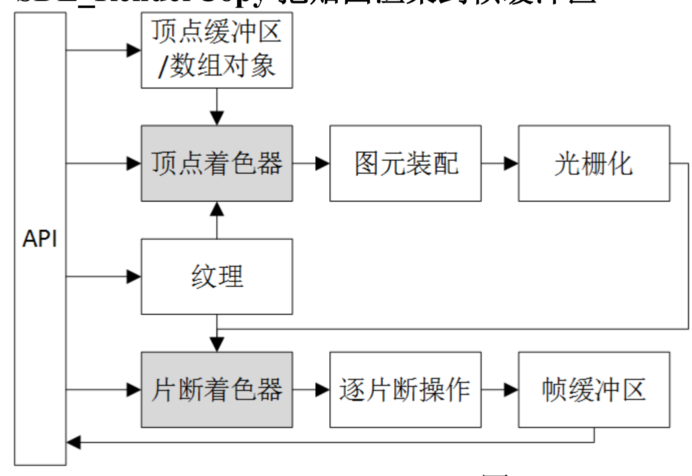


图 2-3 OpenGL ES 管线

SDL_RenderCopy 开始 OpenGL 管线，接下就依次说下各阶段执行的动作。

顶点缓冲区/数组对象

须要哪些顶点属性，这取决于顶点着色器代码，着色器代码是什么？这个内容是预写好的，而且只有一种。

```
const Uint8 GLES2_VertexSrc_Default_[] = " \
    uniform mat4 u_projection; \
    attribute vec2 a_position; \
    attribute vec2 a_texCoord; \
    attribute float a_angle; \
    attribute vec2 a_center; \
    varying vec2 v_texCoord; \
    \
    void main() \
    { \
      float angle = radians(a_angle); \
      float c = cos(angle); \
      float s = sin(angle); \
      mat2 rotationMatrix = mat2(c, -s, s, c); \
      vec2 position = rotationMatrix * (a_position - a_center) + a_center; \
      v_texCoord = a_texCoord; \
      gl_Position = u_projection * vec4(position, 0.0, 1.0);\
      gl_PointSize = 1.0; \
    } \
";
```

对 iOS、Android，SDL 当前支持的是 opengl es2，下表归纳了着色器用到的四个输入属性。

索引	变量	描述
0(GLES_ATTRIBUTE_POSITION)	a_position	顶点位置
1(GLES_ATTRIBUTE_TEXCOORD)	a_texCoord	纹理坐标。直接传给片烦忧着色器
2(GLES_ATTRIBUTE_ANGLE)	a_angle	要旋转的角度
3(GLES_ATTRIBUTE_CENTER)	a_center	旋转时中心坐标

示例中 a_angle、a_center 都是默认值，为此只关心位置 0 的顶点坐标、位置 1 的纹理坐标。因为是 2D，两种属性都是两个分量。

ble.png 尺寸是 224x164。参数 dst 经过视区是否越界检查后，生成的 dstrect 矩形是(208, 382, 224, 164)，以它生成的顶点坐标 vertices：(208, 382; 432, 382; 208, 546; 432, 546)。参数 src 经过图像是否越界检查后，生成的 srcrect 矩形是(0, 82, 224, 82)，以它生成的纹理坐标 texCoords：(0.0, 0.5; 1.0, 0.5; 0.0, 1; 1.0, 1)。绘制图元是用以下代码。

```
data->glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

图 2-2 直观显示了图像前后变化，以及 vertics、texCoords 位置。顶点着色器根据 vertics，对 4 个顶点各执行一次 GLES2_VertexSrc_Default_。由于顶点输入 angle、center 都是默认值，着色器输出的 gl_Position 等于 vertics。v_texCoord 是着色器的另一个输出变量，等于纹理坐标。

图元装配

glDrawArrays 绘制的图元是三角条带，会把这 4 个顶点装配出两个三角形。图元装配中有个处理是裁剪。代码在生成 dstrect 时会确保不越出视区，所以顶点位置不会超出帧缓冲区；生成 srcrect 不会越出图像，所以纹理坐标不会超出贴图。它们保证了绘制出的图像在视景体内部，裁剪时不会发生哪部分被剔除。

光栅化

光栅化是将图元插值出一组片断的过程，说来就是像素化。这里将会产生 36736(224x164)个像素，也就是接下要执行 36736 次片断着色器代码。

片断着色器

片断着色器代码是什么？这个内容是预写好的，和顶点着色器不一样，SDL 预写了多种，依这里情况是使用以下代码。

```
static const Uint8 GLES2_FragmentSrc_TextureABGRSrc_[] = " \
```

```
	precision mediump float; \
	uniform sampler2D u_texture; \
	uniform vec4 u_modulation; \
	varying vec2 v_texCoord; \
	\
	void main() \
	{ \
	  gl_FragColor = texture2D(u_texture, v_texCoord); \
	  gl_FragColor *= u_modulation; \
	} \
";
```

v_texCoord 是从顶点着色器输入的纹理坐标，片断着色器使用它读取纹理中的哪个像素。片断着色器声明一个类型为 sampler2D 的统一变量 u_texture，它将加载一个指定了绑定纹理数据的单元，这个要加载的单元就是前面 SDL_CreateTexture 说到的 GL_TEXTURE0。

逐片断操作

逐片断操作中有个处理是剪裁（Scissor），这里没设过 glScissor，因而不执行剪裁。剪裁之后存在个会产生影响的处理：混合（Blending）。设置的混合模式是 SDL_BLENDMODE_BLEND，它对应以下代码。

```
glEnable(GL_BLEND);
glBlendFuncSeparate(GL_SRC_ALPHA,        GL_ONE_MINUS_SRC_ALPHA,        GL_ONE,
GL_ONE_MINUS_SRC_ALPHA);
```

为更直观，让把代码翻译成公式。

```
Cr = As*Cs + (1-As)*Cd
Ar = 1*As + (1-As)*Ad
```

Cs、As 分别是贴图中像素的颜色和透明度，Cd、Ad 分别是背景中像素的颜色和透明度。

帧缓冲区

逐片断操作生成一个个像素，它们的 R、B、G、A 分量写入帧缓冲区中的颜色缓冲区。

到此管线操作结束，ble.png 这个 2D 图像也渲染到了背景这个帧缓冲区。

2.2.2 像素格式

像素格式用于描述图像彩色空间。包括存在哪些空间分量，像 RGB、YUV；空间中各分量次序，像 RGB、BGR；分量占用位数，像 888。分量中除 RGB、YUV 这些能表示真实颜色的还有个透明度分量：A（Alpha）。

在 SDL，像素格式有两种表现形式，一是 uint32_t 类型 32 位值，二是 SDL_PixelFormat 结构。在相互关系上，从前者解析出后者。

BIT	默认值	描述
D31	1	保留值
D30-D28	0	固定值
D27-D24		像素类型 6：SDL_PIXELTYPE_PACKED32
D23-D20		像素中会出现的各分量及次序（高位到低位） 1：SDL_PACKEDORDER_XRGB 3：SDL_PACKEDORDER_ARGB 7：SDL_PACKEDORDER_ABGR
D19-D16		各分量占用位数 6：SDL_PACKEDLAYOUT_8888
D15-D8		每像素位数 32、24
D7-D0		每像素字节数（以它计算出的位数大于等于每像素位数）

		4

表 1-1 uint32_t 像素格式中各 bit 位的语义

在 D19-D16，命名是以着 uint32_t 的小端序。举个例子，SDL_PACKEDORDER_ARGB 意思是以 uint32_t 存储时，分量序是 A、R、G、B，由于 uint32_t 是小端先存，换到字节序（OpenGL 内部格式）就是 BGRA。

SDL_PixelFormat 是从 uint32_t 位值解析出的结构，是 SDL_Surface 中一个字段。

分配/释放 SDL_PixelFormat

以内存占用上说，单个 SDL_PixelFormat 结构占用 60 字节，单个不算大，但如果要处理数以万计图面（SDL_Surface），每个图面都有这个么 SDL_PixelFormat，这累加起来就大了。以使用经验看，虽然图像格式有好多种，但 app 一次运行时会碰到可能也就十来种，也就是说只要十来个不一样的 SDL_PixelFormat 就可表示碰到的所有像素格式。基于这两个原因，没必要为每个图面都分配 SDL_PixelFormat。SDL 采用单独的链表来管理 app 此次运行碰到的 SDL_PixelFormat，SDL_Surface 存储的只是指向此链表中某一结点的指针。

分配(SDL_AllocFormat)

1）从堆中分配；

2）分配出的堆块被组织成链表。formats 指向链表头。

3）SDL_AllocFormat 分配时，并不是每次都一定从堆中分配出一块。当它发现链表中已存在“相同”像素结构块，它只是把那块的引用计数加一。判断“相同”条件是 SDL_PixelFormat 中 format 值相等。

释放(SDL_FreeFormat)

1）SDL_FreeFormat 释放时，并不是每次都一定从堆中释放出一块。当它发现要释放块的引用计数减一后还大于零，只是简单减一就返回。

2）减一后等于零，则要搜索 formats 链表，删除该 SDL_PixelFormat 结点。

中间格式（Neutral Format）

为什么要有中间格式？图像来源多种多样，即使就硬盘上图像文件，也可能在用不同像素格式，面对这么多种格式，app 须要统一到一种格式，从而方便自个后绪处理，像缩放、裁剪、着色，包括显示。那选什么格式？SDL_PIXELFORMAT_ARGB8888。至少目前来看选它应该没问题，参考“RGBA 还是 BGRA”。

2.2.3 Alpha 混合

一个像素由四分量组成，表示颜色的 R、G、B，表示透明度的 alpha。alpha 混合指的是按照“Alpha”分量值来混合源像素和目标像素。对源像素、目标像素，放到具体使用场景会有不同叫法。当要把一贴图贴到哪背景，源像素往往叫贴图，目标像素叫背景。对 OpenGL 管线中的混合环节，源像素称为输入片断，目标像素则叫已存在片断。为直观，以下会以贴图来等同源像素，背景等同目标像素。

surface 和 texture 都涉及到 alpha 混合，混合在它们表现出的行为基本是一样的。下面是设置混合模式的 API。

```
int SDL_SetSurfaceBlendMode(SDL_Surface* surface, SDL_BlendMode blendMode);
int SDL_SetTextureBlendMode(SDL_Texture* texture, SDL_BlendMode blendMode);
```

SDL 提供了四种混合模式，SDL_BLENDMODE_NONE、BLEND、ADD 和 MOD。综合来看，最常见的是 SDL_BLENDMODE_BLEND。讨论混合公式之前，先定义几个符号。

C - 表示像素的颜色，即(RGBA)的 RGB 部分，C 是 color 的缩写

A - 表示像素的透明度，A 即 alpha。

s - 表示两个混合像素的源像素，s 即 source。

d - 表示两个混合像素的目标像素，d 即 destination。

r - 表示两个像素混合后的结果，r 即 result。

最常见混合模式（BLEND）：Cr=As*Cs + (1-As)*Cd

“Cr=As*Cs + (1-As)*Cd”是颜色分量的计算公式。作为像素的一部分，相应还须要计算 alpha 分量，对如何计算 alpha 分量，即使用这颜色公式的也是不同方法，以下是 SDL 公式。

```
Cr = As*Cs + (1-As)*Cd
Ar = 1*As + (1-As)*Ad
```

这公式的几个特点。1）贴图 alpha 值为 1 时，仅显示贴图，不显示背景。2）贴图 alpha 值为 0 时，仅显示背景，不显示贴图。3）贴图 alpha 值越大，颜色越偏向贴图；alpha 值越小，颜色越偏向背景。

事实已经证明，不可能靠一个公式就囊括所有 alpha 混合，这个最常见混合模式就存在个致命缺陷：它计算颜色时和背景 alpha 值无关，当背景存在 alpha 分量不是 1 的像素时，公式将出问题。

让举个极端例子，当 Ad 等于 0 时，对于混合的最佳结果来说，那就是“Cr=Cs”，可根据“Cr=Cs*As + (1-As)*Cd”，除非 As 等于 1！而且 As 越小，颜色失真越厉害。类似这种场合就要考虑另种计算方法：不混合，即只取贴图。

“Cr=As*Cs + (1-As)*Cd”最好的使用场合是背景中所有像素的 alpha 是 1。

只取贴图（NONE）：Cr=Cs，Ar=As

会遇到这样一种需要，要把一贴图贴到一全透明背景，这时只取贴图是最好方式。它确保了不论颜色还是透明度都不失真。

结合“Cr=As*Cs + (1-As)*Cd”，有人会想到把它稍作修改，即对遇到 Ad=0 时使用只取贴图，其它情况按这公式。这种方法乍看可行，其实没想像中有效。

叠加过程是把图 2 叠加到图 1，图 3 是生成结果。（为清晰在图 3 增加了背景）

```
void BlitRGBtoRGBPixelAlpha()
{
    ......
    if (aalpha == 255 || dalpha == 0) {
        *dstp = *srcp;
    } else {
        ......
    }
    ......
}
```

造成以上问题的原因是满足条件“dalpha == 0”的像素，但这些像素不是看去有异常的中间部分，而是两头！

为直观，让看个输入的 ARGB 是 0x17f4f4f4，然后使用公式“Cr=As*Cs + (1-As)*Cd”。（对 unsigned char 来说，1 等同 256）。

```
*srcp = 0x17f4f4f4;
dR = (sr * alpha + dr * (256 - alpha)) / 256 = (sr * alpha) / 256 = (0xf4 * 0x17)
/ 256 = 0x15;
```

结果值 0x17151515。从看到的说，像素要变暗。让根据这个去解释，月亮两头部分由于

dalpha=0，于是采用“*dstp = *srcp”，即它们的光晕部分保持了图 2 的亮度。月亮中间部分由于 dalpha!=0，于是进入正常 alpha 混合计算像素，结果生成的像素要变暗，即它们的光晕部分要比图 2 的暗。

可以看出，如果只是简单的单独 dalpha=1 时特殊处理，那会造成颜色突变。

如何解决 alpha 混合

1）是全透明背景时，强制使用“只取贴图”。

2）不是情况 1 时，默认使用“Cr=As*Cs + (1-As)*Cd”。

3）不是情况 1 时，提供一种方法让 app 自选某种混合模式，它可以不是 1 也不是 2。

2.2.4 RLE 编码

RLE 是“Run-Length Encoding”缩写，翻译为游程编码，是种无损压缩方案。不压缩的图像数据需要很大内存，举个例子，一张尺寸 1136x640、格式 ARGB 的图像需要 2.7M 字节，更何况 app 可能要同时存储上千张图像。针对图像中存在透明像素这个特点，试着用更有效方法表示透明像素，从而减少图像需要的内存使用。SDL 的 RLE 只压缩内存中的 SDL_Surface，不涉及 GPU 中的 SDL_Texture。

原理

透明像素(transparence)：Alpha 分量是 0 的像素。

半透明像素(translucent)：Alpha 分量>0 并且<255 的像素。

不透明像素(opaque)：Alpha 分量是 255 的像素。

RLE 以行为一个单元进行编码，行和行之间独立、互不影响。

行编码过程及格式（像素数都以两个字节表示）。（1）第 N 行从左到右扫描，形成（透明和半透明像素数，连续的不透明像素数）（不透明像素具体数据）。（2）扫描线回到 N 行零点，从左到右扫描，形成（透明和不透明像素数，连续的半透明像素数）（半透明像素具体数据）。

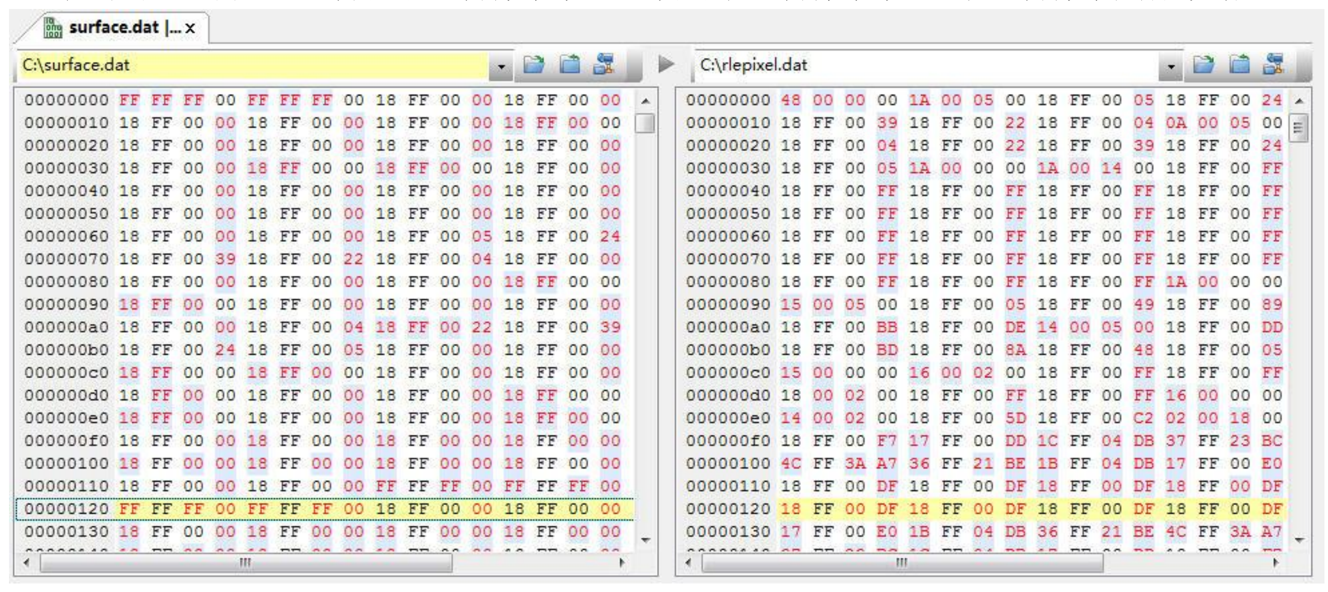


图 2-5 RLE 编码原理

左侧是图像原始数据，格式 ARGB，尺寸 72x72，0000h 到 0119h 是第一行数据。右侧是 RLE 编码后数据。通过解释第一行来看 RLE 是如何工作的。

第一次扫描目的是形成不透明像素。第一行没有不透明像素，于是（透明和半透明像素数，不透明像素数）（不透明像素具体数据）这部分内容只占四个字节：48 00 00 00。“不透明像素具体数据”不占字节。

扫描线回到第一行零点，第二次扫描目的是形成半透明像素。这一行有数段半透明数据。

```
#1：（26， 5）。这一段前面有 26 个连续“透明和不透明像素”，然后是连续 5 个半透明像素。RLE 编码后数据就是：1A 00 05 00。对于紧跟的“半透明像素具体数据，就是后面的 20（4 * 5）个字节：18 FF 00 05 18 FF 00 24 18 FF 00 39 18 FF 00 22 18 FF 00 04。至此这一段形成 00h 到 1bh 字节。
#2：（10， 5）。这一段前面有 10 个连续“透明和不透明像素”，然后是连续 5 个半透明像素。RLE 编码后数据就是：0A 00 05 00。对于紧跟的“半透明像素具体数据，就是后面的 20（4 * 5）个字节。
#……
```

编码就是这样一行一行持续下去。

RLE 编码是针对透明像素，编码思路是要减少存储透明像素需要的字节数。透明像素越多，压缩率越高，要是图像中没有透明像素，编码后形成的数据量比原图像还大。

边解码边复制

对经过 RLE 编码后图面，经常执行的一种操作把它作为贴图，块移到背景。以下描述如何把 RLE 编码后的贴图块移到背景，这过程称为边解码边复制，理解了它，同时也就理解了 RLE 解码。

过程是从上到下一行一行地复制，复制各行的行为是一样的，当复制完所有需要的行后，块移结束。

假设有这样一行 RLE 数据：不透明部分有两块，分别是[(ofs#0,run#0)op#0]、[(ofs#0,run#0)op#0]，半透明只有一块，是[(ofs#0,run#0)tp#0]。操作首先处理不透明。根据 ofs#0 定位出此块的第一个不透明像素位置，然后一连复制 run#0 个像素，数据在 op#0。然后是第二块，根据前面的“ofs#0+run#0”以及“ofs#1”定位出此块的第一个不透明像素位置，然后一连复制 run#1 个像素，数据在 op#1。处理完第二块后，ofs#0+run#0+ofs#1+run#1 等于了图像宽度，复制不透明像素结束，接下是复制半透明像素。根据 ofs#0 定位出此块的第一个半透明像素位置，然后一连复制 run#0 个像素，数据在 tp#0。处理完这一块后，ofs#0+run#0 已等于图像宽度，复制半透明像素结束。——至此复制完了此行所有数据。

虽然都叫复制，不透明、半透明复制行为是不一样的。不透明是真正的内存直接覆盖，半透明则是要和背景像素进行 alpha 混合。至于透明像素，不做任何操作。

复制不需要先解码出整个图像，而是一边解码一边复制。复制结束后，贴图数据依旧保持着原来 RLE 编码后的样子。

贴图存在裁剪矩形时，如果矩形的“y+h”小于贴图高度，“>=y+h”的行不需要解码，其它行则一定要解码，即使是“<y”的行。

此时的背景不能是 RLE 编码后的图面。

复制需要的额外内存。基本不须要额外内存。

复制需要的额外 CPU。每一行会多出数个判断、循环、地址加法，如果一行只有一、两块，那基本不多消耗 CPU。而且如果透明像数多，这种方法直接略过了复制透明像素，相比不编码方案是节省了 CPU。

对压缩后到底有多少字节，最坏情况是图像没有透明像素，而且半透明、不透明像素交替出现。以下公式大致能计算出此时字节数。

```
maxsize = surface->h * 2 * 4 * (surface->w + 1) + 4;
```

背景格式、RLEDestFormat

为避免复制半透明像素时分量匹配，RLE 编码依赖于背景图面。复制半透明像素时，执行的是 alpha 混合，假如贴图是 ARGB，背景是 ABGR，这意味着复制每个像素时都要进行分量互换。为避免复制时分量互换，SDL 用了编码时和背景相关。map->data 存放 RLE 编码出的数据，但它前面的 sizeof(RLEDestFormat)字节存储着一个来自背景格式的 RLEDestFormat 结构，指示了紧挨着的数据的格式。由于编码和背景相关，一旦下次要块移动另一个背景，它要依次进行解码、编码、然后复制。

当然，如果下一次块移到的依旧是上次背景，那只须进行复制。

什么时候用 RLE

从以上分析可看出，RLE 不能包治百病，像没有不透明像素的图面，用 RLE 只会多耗内存和 CPU。SDL 提供了专门 API（SDL_SetSurfaceRLE），让 app 选择针对某个图面要不要使能 RLE。当在决定是不是要使能 RLE 时，除了要看图面中有多少透明像素，还要考虑它被块移时，是不是经常换背景，那会导致多次解码、编码，这多出的 CPU 消耗可能会甚过用 RLE 节省下的那点内存。以下几种情况不推荐使用 RLE。

不会作为贴图的图面。

很少透明像素的图面。

块移到的背景要经常变换的图面。结合上条“很少透明像素”，对小尺寸图像不建议使用 RLE，像长度乘上宽度小于等于 1024。

半透明像素需要使用 NONE 混合的图面。RLE 处理半透明像素会强制使用 BLEND，这对 NONE 混合才正确的图面可能会造成颜色失真。

SDL_LockSurface/SDL_UnlockSurface

这两个函数用于加锁、解锁图面。当 app 试图直接访问一个图面的像素数据时，最好调用下这对函数。

一个使能了 RLE 的图面作为贴图被块移后，它存储的像素数据将保持着 RLE 格式，像 pixels 字段是 NULL，map 字段中的 blit 指向 RLE 块移解码函数（SDL_RLEBlit/SDL_RLEAlphaBlit），flags 字段中会有 SDL_RLEACCEL 标志指示该图面存储的是 RLE 格式。相应地，系统也可能存在非 RLE 格式的 surface，这些图面的 pixels 指向未压缩的 ARGB 数据，flags 字段中没有 SDL_RLEACCEL 标志。因此，app 要同时面对两种图面格式：RLE 和非 RLE。

为实现要求的功能，app 必须知道要处理的图面是 RLE 还是非 RLE，像要减小所有像素的 Alpha 值，以让变得更透明。它们是针对图面每像素进行操作，对这种操作，非 RLE 时，pixels 字段存的就是未压缩的 ARGB，直接编辑就可以，而 RLE 图面则先要进行 RLE 解码，为方便后续处理，编辑完像素后还要进行 RLE 编码，恢复回原来的 RLE 状态。于是它们就形成了一种固定操作模式。

1）判断要处理图面是 RLE 还是非 RLE，是 RLE 的进行解码；

2）执行要求的像素处理操作；

3）此图面原来是 RLE 的进行 RLE 编码，让恢复回原状态。

SDL_LockSurface/SDL_UnlockSurface 就用于以上这个目的，SDL_LockSurface 实现第一步，对图面加锁，SDL_UnlockSurface 实现第三步，对图面解锁。很显然，要实现第三步解锁，它必须知道此个被加锁的图面原来是 RLE 还是非 RLE，那么是哪个字段指示了这个标志？以下是两函数代码。

```
int SDL_LockSurface(SDL_Surface * surface)
{
  if (!surface->locked) {
    if (surface->flags & SDL_RLEACCEL) {
      SDL_UnRLESurface(surface, 1);
      surface->flags |= SDL_RLEACCEL; /* save accel'd state */
    }
  }
  ++surface->locked;
  return (0);
}

void SDL_UnlockSurface(SDL_Surface * surface)
{
  if (!surface->locked || (--surface->locked > 0)) {
    return;
```

```
  }
  if ((surface->flags & SDL_RLEACCEL) == SDL_RLEACCEL) {
    surface->flags &= ~SDL_RLEACCEL;   /* stop lying */
    SDL_RLESurface(surface);
  }
}
```

SDL_LockSurface/SDL_UnlockSurface 没有用额外变量来存储图面先前状态，而是采用继续在 flags 中置上 SDL_RLEACCEL！这样操作会导致一个 RLE 图面经过 SDL_LockSurface 后是被解压了，但 flags 中仍旧有 SDL_RLEACCEL 标志，这不符合块移逻辑要求（块移逻辑一看到 flags 中有 SDL_RLEACCEL，会率先执行 RLE 解码）。由于这个不符合，加锁后的图面不能用于块移，加锁/解锁的使用场合是上层想直接访问、修改像素数据，就像以上说的修改图像透明度。而一旦访问像素结束后，必须调用 SDL_UnlockSurface 进行解锁，这不仅是遵从要求恢复回 RLE，更是要让图面的 flags 恢复到“真话”状态。

仔细看 SDL_LockSurface/SDL_UnlockSurface 会发现，当要处理图面是非 RLE 时，它的加锁、解锁只不过是修改 locked 字段，上层要是不想管 locked，可说就没作用，即对非 RLE 图面的加、解锁可说是空操作。为配合是不是有必要必须执行加、解锁，SDL 提供了一个宏，当要直接访问一图面的像素数据时，程序可调用该宏来知道是否有必要进行加、解锁。

```
#define SDL_MUSTLOCK(S)  (((S)->flags & SDL_RLEACCEL) != 0)
```

虽然宏名叫 SDL_MUSTLOCK，该宏的本质是判断要处理的图面是 RLE 还是非 RLE。通过该宏，再结合 SDL_LockSurface/SDL_UnlockSurface 必须成对出现要求，程序可设计一个自动给待处理图面加、解锁的类，这类就构造、析构这两个函数。

```
surface_lock::surface_lock(surface &surf) : surface_(surf), locked_(false)
{
    if (SDL_MUSTLOCK(surface_))
        locked_ = SDL_LockSurface(surface_) == 0;
}

surface_lock::~surface_lock()
{
    if (locked_)
        SDL_UnlockSurface(surface_);
}
```

为此当应用程序要直接访问图面 surf 的像素数据时，它只要在访问前以 surf 为参数创建 surface_lock 对象就行了。

```
surface_lock lock(surf);
```

一旦 lock 作用域失效，像所在函数退出了，系统会调用析构函数，从而实现自动解锁。

对 SDL_LockSurface/SDL_UnlockSurface 需要重申的是：**处于加锁中图面不要进行任何的块移操作**。

RLEAlphaSurface 依赖 map->dst 导致的潜在问题

RLEAlphaSurface 用于执行 RLE 编码，编码要使用 map->dst 字段，dst 指向最近一次块移到的背景图面。要额外说明的是 RLE 解码不需要 map->dst。

但是，如果不加额外操作，surface->map-dst 可能是无效的！

RLEAlphaSurface 用于两种场合，一是块移(blit)，二是 SDL_UnlockSurface。在块移时，surface->map->dst 总是有效的，但在 SDL_UnlockSurface 可能会变成无效。让看以下这样一个操作序列。

1）A 是一个 RLE 图面。执行一次到图面 C 的块移。块移结束后，A 的 map->dst 指向 C。

2）C 图面被释放。但没有更新 A 中的 map->dst 字段！（要 C 在释放同时更新 A 的 map->dst，这要求太过苛刻。）

3）程序试图直接访问图面 A 的像素数据，于是调用 SDL_LockSurface/SDL_UnlockSurface。当执行 SDL_UnlockSurface 时，dst 指针值虽然非 NULL，但它指向的图面其实已经无效，还用这个无效指针，执行后会倒致程序崩溃。

SDL 对这问题采取的办法是给块移过的背景图面增加额外引用计数，用这引用计数去确保在贴图删除前，dst 一直有效。

步骤一：修改 SDL_MapSurface，增加 dst 的引用计数

贴图要以 RLE 方式块移到新背景时，它会调用 SDL_MapSurface。

```
int SDL_MapSurface(SDL_Surface * src, SDL_Surface * dst)
{
  /* Clear out any previous mapping */
  SDL_BlitMap *map = src->map;
  if ((src->flags & SDL_RLEACCEL) == SDL_RLEACCEL) {
    SDL_UnRLESurface(src, 1);
  }
  SDL_InvalidateMap(map);
  ......

  map->dst = dst;
  if (map->dst) {
    ++map->dst->refcount;
  }
   ......
  /* Choose your blitters wisely */
  return (SDL_CalculateBlit(src));
}
```

具体看“if (map->dst)”部分，它给 dst 增加引用计数，这个增加使 dst 不会被删除。

步骤二：修改 SDL_InvalidateMap，减少 dst 的引用计数

dst 是 map 中字段，而要释放 map 则一定须调用 SDL_InvalidateMap。

```
void SDL_InvalidateMap(SDL_BlitMap * map)
{
  if (!map) {
    return;
  }
  if (map->dst) {
    if (--map->dst->refcount <= 0) {
      SDL_FreeSurface(map->dst);
    }
  }
  map->dst = NULL;
  ......
}
```

步骤一给 dst 增加了个额外引用计数，这里和它配对减少 dst 的引用计数，如果降到没有了则调用 SDL_FreeSurface 释放背景图面。

2.2.5 管理图面、纹理

当程序中存在大量图面时，管理它们就变得重要了。

- 缓存图面数据。从图像文件到生成 SDL 图面需要耗一定 CPU，使用缓存、通过牺牲内存来降低 CPU 使用率。
- 生成的图面要能灵活改变尺寸。像游戏中大地图栅格，在 1280x800 大分辨率下，尺寸是 72x72，但在 480x320 小分辨率下是 48x48。这个 48x48 是 72x72 经过缩小后得到，也就是说从文件得到 48x48 要经过两个步骤，当中就可使用缓存从而减少步骤。
- 生成的图面要能灵活改变颜色。像一天有白天、黑夜，在不同时断下同样内容黑夜中颜色要比白天暗。如果白天图面是文件直接生成的图面，那黑夜图面是白天图面经过调整颜色后得到，也就是说从文件得到黑夜图面要经过两个步骤，当中可使用缓存从而减少步骤。

管理图面是通过某种数据结构，数据结构的优劣直接决定了管理效率。衡量管理效率可使

用四个指标。1）随机访问效率。2）判断命中/未命中效率。3）缓存不可能无休止膨胀，为制止膨胀采用的释放算法。4）管理结构自身占用的内存大小。

几个术语

哈希表（Hash table，也叫散列表）：是根据关键码值(Key value)直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表（哈希表）。

开放寻址法：它是哈希表处理冲突的方法。公式：$H_i = (H(key) + d_i) \bmod M, i=1,2,\cdots, k(k \le M-1)$。key 是关键码，H(key)是散列函数，M 是哈希表长度。

关键码：哈希表中的关键码值。具体到这里的算法是 hash、hash1 这两个整数值。

特征值：位在存缓外部，由它生成关键码。具体是 image::locator 中用于形成关键码的那些字段。包括图像文件名、类型、修饰字符串、坐标等。

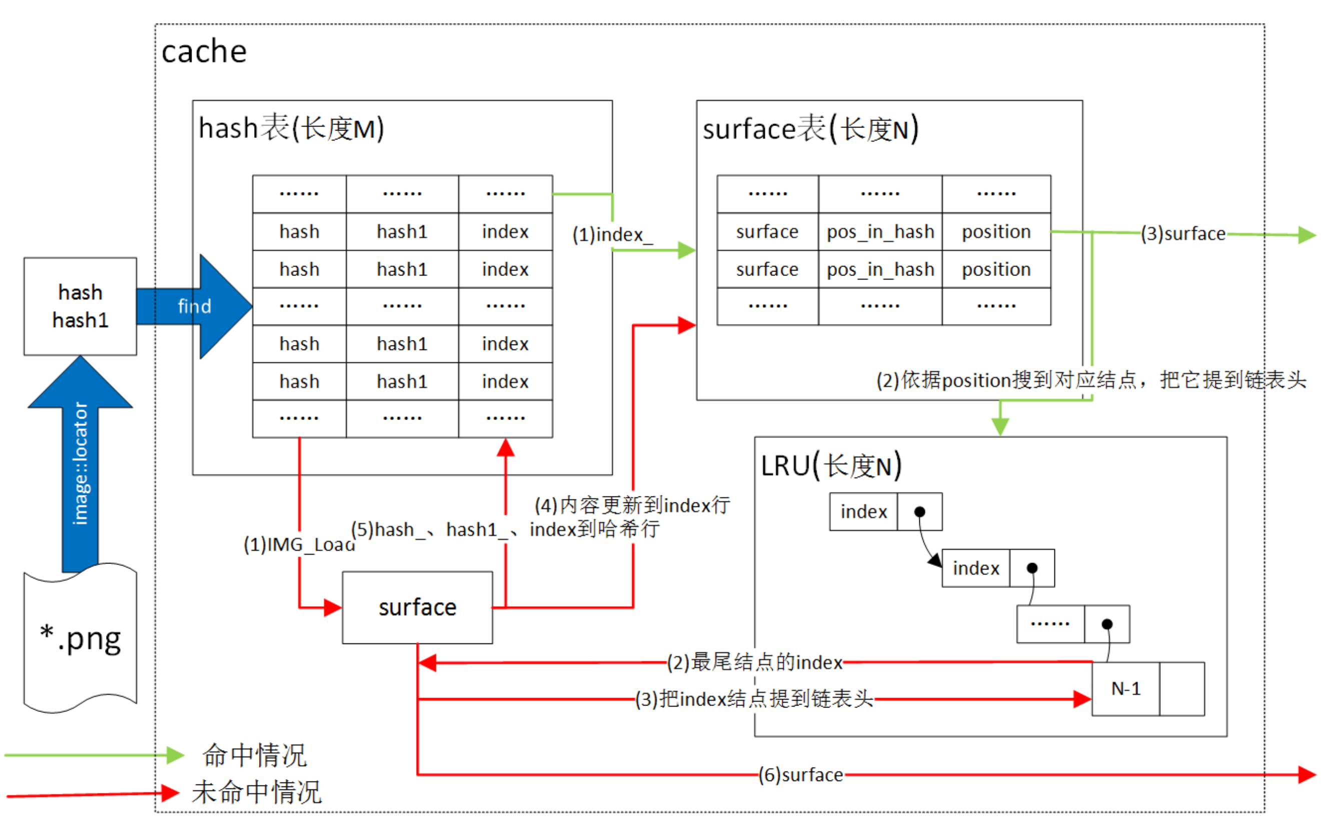


图 2-6 管理框图

流程描述

cache 由一个哈希表、一个 surface 表、一个 LRU 链表组成，构造 cache 时这三个表都有了固定长度的单元，即哈希表长度 M，surface 表、LRU 长度是 N。

命中情况：一个*.png 文件形成一个 image::locator 对象，根据 locator 中特征值形成关键码<hash, hash1>。把关键码映射到哈希表中发现 j 记录和它匹配，命中！取出 j 记录中的 index 字段，以它为索引在 surface 表定位出 index 行，根据 index 行中的 position 字段定位到 LRU 中的 index 结点，把该结点提到 LRU 链表头。最后取出 index 行中 surface 给调用程序。

未命中情况：一个*.png 文件形成一个 image::locator 对象，根据 locator 中特征值形成关键码<hash, hash1>。把关键码值映射到哈希表中发现没有记录和它匹配，未命中！调用 IMG_Load 由图像文件生成 surface，从 LRU 中依释放策略得出 index，由于 index 结点要成为最新访问结点，LRU 中要把 index 结点提到链表首，相应的要把此次 surface 更新到 surface 表中 index 行。接下是更新哈希表，再次以关键码<hash, hash1>定位哈希表，找到第一个空行，把此次形成的<hash, hash1, index>更新该行。最后把 surface 返回给调用程序。

A：image::locator 是什么？特征值有哪些？

Q：image::locator 是个封装图像文件的类，它不产生 SDL 图面，但它提供了在 cache 中定位出它所包含的图像对应的 SDL 图面的方法。一个 image::locator 对象对应着磁盘上一个图像文件。struct image::value 中的字段都是特征值。

filename_：图像文件名。它是特征值主要部分。存储已经去掉了修饰（如果参数给的文件名有修饰，修饰就会被放在 modifications_）的文件名。

type_：enum 类型，值或是 FILE，或是 SUB_FILE。FILE、SUB_FILE 区别在于要不要对从整个图像文件产生的图面进行后续处理。那什么时候会认为是 SUB_FILE？1）构造 locator 时就带了 modifications 或 loc 字段。2）从构造 locator 的参数 filename 中找到了“~”字符，即认为存在 modifications_。

SUB 的字面意思是“子”，但 SUB_FILE 中 SUB 意义不是表示 SUB_FILE 得到图面是 FILE 的子图面，而是说此次得到的图面要在 FILE 图面上进行再处理。当然，如果这个处理是裁剪（~CROP），则恰好符合了“子”这个概念。modifications_、loc_字段指示要执行什么再处理。

modifications_：std::string 类型，它存储以字符串格式指示的再处理。像裁剪（~CROP），缩放（~SCALE）、颜色势力化（~TC）、翻转（~FL）、灰度化（~GS），等等。

loc_：map_location 类型，它用于实现移位。为实现移位，还须要 center_x_、center_y_两个 int 字段。移位再处理在 loc_.valid()==true 时才发生作用。

FILE 得到的图面肯定是整个图像文件产生的图面。SUB_FILE 则要看再处理操作，如果再处理是空操作（modifications_为空并且 loc_.valid()==false），它得到的图面就是 FILE 图面。

A：关键字为何有两个？如何计算 hash、hash1？

Q：关键字中有两个值是为了让不同特征值却产生相同关键字的概率趋向零。

作为硬性规则，要在哈希表定位，必须使用整数作为关键字。如果不同特征值一定得能到不同整数，那自然是最理想的，可这里遇到的特征值（locator）中存在多种字符串，像文件名、modifications，要把字符串转为整数，没法做到一对一。作为补充算法，对同一特征值用两个独立函数分别计算出哈希值：hash、hash1。不同特征值对应到同一个 hash、或同一个 hash1 已是小概论事件，要同时对应，则概率趋向零，于是可认为 locator 和 <hash, hash1>)是一一对应。

hash 计算方法是通过 Boost 库提供的 hash 算法。hash1 则是对文件名进行简单的 RLEHash。

A：哈希映射函数和处理冲突方法？

Q：映射函数 y= hash MOD M。M 是哈希表长度，hash 是两个关键字中的 hash，即 Boost 库计算出的哈希值，相比 hash1，它有更好均匀特性。

处理突冲方法是当 y = hash MOD M 计算出的 y 已被使用时（index!=-1）执行 y=y+1，如果这个 y 还是被使用，再执行 y=y+1，直到找到空闲记录。

综合映射函数和处理冲突方法，就是哈希表常见的开放寻址，Hi = (H(key) + di) MOD M, i=1,2,…, k(k<=M-1)。

在找空闲记录时，映射函数 y= hash MOD M 就够了，但要取属于自己的记录时，还要加上校验该记录上的 hash1 值，只有满足了该记录中的 hash1 等于自个计算出的 hash1 才能认为是属于自己的记录。

A：LRU 表作用？它在溢出策略中起什么作用？

Q：LRU 全称 Least Recently Used，即最近最少使用。引入它就是实现溢出策略。

LRU 表使用规则：链表中结点从头到尾的排列次序表示了结点的使用情况，越靠头结点是最靠近现在使用过的，也就是说，最后一个结点是最不常用的，当要溢出时就释放该结点。

表中结点的值是 surface 表索引。

LRU 总是保持一个固定长度 N，这个长度就是 surface 表长度。因而当 LRU 表擦掉最尾结

点时，只要它发现该结点指向了哈希表中一条记录，会把那记录置为无效，意味着 LRU 表在处理 surface 表溢出时同时解决掉了哈希表的溢出问题。**LRU 保证了哈希表中最多只存在 N 条有效记录。**

```
template<typename T>
int cache_type<T>::add(const T &item, size_t hash, size_t hash1)
{
    // in hash_table, find a empty position.
    size_t pos = hash % hash_table_size;
    while (hash_table_[pos].index != -1) {
        if (++ pos == hash_table_size) {
            pos = 0;
        }
    }

    // calcuate index of content_
    int index = lru_list_.back();

    cache_item<T>& elt = content_[index];
    if (elt.pos_in_locator_table != -1) {
        hash_table_[elt.pos_in_hash_table].index = -1;
    }
    elt.item = item;
    elt.pos_in_hash_table = pos;

    lru_list_.erase(elt.position);
    lru_list_.push_front(index);
    elt.position = lru_list_.begin();

    // fill (hash,hash1,index)
    hash_table_[pos].hash = hash;
    hash_table_[pos].hash1 = hash1;
    hash_table_[pos].index = index;

    return index;
}
```

add 函数处理把 item 加入 surface 表。它主要分三个步骤：

1. 在哈希表中找到一个空闲记录，pos 指示了这条记录。
2. 处理 surface 表和 LRU。index 直接取自 LRU 最后一个结点，如果该结点指向哈希表记录，就把该记录置为无效。item 赋给 surface 表中记录。接下对 LRU 处理就是删除最后一个结点，然后再表头新加一个结点。
3. 此次形成的<hash, hash1, index>更新到哈希表相应记录。

A：如何表示哈希表中记录是占用状态还是空闲状态？状态之间如何转变？

Q：哈希表记录中有个 index 字段，-1 表示空闲，非-1 时表示占用。

　　初始时所有 index 是-1，整个表是空的。

　　空闲到占用：当有记录要被占用时，index 值被置为一个大于等于零的值，这个值范围[0, N-1]（N 是 surface 表长度），它正是 surface 表的索引。

　　占用到空闲：LRU 链表，也可认为 surface 表满时，当又来新记录，它就须要无效掉一条记录，surface 表那条要被无效的记录通过内部一个变量（pos_in_hash_table）无效掉它在哈希表中的对应记录，也就是把那记录的 index 值置为-1。

A：如何处理冲突地址上意外无效？

Q：让看一种情况：图像 P 计算出的关键值是 6，定位到 6 记录，结果发生冲突，依据冲突算法本应在位置 6 的被移动位置 10，接下出现 7 图像被释放，当此后 P 图像再来访问，搜到 7 时，发现它是空的，于是就不再搜了，认为 P 不在哈希表中，不命中！可 P 图像其实在哈希表。把这种情况叫冲突地址上意外的无效。

解决办法有两种。1）检查到 7 时，虽然是无效但还是继续搜。当然不可能无休止搜下去，必须约定一个次数，超过遇到这个数的-1 才认为确实没在哈希表。2）加大哈希表长度，减少冲突机率。

哈希表度长度要定为多少？哈希表中最多只有 N 条有效记录，N 是 surface 表长度。哈希表越长，出现冲突机率越低。

遇到多少个-1 才认为确实不存在？这个数设得越大，误判不命中次数越少，但设得越大，无疑会增大判断命中/不命中时间，导致算法效率变差。当前这个值是 4，以下是两组经验数据。

surface 表长度	1500	1500
哈希表长度	4500	6000
运行时间	60 秒（一直在滚动地图）	60 秒（一直在滚动地图）
最大冲突长度	21	13
忽略 1 个-1 后从不命中变为命中次数	77286	108729
忽略 2 个-1 后从不命中变为命中次数	10356	29822
忽略 3 个-1 后从不命中变为命中次数	X	706

A：使用 cache，如何减少改变过尺寸/颜色图面的生成步骤？

Q：有个地形图像 P，在 1280x800 大分辨率下，它是 72x72 尺寸，但在 480x320 小分辨率下是 48x48。这个 48x48 是 72x72 经过缩小后得到，也就是说从文件得到 48x48 要经过两个步骤，这时就可以为每个步骤使用一个 cache。

images_：存的是直接从磁盘图像文件生成的 72x72 图面。

scaled_to_hex_images_：存储 72x72 图面经过缩小后图面，像 48x48。

当主程序序分辨率发生变化，希望是 56x56 时，scaled_to_hex_images_将全部清空以存储新的 56x56，56x56 也要比 72x72 缩小而来，由于 images_依旧有效，到此第一步骤时全部命中，等于减少一个步骤。

改变颜色基于的也是同一过程，多使用一个 cache：tod_colored_images_。

A：cache 中的 clear_cookie_变量是干什么用的？

Q：clear_cookie_指示了在 flush 时是否要无效掉 surface 表中的所有记录。

当程序改变环境时，它会调用 cache 的 flush 操作，像下一回合，下一回合可能从黄昏变为黑夜，像改变了分辨率，要求格子尺寸从 72x72 变为 48x48。而当碰到这些操作时，像黄昏变为黑夜，tod_colored_images_这个 cache，它存的是之前改了颜色图像，即黄昏时图像，变为黑夜，颜色要更暗，但由于颜色这个参数不是 image::locator 特征值，即黄昏和黑夜下，同一图像它的 hash、hash1 值是一样的，要是不无效掉 tod_colored_images_就会使得黑夜下取某个图像，它在 tod_colored_images_命中，取出的却是黄昏时图像！

只有 images_这个 cache 的 clear_cookie_置为 false，它可说是直接从磁盘得到的图面，不经过缩放、不经过改色，一直有效，而为了让缓存不断发挥作用，它在 flush 时不无效 surface 表。

管理效率

让按开头写的四个指标看下此种算法的管理效率。

随机访问效率。由特征值计算出 hash、hash1，hash 模 M 得到哈希表中一个位置，接下是数个加法处理冲突，最坏的加法次数等于最大冲突长度。

判断命中/未命中效率。由特征值计算出 hash、hash1，hash 模 M 得到哈希表中一个位置，由于会出“冲突地址上意外无效”，致使遇到一个-1 时还要继续搜，要执行更多加法。最坏加法

次数等于最大冲突长度*可认为确实不存在的-1 个数。

缓存不可能无休止膨胀，为制止膨胀采用的释放算法。要做的操作是删除 LRU 链表尾结点，向 LRU 头加入一个结点。会保证释放的是最长时间未使用的图面。

管理结构自身占用的内存。以 surface 表长度等于 1500，哈希表长度等于 6000 为例。哈希表占用内存：长度*一条记录字节数 = 6000*12 = 70.2125K。surface 表占用内存：长度*一条记录字节数 = 1500*cache_item 对象长度。LRU 表占用内存：长度*一条记录字节数 = 1500*std::list<int>一个节点长度。

小结

管理结构自身内存还有点的，加上程序中还需要多个 cache，内存占用成倍上升。为此 cache 数量能省要省，像确定不希望进行地图编辑的 iOS 系统就不要 semi_brightened_images_。

“冲突地址上意外无效”带来的不仅仅是降低命中/未命中判断效率，更是会导致误判。虽然加大哈希表长度、增加认为确实不存在的-1 个数可以减少误判，但解决不了根本问题。

M >=N。哈希表长度 M 必须大于等于 surface 表长度 N，否则判断是否命中时会进入死循环（locator::in_cache(...)）。加上命中效率是以判断时能够多快遇到“-1”来衡量，出现更多“-1”则要求 M 比 N 越大越好，因而实际使用时 M 会数倍于 N，默认是 4 倍。

2.2.6 images、unscaled_textures 和 hex_masked_textures

```
cache_type<surface> images(false);
cache_type<texture> unscaled_textures;
cache_type<texture> masked_textures;
```

把磁盘文件渲染到屏幕要经过三个步骤，1）文件加载到内存形成 surface，2）surface 生成 texture，3）texture 渲染到屏幕。images 就用于缓存第一阶段生成的 surface，unscaled_textures 缓存第二阶段生成的 texture，那 masked_textures 作用是什么？

masked_textures 用于缓存经过栅格形状掩码后的纹理。什么是栅格形状掩码？在彩图 3，大地图中栅格是正六边形，为正确显示，地形图面在被生成为纹理前要被一个正六边形图面掩码。

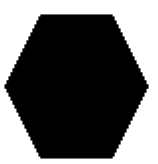


图 2-7 alphamask.png

直观解释掩码操作就是把 alphamask.png 盖在图面上，和 alphamask.png 中非透明位置对应的像素让通过，其它屏蔽掉。masked_textures 就用于存放由这些掩码后图面生成的纹理。在这里，有人当心一个图像的纹理被两处都缓存了，导致浪费 GPU，——它们缓存的是两类不同用途的纹理，很少有“重叠”。彩图 3 中，masked_textures 缓存的是地形图像，像水、草地、沙漠，森林，其它则放在 unscaled_textures，包括站栅格上的部队、窗口控件中图像。

代码是如何把图像分到这两个纹理缓存？说它之前让看下大地图中栅格有两种尺寸，配置尺寸（tile_size）和显示尺寸（zoom）。配置尺寸用在制作素材时，所有地形按这尺寸去绘画。每个 app 依着自个需要决定这个尺寸，像彩图 3 用的是 72，彩图 1 用的是 128。显示尺寸是 app 运行时栅格正显示的尺寸。显示尺寸除以配置尺寸等于缩放系数。

```
double display::get_zoom_factor() const {
    return (double)zoom_ / image::tile_size;
}
```

要渲染大地图中的图像，代码一定会调用 render_locator_texture，当然它也可能渲染大地图外图像。

```
void render_locator_texture(texture& screen, const tblit& blit, int dstx, int
dsty, const SDL_Rect* clip_rect)
```

blit 指示了要渲染素材，当中有个 loc_type 字段，代码就是按这字段把纹理放到了不同缓存。

loc_type	Cache	最后尺寸	裁剪	使用场合

UNSCALED	unscaled_textures	图面尺寸	支持	直接从磁盘加载图像生成的图面
SCALED_TO_ZOOM	unscaled_textures	w = (tex_width * zoom) / tile_size; h = (tex_height * zoom) / tile_size;	支持	须要按缩放系数变化尺寸的图像，彩图 3 中站栅格上的部队
SCALED_TO_HEX	masked_textures	w=zoom, h=zoom	不支持	栅格中地形
TOD_COLORED	masked_textures	w=zoom, h=zoom	不支持	栅格中地形。SCALED_TO_HEX + 修改颜色。修改根据的是 red_adjust、green_adjust、blue_adjust
BRIGHTENED	masked_textures	w=zoom, h=zoom	不支持	SCALED_TO_HEX + 高亮 + 修改颜色。修改根据的是 red_adjust、green_adjust、blue_adjust

注 1。裁剪指的是用 clip_rect 参数。

app 如何指定 loc_type？loc_type 用在场景编程，app 向指定层增加要渲染的 tblit 时，需要调用 display::drawing_buffer_add，后者会有个 loc_type 参数。

2.2.7 渲染 API

```
void render_blit(SDL_Renderer* renderer, const image::tblit& blit, const int xpos, const int ypos);
void render_surface(SDL_Renderer* renderer, const surface& surf, const SDL_Rect* srcrect, const SDL_Rect* dstrect);
void render_line(SDL_Renderer* renderer, Uint32 argb, int x1, int y1, int x2, int y2);
void render_rect(SDL_Renderer* renderer, const SDL_Rect& rect, Uint32 argb);
```

从函数名大概就能知道它们有什么不同，rendere_blit 渲染的是 blit，render_surface 是图面，render_line 是画直线，render_rect 则是画矩形。后面三个容易理解，render_blit 的功能是渲染 blit 指定内容，blit.type 是 image::BLITM_LOC 时会自动使用 cache，参数 x、y 指示贴图要放在的左上角的后半部分。要使用它，app 要做的是理解 tblit 这个结构中各字段意义。

字段	类型	语义	_LOC	_SURFACE	_RECT	_LINE
type	int	素材类型				
x	int	左上角坐标 x 前半部分	必须	必须	必须	必须
y	int	左上角坐标 y 前半部分	必须	必须	必须	必须
clip	SDL_Rect	裁剪矩形	可选	可选	x	x
surf	surface	要渲染的图面	x	必须	x	x
width	int	最后宽度	可选	必须	必须	必须
height	int	最后高度	可选	必须	必须	必须
loc	const locator*	封装磁盘文件的 locator	必须	x	x	x
loc_type	TYPE	类型	必须	x	x	x
flip	int	如何翻转	可选	x	x	x
modulation_alpha	int	对 alpha 的调制比例	可选	x	x	x
blend_ratio	uint8_t		可选	x	x	x
blend_color	uint32_t		可选	x	必须	必须

为什么存在两部分 x、y？有时要渲染 std::vector<tblit>，像 blit 风格的图像控件，这时 vector 中 tblit 共用 render_blit 中的 x、y，tblit 中的 x、y 则是私有偏移。

modulation_alpha、blend_ratio、blend_color 是三个和修改颜色相关的字段。BLITM_RECT、BLITM_LINE 时，blend_color 用于指定画矩形/线条时要使用的颜色（包括 alpha 分量），它比较好理解，这里着重说下 BLITM_LOC 时。以下公式说明了这三个字段如何发挥作用，As、Cs 是原来的 alpha、rgb；Ar、Cr 是生成的 alpha、rgb。

```
Ar = modulation_alpha / 255 * As;
Cr = Cs + blend_color * blend_ratio / 255;
```

modulation_alpha 默认值是 255，指示对 alpha 的调制比例。这里要注意，当前只支持 modulation_alpha 小于 255，即只能把 alpha 变小！blend_ratio、blend_color 则用于修改 argb 中的 rgb。这里要注意，当前只支持变大 rgb，也就是说，只能让分量颜色变深，或使得颜色更白（rgb 一块变大）。

render_blit 根据 type 不同去调用不同执行函数。BLITM_LOC 时调用 render_locator_texture，BLITM_SURFACE 对应 render_surface，所以理论上渲染图片、矩形、直线也能用它，只是调对应函数时更直观而已。

2.3 字符串

2.3.1 渲染字符串原理

SDL_ttf 提供了三种方式渲染字符串：Solid、Shaded 和 Blend。三种方式中 Solid 渲染速度最快，但是字体不太美观。Shaded 有点慢，但字体美观，不过是空心字体。Blend 渲染最慢，但字体最为美观。为达到最好显示效果，要选择 Blend 方式，虽然最耗 cpu，但以现在 cpu 处理速度是可接受的，即使是较低性能的移动平台。

SDL 是如何以 Blend 方式渲染字符串，让进行入代码级调试：

1. 在 kingdom 工程中打开 SDL_ttf 工程中的 SDL_ttf.c。
2. 在 TTF_RenderUNICODE_Blended 函数内的“alpha = *src++;”处设断点。
3. 运行时选“Debug”——“Starting Debug”。
4. 调出“Call Stack”窗口。

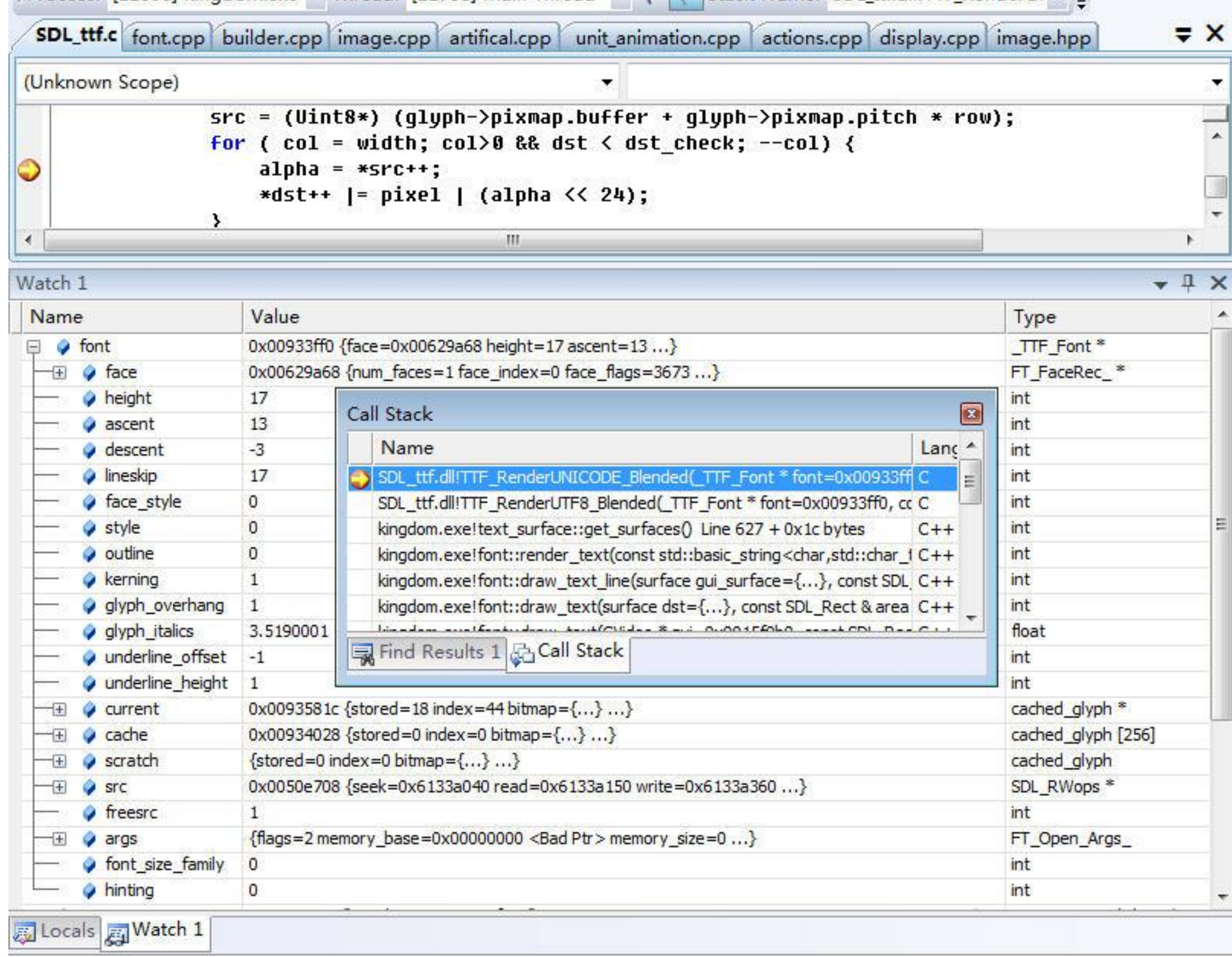


图 2-13 TTF_RenderUNICODE_Blended

此次任务要渲染的是 UTF8 编码字符串，具体渲染过程：

1. kingdom.exe 调用 SDL_ttf.dll 提供的库函数 TTF_RenderUTF8_Blended。
2. TTF_RenderUTF8_Blended 调用 UTF8_to_UNICODE 把 UTF8 字符串转为 UNICODE 编码。然后调用 TTF_RenderUNICODE_Blended 渲染这个 UNICODE 字符串。
3. TTF_RenderUNICODE_Blended 渲染字符串过程：首先调用 SDL_AllocSurface 创建出一个 ARGB 像素格式的空图面；然后一个字符一个字符画在这个空图面上。

TTF_RenderUNICODE_Blended 是如何把字符 ch 画在空图面上的？——它从 font 参数指定的字体文件中取出 ch 对应字模，依据字模信息，一像素接一像素，扫完一行后回到行首继续下一行，就这样一行接着一行在空图面的对应像素上进行着色。字模信息中每像素是介于[0,255]中一个值，这个着色是把这个值直接作为空图画上该像素的 Alpha 分量。

width：每一行都不一样。

小结

1. 渲染字符串结果是形成一个 SDL 图面，字符串“画”在这个图面上。上层程序要记得释放这图面。
2. 图面是 32 位的 ARGB 像素格式。为什么 Blend 方式比 Solid、Shaded 要费时，它采用 32 位而 Solid、Shaded 都采用 8 位像素格式是个重要原因。
3. 为什么此种方式要叫 Blend？它是用字模信息修改 Alpha 分量，Alpha 是个混合时参数，Blend 中文就译作混合。由于它改的是 Alpha 分量，字模信息中各像素取值范围[0, 255]，致

使在一个字符内它画出的像素间也会存在明、暗差异（透明度不同造成）。。

4. 渲染后得到是透明背景图面。
5. 在字符编码上，SDL_ttf 最终要处理成 UNICODE 编码，因而在上层编码选择时它是推荐用 UNICODE。

2.4 渲染音频

2.4.1 音频格式

音频格式三要素：样本格式、通道数、采样率。

- 样本格式：以什么样数值表示一个声音样本。样本格式包括三个参数：有符号还是无符号；一个样本占多少位，理论它可以任意，但使用中只有 8 和 16；占用多个字节时，小端序（LSB）还是大端序（MSB），
- 通道数：声音通道数目，像单声道，立体声（双声道）。各个通道中样本可以是不同格式，但同一通道内样本只一种格式。
- 采样率：单位时间采样数。常用采样率有 8K、16K、32K、44.1K、48K，当中的 K 不是 1024 而是 1000。当存在多通道时，它是同时对各个通道采样，因而把采样率定义为单位时间采样样本数时，要注意这个样本不是指一个通道样本。

基于三要素计算一持读时间为 T 秒的声音占用字节数。

```
声音数据字节数 = 一个样本字节数 X 通道数 X 采样率 X T
```

以上计算可以看出，声音数据字节数一定是“一个样本字节数 X 通道数”整数倍，也就是说，如果出来不是“一个样本字节数 X 通道数”的整数倍数据，则可以肯定这是段不完整声音，要正常播放需去掉多余字节。为此把“一个样本字节数 X 通道数”称为声音粒度（Granularity），又因为一个样本长度不是 8 就是 16 位，以下是根据声音粒度去除多余字节公式。

```
audio_len &= ~(Granularity - 1)。
注：audio_len 是要调整声音数据字节数，同时存储调整后字节数。
```

从音频文件中读出声音到声卡放出声音，要经过两种音频格式：音频文件中格式、渲染格式。

音频文件中格式：它是音频文件中用以记录声音的格式。这里注意区别音频压缩格式，文件中音频格式和音频是采用什么压缩无关，压缩、解压缩不会改变音频格式。

渲染格式：音频数据写到声卡缓冲时的格式。声卡要能正常播放一段声音，它必须知道播放缓冲是什么样格式数据。渲染可以是什么格式，这是由声卡驱动决定的，一般来说这个格式会很灵活，但程序一次运行时一旦选定一种渲染格式，一般就不大可能会改变。

很显然，音频文件中格式和某个音频文件相关，因而不同文件可以有不一样音频格式。但这段声音要能播放出来，这些格式必须统一转化为渲染格式，以着渲染格式写入声卡缓冲。为节省转换时间，需要的音频文件格式和渲染格式越相似越好。

Mix_LoadWAV_RW

Mix_LoadWAV_RW 从一个声音文件读出数据，边读边解码（如果需要），最后把数据转换为渲染格式。

进入代码级调试。

1. 在 kingdom 工程中打开 SDL_mixer 工程中的 mixer.c。
2. 在 Mix_LoadWAV_RW 函数内的“if (SDL_ConvertAudio(&wavecvt) < 0) {”处设断点。
3. 运行时选“Debug”——“Starting Debug”。
4. 调出“Call Stack”窗口

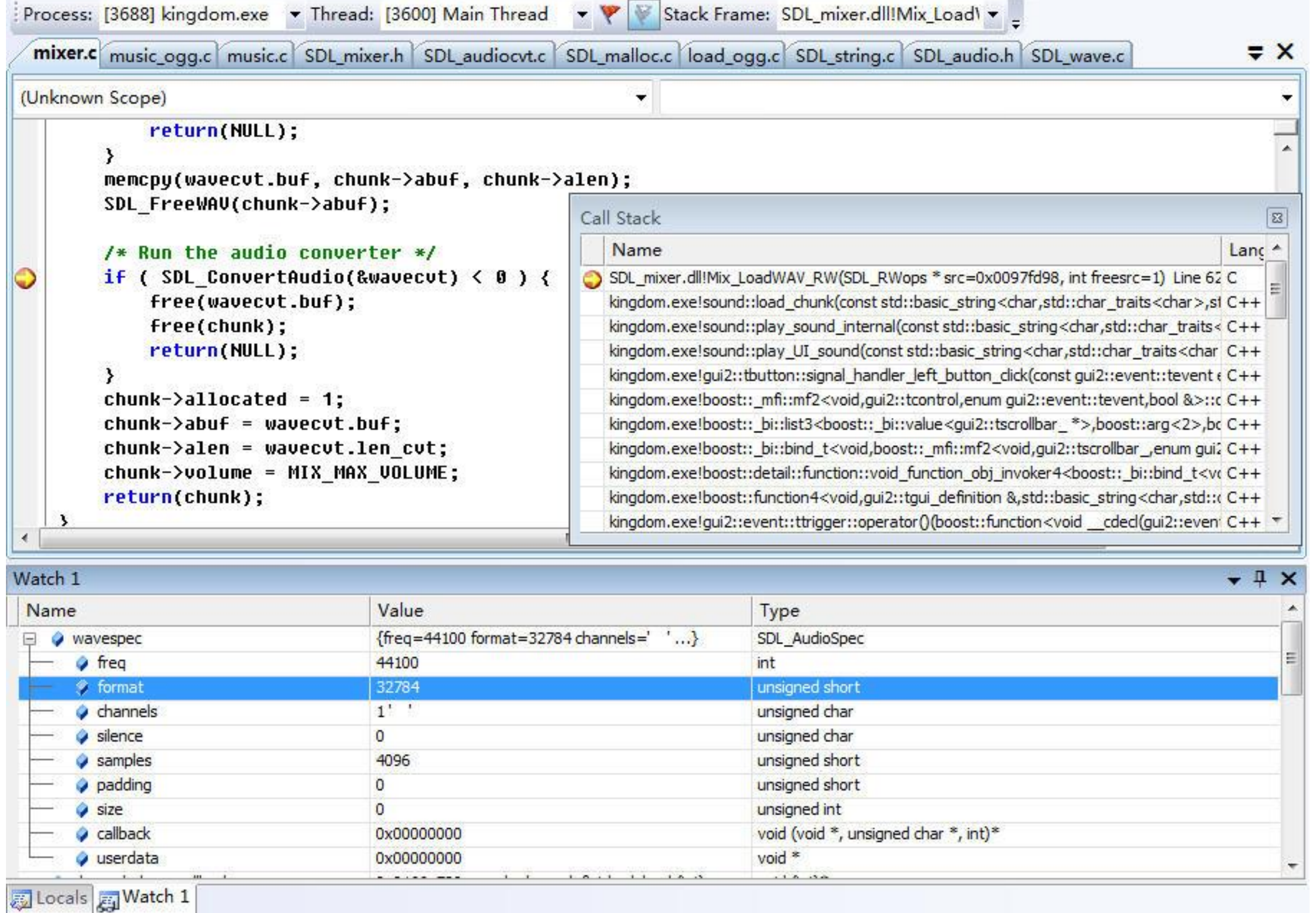


图 2-14 Mix_LoadWAV_RW

wavespec 表示音频文件中格式。图示中格式是：

- format：样本格式。值 0x8010（AUDIO_S16LSB），有符号 16 位数，小端序。
- channels：通道数。值 1，即单声道。
- freq：采样率。值 44100，它是 44.1K 采样率。

再看此时的 mixer 变量，它表示渲染格式。

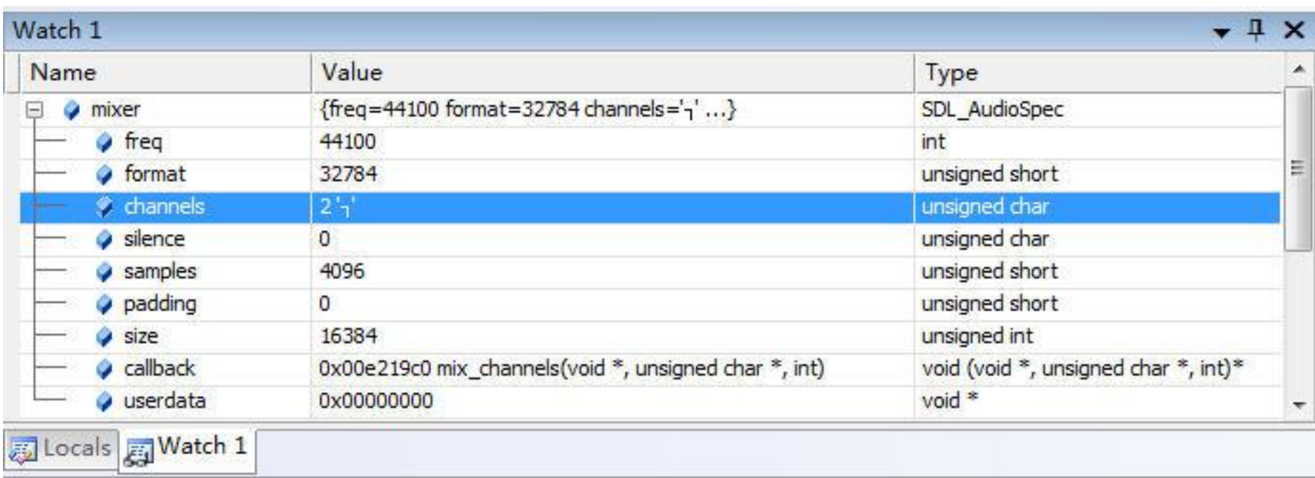


图 2-15 mixer 变量

- format：样本格式。值 0x8010（AUDIO_S16LSB），有符号 16 位数，小端序。
- channels：通道数。值 2，即双声道。
- freq：采样率。值 44100，它是 44.1K 采样率。

结合 wavespec 和 mixer，它们有一样的样本格式、采样率，但不一样声道数，要能播放这段声音，需要把单声道转换为渲染格式要求的双声道。

SDL_BuildAudioCVT、SDL_ConvertAudio 执行这个转化。转换分两步：SDL_BuildAudioCVT 构造转换函数表，SDL_ConvertAudio 调用转换表中函数进行转换。

让进入 SDL_ConvertAudio，把断点设在 cvt->filter_index = 0，执行到断点看 cvt 变量。

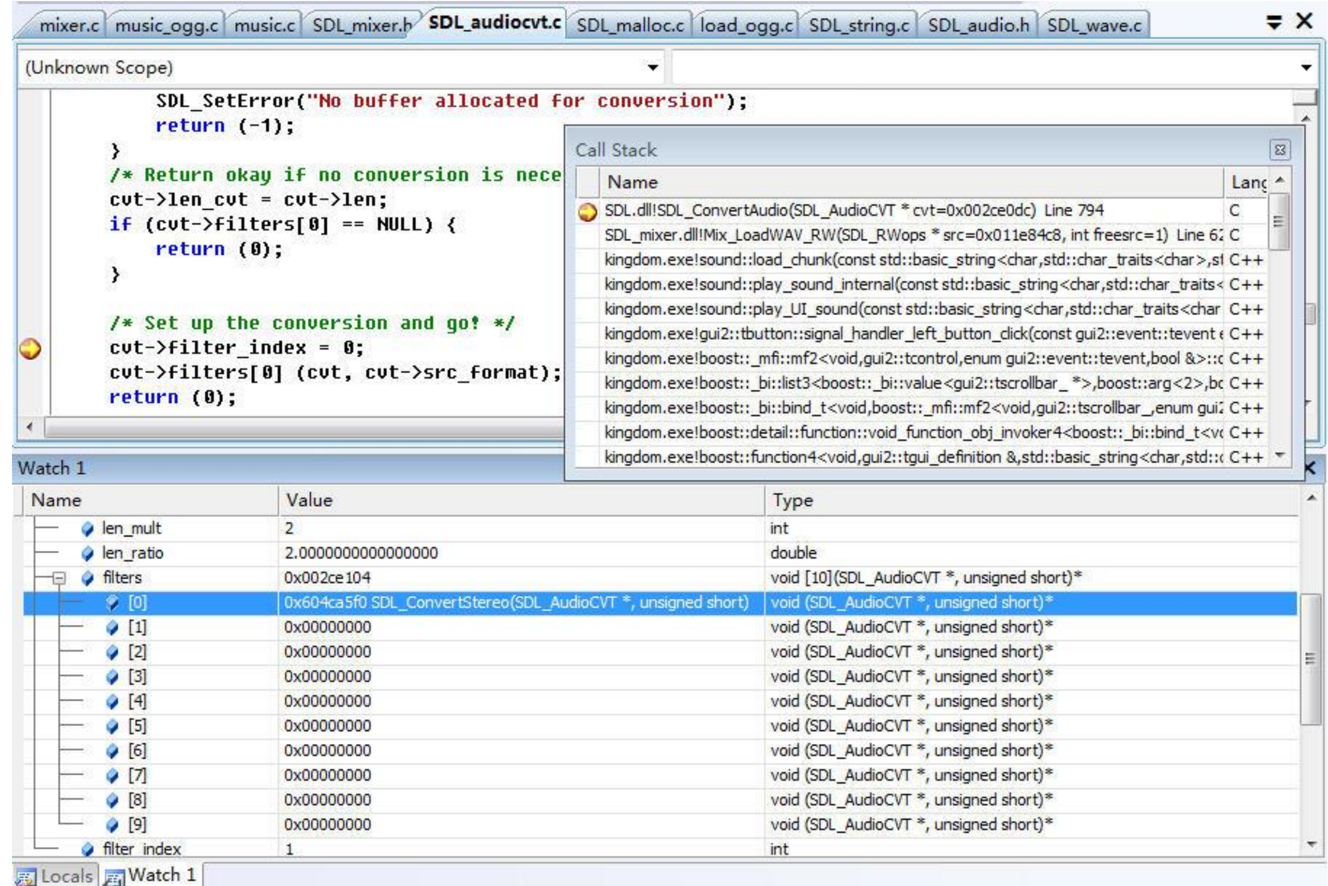


图 2-16 SDL_ConvertAudio

展开 filters，它是一个函数指针数组，函数表中存在的 SDL_ConvertStereo 执行的就是把单声道转为双声道。此次转换只要这么一个函数就够了。

filter_index 表示转换函数个数，但此处要执行时立即把它置 0，那执行转换函数时怎么知道哪个是最后一个转换函数？

让进入 SDL_ConverStereo，在它的函数尾会看到：

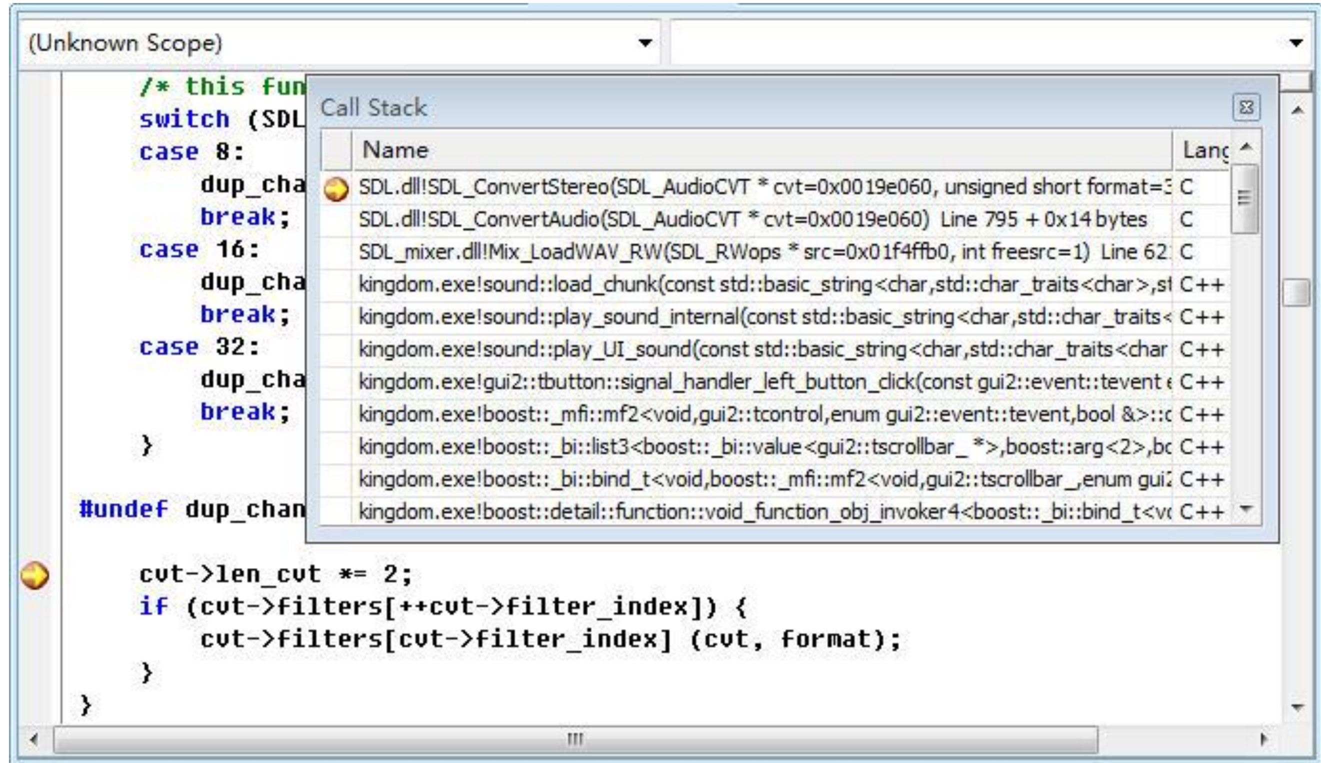


图 2-17 SDL_ConverStereo

其它转换函数都有这样代码，它判断表中下个函数地址是否为 NULL，如果 NULL 就是最后一个转换函数。因而虽然表中 10 个入口，但最多可存在 9 个转换函数。

存储声音数据的内存块

以文件中格式表示的声音数据

SDL_LoadWAV_RW/Mix_LoadOGG_RW 边读声音文件边解码（如果需要），解码后数据放在 chunk->abuf。

- chunk->abuf：音频数据缓存，它用 SDL_malloc 在 SDL_LoadWAV_RW/Mix_LoadOGG_RW 分配，上层记得要用 SDL_free 进行释放。它是以音频文件中格式表示未压缩过（如果文件是压缩过则已经解压）音频数据。
- chunk->alen：chunk->abuf 的有效字节数，也就是音频数据字节数，它除以粒度，再除以采样率就是声音能持续播放时间。

以渲染格式表示的声音数据

文件中格式表示的声音数据进入 SDL_ConvertAudio，这函数输出是以渲染格式表示音频数据，它把生成数据放在 wavecvt.buf，wavecvt.len_cvt 指示 wavecvt.buf 有效字节数。

从分析两种格式音频数据可以看出，Mix_LoadWAV_RW 执行完后所有数据已经解码，并转化为渲染格式。到此看下它的返回值，即 Mix_Chunk 各字段意义。

- allocated：1。
- abuf：已经支持渲染格式的未压缩音频数据。
- alen：abuf 数据长度。
- volume：MIX_MAX_VOLUME（128）。

Mix_LoadWAV_RW 缺陷

Mix_LoadWAV_RW 执行读文件、解码、转为渲染格式，一个函数就干了这么多事，但这种做法存在很大问题，因为它“全”都做了，一旦声音数据量大，界面反应慢是小事，更严重的

是要存放整块解码后数据，系统就没法提供如此大内存！让考虑下格式是 16 位、双声道、44.1K，持续时间是 4 分钟的一段声音，它要占用近 41M 内存（4*44100*240），4 分钟如此，更别说比它还大的文件。Mix_LoadWAV_RW 适合处理小段声音，也就是后面要说的音效，对长时间声音不能用这函数，也就是后面说的音乐。音乐虽不用这函数，但处理要经过阶段和音效差不多，都要经过读文件、解码、转换为渲染格式，转换时用的也是 SDL_BuildAudioCVT、SDL_ConvertAudio，只不过它是一段一段地读、解码、转换、并播放。

2.4.2 播放声音

这里的播放声音特指把符合渲染格式音频数据放入声卡缓冲，然后声卡播放声音这个过程。让进入调试状态

1. 在 kingdom 工程中打开 SDL_mixer 工程中的 mixer.c。
2. 在 mix_channels 函数内设断点。
3. 运行时选“Debug”——“Starting Debug”。
4. 调出“Call Stack”窗口，设置到 SDL_RunAudio。

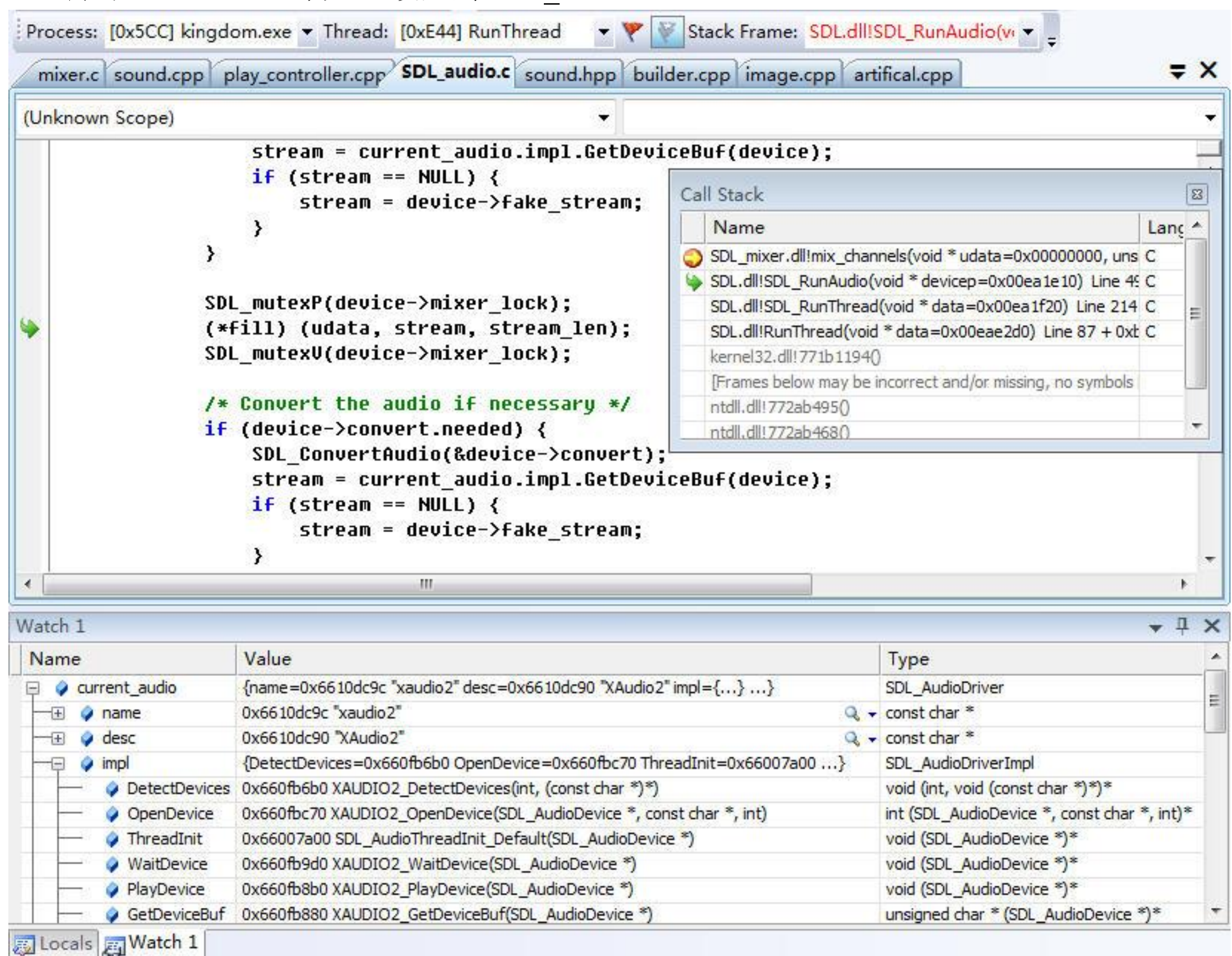


图 2-18 SDL_RunAudio

SDL_RunAudio 负责播放声音，它是 SDL.dll 中函数。

Watch 窗口中显示 current_audio 这个 SDL_AudioDriver 实时信息，SDL_AudioDriver 是对声卡驱动抽像，它实现播放声音是通过 impl 这个 SDL_AudioDriverImpl 对象。impl 是个函数表，它提供了播放声音需要的一系列操作，像 GetDeviceBuf 负责把声卡缓冲地址返回给上层，WaitDevice 阻塞住上层线程，直到声卡有有效缓冲。

impl 对象是对声卡操作的抽像，正是它在细节上实现了 SDL 的跨平台渲染音频。示例中的

“xaudio2”是使用 DirectX 实现渲染音频，MacOS X/iOS 则使用“coreaudio”实现渲染音频。

通过查看 SDL_RunAudio，可以小结下播放声音步骤，更直观说是如何调用 impl 中相关函数。

1. impl.GetDeviceBuf 获得声卡缓冲地址。缓冲长度是个固定值，程序初始时设定，通常设为 16384。
2. “(*fill)(udata, stream, stream_len)”向缓冲填入要被播放的音频数据。此处 fill 对应的是 mix_channels。
3. impl.PlayDevice 通知声卡驱动，缓冲区数据有效了，可以播放。声卡播放声音。
4. 播放速度是要按采样率来的，有持续时间，那上层程序如何知道同步这个持续时间？它是调用 impl.WaitDevice，这函数会阻塞调用线程，直到又一个声卡缓冲区有效。

fill 一次填充的只是 16384 字节，声卡驱动不可能只有 16384 字节。以小字节填充一是为满足播放连贯性，A 段在播放时 B 段在填充，A 段一播放完 B 段已有效；二是为实现其它功能，像混合。因为这个小字节，impl.WaitDevice 播放的一般不是刚填入的那段声音。

SDL_AudioDriver 它是在“RunThread”这个线程上下文执行的，区别于像 Mix_LoadWAV_RW 的“Main Thread”线程。一旦开启 SDL 支持音频，它就会创建一个 RunThread 线程，它负责混合、播放声音。

以上播放步骤小结是 device->convert.needed=0，注意下 device->convert.needed=1 时，它指示 fill 填入的音频数据不是渲染格式，要渲染须要转换。因而它 fill 进的不是 impl.GetDeviceBuf 得到的缓存，而是 device->convert.buf 这个内存地址。在把数据写入声卡缓存时要调用 SDL_ConvertAudio 把格式转换为渲染格式，再调用 impl.GetDeviceBuf 得到声卡缓存，把转换后数据写入缓存。

以上大概说了播放声音过程，接下让考虑一个问题：混合声音。开着电脑，一边玩游戏听背景音乐，一边开 Winamp 听 mp3，这就是个混合声音例子。两个应用程序在产生声音，但听去它们被“无缝”接合了，好像声音是“并行”播放。

对声卡来说，它任意一时刻只能播放一种声音，或播放游戏背景音乐或播放 mp3。我们听去“无缝”接合，那是声卡对声音进行“时分复用”，把多种声音“时分复用”到一个缓冲区上，由于每次播放一小段声音，像 16384 字节，人耳分辨不出如此细粒度声音，于是认为声音是“并行”播放。

同时开游戏背景音乐和 mp3 是复合两个应用程序声音，混合也发生在一个应用程序中。像开着背景音乐，然后按下一个按钮，为有好的用户界面，按下按钮有会出声音提示，这就是背景音乐和用户界面音效混和例子。或开着背景音乐，游戏中出现魔法攻击，魔法攻击时会放出魔法飞的声音，这是背景音乐和攻击音效混和例子。

回头看播放过程，它在向缓冲区填数据时用“(*fill)(udata, stream, stream_len)”，此处让考虑个想法，如果 fill 实现的不仅仅是把一种声音填入缓冲，而是把数种声音按策略进行混合，然后再填入缓冲，会不会就实现混合声音功能？——实际正是如此，fill 这个函数指针一般情况下就是指向 mix_channels 这个以着通道方式将声音进行混合的函数。

2.4.3 混合声音

先说下 SDL_mixer 引入的两个术语：音乐和音效。

音乐（music）：表现在有较长的持续时间。常见例子是歌曲，程序的背景音乐。

音效：表现在较短的持续时间，可能就是毫秒级。常见例子像按下按钮时提示音，特定事件发生的提示音，像新邮件到了，哪个好友上线了。

SDL_mixer 提供的一大功能是混和声音，它混合声音的基础是把声音分成音乐和音效，至于具体到特定程序中哪段声音属音乐、哪段声音属音效，这是由程序员决定的。

为理解 SDL_mixer 如何混合声音，让进入调试状态。

1. 在 kingdom 工程中打开 SDL_mixer 工程中的 mixer.c。
2. 在 mix_channels 函数的“while (mix_channel[i].playing > 0 && index < len) {”上设断点。
3. 运行时选“Debug”——“Starting Debug”。
4. 等待程序直到进入主菜单界面，按下主菜上一个按钮，这时会要求声卡播放按钮按下提示音，导致触发第 2 步设置的断点。
5. 代码窗口定位到 mix_channels 内以下窗口显示的位置。

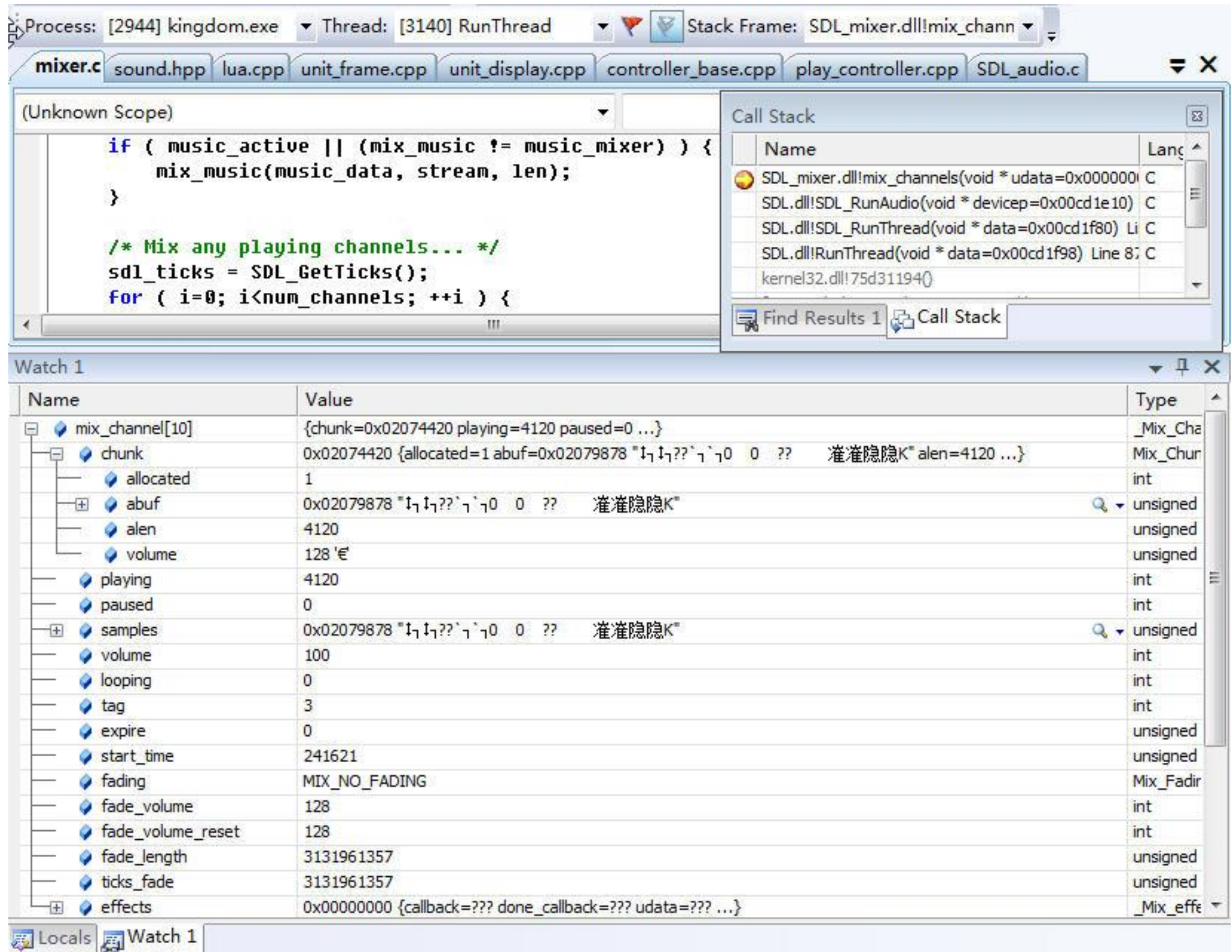


图 2-19 mix_channels

结合 2.4.2 播放声音，可以看到 stream 就是声卡缓存，len 是缓存可用字节数。

由图示中代码可以发现，mix_channels 混音策略是先处理音乐（mix_music）、然后处理音效（mix_channel）。

知道这个基本顺序，接下让问两个问题？

1、程序设定背景音乐是不断重复，是否意味着 mix_music 总会填满声卡缓存？如果填满，此次音效数据如何放入缓存？

程序设定背景音乐是不断重复，的确会造成 mix_music 总会填满声卡缓存。但是，虽然音乐填满了缓存，可如果查到此次有音效，音效还是会播放，因为音效数据填充方式是从缓存地址 0 处开始填！

这种覆盖式填法导致的实际情况是，虽然处理音乐早于音效，但音效播放优先级反高于音乐！

通过看整个 mix_channels，会发现它向缓存填了三次数据。

1. memset(stream, mixer.silence, len)。从地址 0 开始，整缓存置为静音。
2. mix_music。从地址 0 开始，填充音乐。

3. mix_channel。从地址 0 开始，填充音效。

2、为什么音效要分散到数个通道？

原因主要出在音效多样性。举个例子，按下按钮同时来了新邮件，这时要出两个音效，作为处理原则，这两个音效数据不能交叉。当然，应用程序自可处理不交叉，只是 SDL_mixer 提供了一种辅助手段，上层只要使用两个通道，把按钮声效放在 A 通道，新邮件声效放在 B 通道，SDL_mixer 就会保证了不交叉，并且引入优先级，通道号小的那种声音会被优先播放。

最后让说下_Mix_Channel 中几个字段语义。

- chunk：音效是通过 Mix_LoadWAV_RW 从文件“一步”生成，它返回的 Mix_Chunk 就被直接赋给这个 chunk 指针。
- samples：要开始播放的音频数据地址。初始值是 chunk->abuf，伴随着播放会不断向后移。
- playing：还需要播放的音频字节数。初始值是 chunk->alen。
- tag：通道标识。这个是应用程序主观设置的值，用于区别各种通道。
- looping：循环播放次数。0 指示不循环播放。

2.4.4 播放 API

SDL.dll 和 SDL_mixer.dll 联合实现播放声音，一旦要使能播放，SDL.dll 会新建专门播放声音的线程，该线程的函数是 SDL_RunAudio。理论上说，没有 SDL_Mixer 也能播放声音，但这样的话编程声音太累了，于是出来 SDL_mixer，它在播放声音的基础上追加混音功能。SDL_mixer 把声音分成音效、音乐，可以“同时”播放数首音效，但在任一时刻只能播放一首音乐。通道是个和音效相关的概念，和音乐无关。音效播放优先级要高于音乐。

通道可分为两段，前一段是保留通道，后一段非保留通道。这两类通道区别是以“channel=-1”调用 Mix_PlayChannelTimed、Mix_FadeInChannelTimed 时，只会选择非保留通道。

以下叙述时会碰到“逐渐增强”（MIX_FADING_IN），它是指播放时让音量缓慢增加，相应的“逐渐减弱”（MIX_FADING_OUT）是指播放时让音量缓慢变小。

对应用来说，以下 API 或许是太低级了，Rose 基于它们写了更直接面像应用的声音接口，参考“**3.5 声音和图像**”。

初始化

```
int Mix_OpenAudio(int frequency, Uint16 format, int nchannels, int chunksize);
```

初始化 SDL_Mixer，执行逻辑包括预分配音效通道，加载音频压缩解码模块，设置和播放音乐相关的全局变量到“0”状态。参数指示渲染格式，程序要播放声音必须先调用这函数。

音效

```
Mix_Chunk *Mix_LoadWAV_RW(SDL_RWops *src, int freesrc);
#define Mix_LoadWAV(file)  Mix_LoadWAV_RW(SDL_RWFromFile(file, "rb"), 1)
```

Mix_LoadWAV_RW 从一个声音文件读出数据，边读边解码（如果需要），最后把数据转换为渲染格式，形成的结果数据放入 Mix_Chunk 这个内存块。由于 Mix_Chunk 已存储全部结果，程序接下可以不须要 SDL_RWFromFile 创建的 src，于是在调用 Mix_LoadWAV_RW 时把 freesrc 设为“1”，既然如此，应用为获取 Mix_Chunk 一般直接使用 Mix_LoadWAV 宏。

```
int Mix_PlayChannelTimed(int which, Mix_Chunk *chunk, int loops, int ticks);
#define Mix_PlayChannel(channel,chunk,loops)
Mix_PlayChannelTimed(channel,chunk,loops,-1)
```

把一个音效放入指定的通道，并进行播放。chuck 是表示音效的 Mix_Chunk，which 是要放入的通道，-1 表示由系统自动选择第一条正空闲的通道（正空闲是_Mix_Channel.playing<=0 的非保留通道）。loops 指示循环播放次数。ticks 指示最长播放多少时间，<=0 表示不限停止时刻，否则在 ticks 毫秒后会强行停止（如果那时这音效还没播放完）。

```
int Mix_FadeInChannelTimed(int which, Mix_Chunk *chunk, int loops, int ms, int
ticks);
```

```
#define Mix_FadeInChannel(channel,chunk,loops,ms)
Mix_FadeInChannelTimed(channel,chunk,loops,ms,-1)
```

Mix_FadeInChannelTimed 功能也是把把一个音效放入指定的通道，并进行播放，但相比 Mix_PlayChannelTimed，它让在开始 ms 毫秒内出现音量逐渐增大效果。这两函数参数的语义一样，Mix_FadeInChannel 则是不限制播放时长的版本。

```
int Mix_ExpireChannel(int which, int ticks);
```

设置 which 通道的播放时长到 ticks 毫秒。whick=-1 表示设置所有通道。ticks 指示最长播放多少时间，<=0 表示不限停止时刻，否则在 ticks 毫秒后会强行停止。

```
int Mix_HaltChannel(int which);
```

停止播放 which 通道的音效。whick=-1 表示停止所有通道。

音乐

```
Mix_Music *Mix_LoadMUS(const char *file);
```

该函数读取声音文件，形成一个可解析该音乐的 Mix_Music 结构，程序接下就用这 Mix_Music 去播放此音乐。它解析声音是什么压缩只是根据扩展名，像“wav”就认为是 WAVE 格式，“ogg”就是 OGG 格式。此函数不读声音数据，基本不耗内存。

```
int Mix_FadeInMusic(Mix_Music* music, int loops, int ms);
```

以逐渐增强方式播放指定的音乐。music 是之前 Mix_LoadMUS 返回的结构。loops 指示循环播放次数，-1 表示不限制次数，0 和 1 一样表示只播放一次。ms 是指示一直以逐渐增强方式播放的持续时间，它只在第一轮播放时有效，在第二轮或之后是维持正常音量。为什么要用逐渐增强？试想下你希望实现在几首音乐间切换，要切换到一下首时，为不让显得突变，于是让切换到播放它时音量是缓慢变高，即从零到正常音量，在此种场合，第二次或之后轮因为一直播放是同一首音乐，也就不需要这效果。

看到 Mix_FadeInMusic 中的逐渐增强，有人会想到是否有 Mix_FadeOutMusic。SDL_Mixer 还真有提供这函数，但作用不是以逐渐减弱方式播放指定音乐，而是在 ms 时间内以逐渐减弱方式播放当前音乐。注意是当前音乐，它不能播放新音乐。

```
int Mix_FadeOutMusic(int ms);
```

ms 指示要以逐渐减弱方式播放的持续时间，当 ms<=0 时，它的功能是立即停止播放当前音乐，等同 Mix_HaltMusic。

```
int Mix_HaltMusic(void);
```

作用是停止播放当前音乐，由于 SDL_Mixer 在任一时刻只能播放一首音乐，这意味是停止播放音乐。

```
int Mix_PlayingMusic(void);
```

它用于判断当前是否正在播放音乐。虽然它的返回值是 int，但基本可用 0 和非 0 进行判断。

2.5 键盘相关

键盘包括物理键盘和软件盘，SDL 中和键盘相关的主要是三种事件，SDL_KEYDOWN、SDL_KEYUP 和 SDL_TEXTINPUT。

2.5.1 SDL_KEYDOWN/SDL_KEYUP

收到一个键盘按键时，SDL 内部处理这按键要依次经过三种码：os 特定码、扫描码和 SDLK_xxx 码。

	按键：回车	按键：Backspace	按键：数字 2
os 特定码 (android)	66(0x42)	67(0x43)	9(0x9)
os 特定码 (windows)	28(0x1c)	14(0xe)	3(0x3)
扫描码	SDL_SCANCODE_RETURN	SDL_SCANCODE_BACKSPACE	SDL_SCANCODE_2(31)

SDLK_xxx 码	'\r'	'\b'	'2'

os 特定码。根据操作系统提供的键盘 api 直接得到的码值，对某个键盘按键，各操作系统极可能不一样，像数字 2，android 收到的值是 0x9，windows 是 0x3。

扫描码（全 os 通用）。os 特定码转到扫描码方法是以 os 特定码为索引的查表法，各操作系统用的表不一样。windows 查的是 windows_scancode_table，Android 查的是 Android_Keycodes。值 0 的扫描码 SDL_SCANCODE_UNKNOWN 表示该 os 特定码没对应的扫描码。

SDLK_xxx 码（全 os 通用）。app 常在处理的码值，属于 ASCII 部分的值就是 ASCII。扫描码转到 ASCII 方法也是查表法，查表索引是扫描码，这个表是所有操作系统共用，SDL_default_keymap。

SDL_KEYDOWN/SDL_KEYUP 是 SDL 向 app 发的和键盘相关两种事件，事件上下文有个叫 SDL_Keysym 结构。

```
typedef struct SDL_Keysym {
    SDL_Scancode scancode;
    SDL_Keycode sym;
    Uint16 mod;
    Uint32 unused;
};
```

scancode 对应该键扫描码，sym 对应该键 SDLK_xxx 码。unnsed 正如名称写的，未使用。mod 是按下该键时还有的修饰键，同时按下多个修饰键时，mod 是这些 mod 值的“|”。

常用名称	mod 值	SDLK_xxx 码
数字控制键	KMOD_NUM	SDLK_NUMLOCKCLEAR
CAPS Lock	KMOD_CAPS	SDLK_CAPSLOCK
左侧 CTRL	KMOD_LCTRL	SDLK_CAPSLOCK
右侧 CTRL	KMOD_RCTRL	SDLK_RCTRL
左侧 SHIFT	KMOD_LSHIFT	SDLK_LSHIFT
右侧 SHIFT	KMOD_RSHIFT	SDLK_RSHIFT
左侧 ALT	KMOD_LALT	SDLK_LALT
右侧 ALT	KMOD_RALT	SDLK_RALT
左侧 GUI	KMOD_LGUI	SDLK__RGUI
右侧 GUI	KMOD_RGUI	SDLK__RGUI
MODE	KMOD_MODE	SDLK_MODE

2.5.2 SDL_TEXTINPUT

SDL_TEXTINPUT 上下文中存储字符的格式是 UTF-8。

会发出 SDL_TEXTINPUT 的字符来自两处，一是 IME 输入，二是 KEY 输入的>=0x20（空格）且不是 0x7f 的 ASCII。IME 输入，只会出现在 SDL_TEXTINPUT，>=0x20 且不是 0x7f 的 ASCII 既会出现在 SDL_TEXTINPUT（须核验 iOS？），也可能出现在 SDL_KEYDOWN/SDL_KEYUP，其它的 ASCII 则只能靠 SDL_KEYDOWN/SDL_KEYUP。

为什么收到 KEY 也能发到 SDL_TEXTINPUT？以 Android 为例，当以 KEY 方法收到的是可打印字符和空格，会同时发向 SDL_TEXTINPUT。

```
public static boolean isTextInputEvent(KeyEvent event) {
   return event.isPrintingKey() || event.getKeyCode() ==
KeyEvent.KEYCODE_SPACE;
}
```

对>=0x20 且不是 0x7f 的 ASCII，目前 PC 平台可保证出现在 SDL_KEYDOWN/SDL_KEYUP，移动平台则无法保证，为此要获得它们安全的方法是使用 SDL_TEXTINPUT。基于这些，建议 app 处理键盘采用下面规则。

SDL_TEXTINPUT 处理所有可打印字符，包括 IME 输入，>=0x20 且不是 0x7f 的 ASCII。

SDL_KEYDOWN 处理不可打印符，像回车、Backspace、Delete 等。

SDL_KEYDOWN 处理快捷键。一来快捷键含有修饰符，像 CTRL，这只能由 SDL_KEYDOWN 收到，二来快捷键只用在 PC 平台，PC 平台是能收到>=0x20 且不是 0x7f 的 ASCII，这样就能正常处理像 CTRL+c 等。

SDL 按键包括 SDL_KEYDOWN、SDL_KEYUP，上面只处理 SDL_KEYUP，因为有了 SDL_KEYDOWN 后，SDL_KEYUP 一般总会收到，于是可把 KEY 处理统一放在 SDL_KEYDOWN 这个入口。

2.6 处理事件

事件包括键盘输入、鼠标输入、缩放窗口、退出程序等，不同操作系统以不同机制触发事件，SDL 作为实现跨平台编程的基础库，它需要向上层提供统一的处理事件抽像。

要如何处理事件？SDL 是个库，它需要负责收集事件，但针对具体事件如何处理，那是上层 app 的事，不过这同时要求 SDL 还有个任务是要能提供 API，让上层从它那里读取出正发生的事件。由此可看出 SDL 需实现收集、读取功能，那这两个功能是如何融入整个系统，SDL 实现了以下方案。

SDL 收集事件。收集是把捕捉到的事件表示为统一格式（SDL_Event）并放入事件队列。应用程序“有空”时就查询事件队列，依次读取出各事件，针对具体事件实施具体处理。“有空”指什么？一种是应用程序专门有个事件线程，线程不断从 SDL 读取事件；另一种是应用程序不专门开线程，只是作为主线程中一步骤，那里去读取事件。SDL 建议是采用第二种。

接下让以鼠标弹开事件为例，分析 SDL 如何处理事件。

2.6.1 收集

SDL_PumpEvents 是 SDL 提供的收集 API。收集过程不仅要实现捕捉事件，还要把事件放入事件队列。让进入源码级调试。

1. 在<app>工程打开 SDL 的 SDL_mouse.c，SDL_PrivateSendMouseButton 函数内设断点。
2. 运行时选“Debug”——“Starting Debug”。加载界面时不要按任何鼠标键。
3. 进入标题标幕后，按下鼠标左键，会触发断点。

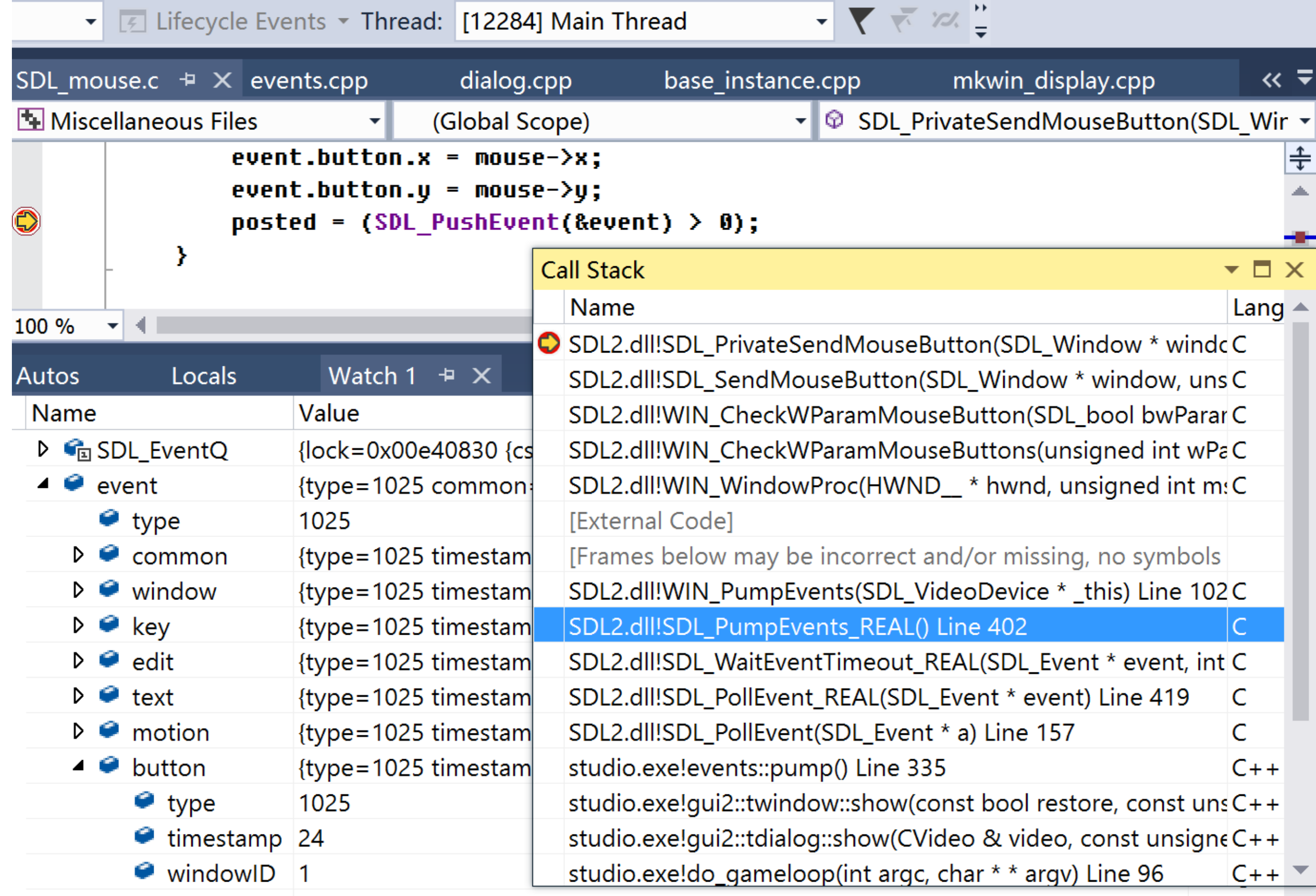


图 2-20 SDL_SendMouseButton

图 2-20 显示 SDL 收集鼠标事件，并把事件放入事件队列（SDL_EventQ）。通过 Call Stack，可描述出 SDL 是如何收集此个事件。

- 在主线程（Main Thread），app 调用自个实现函数 events::pump，后者调用 SDL 提供的 SDL_PoolEvent，它会调用 SDL_PumpEvents。
- SDL_PumpEvents 是个跨平台收集事件函数，它要实现收集需调用各操作系下“真正”的收集函数，Windows 系统就是 WIN_PumpEvents。
- 在“2.1.1 创建窗口”有提到 WIN_PumpEvents，这里重复抄下该函数实现。

```
void WIN_PumpEvents(_THIS)
{
    MSG msg;
    while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

它从系统收集消息时调用 PeekMessage，而不是阻塞式的 GetMessage，这保证了要是没事件时主线程可以立即继续做其它事。

- 捕捉到鼠标弹开事件，DispatchMessage 接下把这事件发去窗口过程，即 WIN_WindowProc，后者以个大 case 处理各个事件，鼠标弹开事件的消息码是 WM_LBUTTONDOWN。

```
LRESULT CALLBACK WIN_WindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam)
{
    switch (msg) {
    ......
    case WM_LBUTTONDOWN:
        SDL_SendMouseButton(data->window, SDL_PRESSED, SDL_BUTTON_LEFT);
        break;
    ......
}
```

- SDL_SendMouseButton 根据事件状态值（state、button、x、y 等）形成以 SDL_Event 结构表示的一个事件，并把这事件放入 SDL 事件队列 SDL_EventQ。至此完成了捕捉事件、把事件放入事件队列。

结合 Watch 中的 event 变量，让说下 SDL 如何向上层提供统一的事件表示。各个操作系统以自个格式表示事件，SDL 要跨平台实现处理事件则必须以一种统一的格式表示事件，这个统一格式就是 SDL_Event。但事件种类繁多，光以“一种”struct 是不可能表示它们的，为此 SDL_Event 采用 C/C++语法中的 union。

```
typedef union SDL_Event
{
    Uint32 type;                    /**< Event type, shared with all events */
    SDL_WindowEvent window;          /**< Window event data */
    SDL_KeyboardEvent key;           /**< Keyboard event data */
    SDL_TextEditingEvent edit;       /**< Text editing event data */
    SDL_TextInputEvent text;         /**< Text input event data */
    SDL_MouseMotionEvent motion;     /**< Mouse motion event data */
    SDL_MouseButtonEvent button;     /**< Mouse button event data */
    SDL_MouseWheelEvent wheel;       /**< Mouse wheel event data */
    ......
} SDL_Event;
```

SDL_Event 表示事件有以下几个特点。

- 一个 SDL_Event 实例只能表示一个事件。
- 采用 union，使不同事件占用同一块内存。
- 各事件按相似程度归类。WM_LBUTTONDOWN、WM_LBUTTONUP 就归为 SDL_MouseButtonEvent。
- 各事件的特定结构中第一个字段必须是 Unit32 type。

2.6.2 读取

SDL_PeekEvents 是 SDL 提供的读取 API。收集过程已把事件放入 SDL 事件队列 SDL_EventQ，以着一般逻辑去猜测，在图 2-20 中，app 实现的自个函数 events::pump 调用完收集函数 SDL_PumpEvents 后，接下应该是调用读取函数。事实的确如此， app 调用的 SDL_PoolEvent 其实由两部分组成，一是提供收集功能的 SDL_PumpEvents，二是用于读取事件的 SDL_PeekEvents。

```
int SDL_PeepEvents(SDL_Event* events, int numevents, SDL_eventaction action,
Uint32 minType, Uint32 macType);
```

SDL_PeekEvents 从事件队列 SDL_EventQ 中取出顶头事件。参数 action 决定了是否要把事件同时从队列移除，SDL_PoolEvent 用了会导致移除的 SDL_GETEVENT。

补充下 2.5.1 在进入调试时为什么要写“加载界面时不要按任何鼠标键”（为更直观了解接下要描述内容请重新 Debug”——“Starting Debug”，并在加载界面时按下鼠标键以让触发第一步设置的断点）。加载界面时收集事件、读取事件是通过 SDL_PoolEvent。SDL_PoolEvent 以一个函数就实现了收集、读取，但在编写自个代码过程中往往并不是采用这种方式，而是把收集、读取分开处理，即进入标题屏幕后的那种处理方式。

第三章 框架逻辑

本章目标

- 知道 Rose 如何定义应用框架。
- 理解 events::pump。
- 使用网络框架。

3.1 概述

Rose 是个直接面向 app 的 SDK。它基于 SDL 构建，以提供跨平台这个目标来说，有 SDL 这个 SDK 就够了，既然要拥有存在的价值，Rose 必须提供 app“最直接”需要的接口。“最直接”的接口具体包括哪些 API？这是没有答案的，它要跟着时代对 app 的要求在变。在字符时代，一个画矩形函数可能已属于“最直接”接口，但到桌面时代，它不够直接了，app 希望 SDK 直接提供包含画矩形功能的窗口控件操作。在 PC 占统治地位时代，接口可以不包括触摸，到移动时代支持触模就成了“最直接”中一员。在单机时代，网络功能少，但到了希望随时随地可沟通的现在，SDK 就到了应该直接内置聊天。

虽然 app 五花八门，但 SDK 提供的接口不是需要广而全，而是突出特色。不要希望 SDK 实现“所有”接口，越去追求这个目标反而会出来相反结果。Rose 设计接口时要明白自个定位，它是“最后”面向 app，如果把众多 SDK 比作产品线，它属于产成品。作为产成品，一个任务是要包容众多半成品 SDK，基于它们编写接口，甚至直接暴露它们成为接口一部分。“不要重复发明轮子”，开源社区经过数十年发展已形成不少“标准”SDK，像处理图像的 OpenCV，深度神经网络框架 TensorFlow，网络音/视频 Webrtc。除去开源，还有 C/C++标准或半标准库，这就不得不说 Boost。个人对 Boost 大量使用泛型持保留意见，很多时候代码易读比简化更重要，但 Boost 怎么说都有着去用它的价值，像把类成员函数作为回调函数、变参回调、gzip 压缩。

作为最终面向 app 的 SDK，它要清晰指出编程方向，即程序员在基于它写 app 时，时刻都明白为什么写这部分代码，接下该写哪部分代码。这里说下 MFC，我认为它是失败的，很大一个原因是没有向程序员提供一条驱动 app 发展的主线，使得那些个类平行向前发展，没有交集。SDK 应该使提供的各模块绞合在一起，它们朝同一方向延伸，延伸去的方向清晰指出了 app 的编程方向。为实现它，框架逻辑就应运而生了，它自然而然成了 Rose“最直接”接口中一员。以实现需要的代码量看，框架逻辑是排在窗口子系统后的第二大模块，但从指导编程方向上，它无疑是最重要模块。

对框架，在编程上是 Rose 定义了框架主线，主线是由一些类、函数组成，app 则或提供这些逻辑需要的回调函数，或从特定类派生、然后通过重载虚函数来定制出自个的框架。

3.2 app 框架

深入这问题前让先把窗口分为两类，对话框和场景，区分它们的依据是当中是否存在大地图。大地图不是指现实世界中地图，它是泛指，指那些拥有以下特点的矩形区。

- 在此窗口中，不论尺寸或功能都拥有无可争辨地位。由于地位重要，把整窗口放大时往往是只放大它，其它部分依旧保持原尺寸。
- 在此窗口时，用户往往集中在此矩形进行操作。

依据此区分，安装向导出现的窗口是对话框；运行记事本这个 app 时，编辑窗口是场景，弹出的“查找”、“设置字体”是对话框。

Rose 使用的一条规则：任何窗口或是对话框或是场景。接下让继续分析框架，它控制着整个 app 怎么运行。作为一个面向应用的 SDK，它制定出的框架至少要满足绝大多数 app 的需要，Rose 提供的到底是什么个框架，让看几个 app 的生命期。

记事本（notepad.exe）。启动过渡——编辑场景——退出过渡。记事本是个小程序，启动过渡阶段时间极短。进入编辑场景后向用户展现编辑窗口，用户在此窗口按需操作，像输入字符，弹出“查找”对话框、弹出“设置字体”对话框。一旦关闭编辑窗口，程序进入退出过渡阶段。

Outlook。启动过渡——邮箱场景——退出过渡。虽然和记事本一样分三个阶段，不过比记事本要复杂得多。启动过渡时间变长，为增强用户体验需在这阶段显示加载进度。进入邮箱窗口，按下“新建邮件”还会建立编写新邮件窗口，此个窗口也是场景，即场景上弹出场景。

分析下过渡。过渡主要功能是显示一进度条，让用户看到完全进入新场景或对话框大概还须要多少时间。它也是一个窗口，按 Rose 分类规则属于对话框，只是它可能简单到就一个进度条控件，也不须要支持用户交互。

到此小结 app 框架。一旦存在窗口，app 在任何时刻或处于对话框或处于场景。针对两种窗口，使用不同消息循环。对窗口构成元素，对话框中的全是控件，场景则还要包括大地图，大地图不是控件。

3.2.1 模态和非模态对话框

不论是对话框还是场景，在它之上都可再弹出对话框，而这个对话框可能是模态和非模态。模态对话框是不关闭它就没法处理父窗口，而非模态对话框则不用先关闭此对话框也可以处理父窗口。以功能完整性说，自然是两者都需要支持，但 Rose 不支持非模态对话框。

对于记本事中“查找”这类小对话框，即使不用非模态对话框也大可用其它方法，其效果可能并不比传统的差。让场景底部出来个动态悬浮区，类似于视频播放器控制面板，要弹出“查找”对话框了，取而代之是在此悬浮区显示查找“对话框”，要关闭时，隐藏悬浮区即可。这个对话框要加引号，它不是独立的，其实是场景中控件。

3.2.2 多层场景

多层场景指的是在场景上再弹出场景。Rose 支持对话框上弹对话框，场景上弹对话框，但不支持多层场景，而且也不支持对话框上弹出场景。

要进入场景，须要先释放掉已有对话框、场景。

3.2.3 单线程和多线程

Rose 以单线程实现框架逻辑。

增加线程无疑会增加系统开销，像调度开销，但采用单线程主要原因不是开销，而是同步。让多个线程可以去设置同一按钮，这就存在争抢问题，事实证明，再严格的代码都很难安全控制此类同步。使用单线程，一般人很快想到的是害怕不能并发播放动画，依据人眼视觉停留特点，只要刷新足够快，即使用单线程依次处理多控件，用户还是会认为这些控件是在并发显示。

虽然尽量不要使用多线程，但有时不得不使用，像播放声音、处理网络和后台任务。SDL 内置实现了用新建线程播放声音，处理网络和后台任务则要基于 Webrtc，具体见“**3.9 Webrtc**”。

3.2.4 events::pump()

接下分析两类消息循环，它们都基于 events::pump。events 是名字空间，集中存放和事件相关代码，pump 是基于 SDL_PoolEvent（一个函数实现收集、读取事件，参考“**2.5 处理事件**”）的再封装，是 Rose 分发消息的核心函数。

```
void pump()
{
    std::vector<SDL_Event> events;
    while (SDL_PollEvent(&temp_event)) {
        events.push_back(temp_event);
    }
    std::vector<SDL_Event>::iterator ev_end = events.end();
    for (ev_it = events.begin(); ev_it != ev_end; ++ev_it){
        SDL_Event& event = *ev_it;
```

```
        switch (event.type) {
        }
        const std::vector<handler*>& event_handlers = contexts.handlers;
        for  (size_t  i1  =  0,  i2  =  event_handlers.size();  i1  !=  i2  &&  i1  <
event_handlers.size(); ++i1) {
            event_handlers[i1]->handle_event(event);
        }
    }
    instance->pump();
    for   (size_t  i1   =   0,   i2   =   pump_monitors.size();   i1   !=   i2   &&   i1   <
pump_monitors.size(); ++i1) {
        pump_monitors[i1]->monitor_process(info);
    }
}
```

上面代码提取了 pump()主要操作。它不断用 SDL_PoolEvent 读取事件，针对得到的每个事件，调用挂接的事件处理者（contexts）。处理完所有事件后，它依次调用挂接的监控者（monitor_process）。

pump 会调用两种处理者。事件处理者（contexts）：它们关心的是事件，只有事件发生时才会调用它们。监控者（pump_monitors）：它们须要的是有时间片让执行私有操作，即使没发生事件时也要被调用。基于 pump 逻辑，对于希望处理事件的对象，它须要让自个成为事件处理者集合中一员，希望分到时间片的则要让进入监控者集合。

成为事件处理者。要成为事件处理者首先是要让对象从 tevent_handler 派生，整系统就两种对象要成为事件处理者，一是 controller_base，二是窗口子系统（gui2::event::thandler）。是对话框时，只 gui2::event::thandler 生效，是场景时两个生效。gui2::event::thandler 是对窗口子系统的处理入口，它把事件发向各对话框，对话框再发向内中各控件。controller_base 则让场景中的大地图拥有处理事件能力。举个例子，彩图 1 中，大地图需要关心鼠标移动在什么地方，即关注 SDL_MOUSEMOTION 事件，移动到一个感兴趣格子上时，它可能就让某处高亮。但彩图 1 中，发生在大地图外的事件划归 gui2::event::thandler，像右侧控件列表。

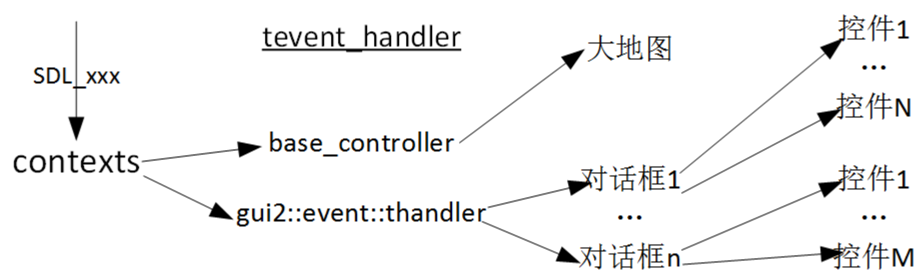


图 3-1 发分事件

contexts 存储着所有事件处理者。对话框时，只有唯一的 gui2::event::thandler。场景则存在两个，第一个是 controller_base，第二个是 gui2::event::thandler。这么排序是基于两个原因，原因一,controller_base 表示了整场景，gui2::event::thandler 是场景中对话框部分，就通常认为的先父后子，应该先存储 controller_base。原因二，场景会出现改变分辨率操作，一旦发生会从 contexts 移除 gui2::event::thandler，然后重建，为避免出现临时的次序混乱，先存储一定不会变的 controller_base。

收到事件后，要分发向内中希望处理此事件的对象。对 controller_base，分发对象就是大地图，Rose 给处理了。gui2::event::thandler 较为复杂，它把事件分发向当前存在那些个窗口，由窗口再分发向内中控件。对窗口（twindow），Rose 已执行了让它成为分发处理者。app 要关心的是让控件成为进一步被分发的对象，代码要做的是把具体控件和哪消息挂钩。

```
connect_signal_mouse_left_click(
    find_widget<tbutton>(&window, "help", false)
    , boost::bind(&trose::set_retval, this, boost::ref(window), retval));
```

上面实现了在“help”按钮按下鼠标左键消息时会调用 trose::set_retval。一般在 tdialog 的 pre_show 执行挂接。

成为监控者。要成为监控者首先是要让对象从 events::pump_monitor 派生。

```
class tlobby: public events::pump_monitor
{
    void monitor_process();
}
```

以上把 tlobby 注册为监控者，一旦 events::pump()被执行，它的 monior_process 就会被调用。

两种处理对象除了要从指定类派生，还须要执行个 join 操作。它们都有提供构造时自动加入功能，但可能碰到构造时不满足加入条件，这时就要在构造之后的恰当时刻调用“join”。events::pump 只会调用加入后的对象。

“instance->pump()”处理网络收发，以及和 Webrtc 多路事件分离器相关的事件，具体见“**3.9 Webrtc**”。

3.2.5 消息循环：对话框

tdialog 实现了对话框框，app 为处理对话框须要增加个从 tdialog 派生的对话框类。对话框的消息循环是 tdialog::show，具体的循环代码是放在 twindow::show。

```
twindow::show(...)
{
    ......
    for (status_ = (status_ == REQUEST_CLOSE)? status_: SHOWING; status_ != REQUEST_CLOSE; ) {
        // process installed callback if valid, to allow e.g. network polling
        events::pump();
        // Add a delay so we don't keep spinning if there's no event.
        SDL_Delay(10);
    }
    ......
}
```

理解 events::pump 功能，就基本知道这循环能做什么了。它不断读取消息、处理消息，调用监控者的 process，直到哪个处理过程把 status_变量置为了 REQUEST_CLOSE。

在调用栈上，tdialog::show 是 twindow::show 父函数。tdialog 是所有对话框基类，app 在创建对话框后，就可调用 tdialog::show 让进入消息循环，类似以下代码。

```
gui2::ttitle_screen dlg(game.heros(), group.leader());
dlg.show();
```

app 可认为 dlg.show 是个阻塞式调用，执行逻辑在此被阻住了，除非退出对话框，该函数才会返回。

对话框上弹出对话框

tdialog::show 是阻塞式调用，而系统是单线程实现，存在多层对话框时系统控制权停留在最顶层对话框的 tdialog::show。

此时存在个问题，多层对话框时是否会渲染底下父窗口？——肯定不会，而且希望以后都不会。浮动在对话框上的功能建议用悬浮控件，像提示。

3.2.6 消息循环：场景

base_controller +display+base_map 实现了场景框架。要没意外，app 须要定义三个从它们派生的子类。场景时消息循环的基本格式是外层用 for/while，内部执行 play_slice。

```
void controller_base::play_slice(bool delay_enabled = true)
{
    events::pump();
    scrolling_ = handle_scroll(key, mousex, mousey, mouse_flags);
    get_display().draw();
    if (delay_enabled && (SDL_GetAppState(video_.getWindow())&SDL_APPACTIVE) == 0) {
        get_display().delay(200);
    }
```

```
}
```

上面代码提取了 play_slice()主要操作。之前已解释过 evets::pump，handle_scroll 是处理滚屏大地图，display.draw()则是画场景界面，包括大地图、各样动画和状态报表。delay_enabled 是 ture 时表示场景不在 ACTIVE 状态，为节省 cpu 消耗，两次刷新间隔让至少 200 毫秒以上。什么是不在 ACTIVE 状态？一种比较常见的是 PC 平台，app 被最小化到任务栏。

play_slice 是加了场景私货的“events::pump”，是时间片操作，用它就可构造消息循环了。

```
int base_controller::main_loop()
{
    while (!do_quit_) {
        play_slice();
    }
    return quit_mode_;
}
```

那这个消息循环放哪里？“**6.12 顶层逻辑**”指示了一种标准用法，即在构造控件器后就让代码进入 main_loop，实现阻塞。当然，app 按自个需要可灵活放置。举个例子，正编写的是一个回合策略游戏，回合游戏特点是轮到玩家操作时只能等到玩家按下“结束回合”才能进入自动操作，于是可以在等待玩家操作过程中放置消息循环。

```
void playmp_controller::play_human_turn()
{
    ......
    while (!end_turn_) {
        play_slice()
        ......
    }
    ......
}
```

玩家回合时，app 用个消息循环等待玩家操作，有操作就处理，直到按下“结束回合”把 end_turn_变量置 true。

除了 controller，等待动画结束往往也需要存在这个消息循环。要播放动画了，往往要用一个函数就处理播放此动画的整个过程，即一旦函数退出就意味着结束了动画播放。为达到这目的，动画开始到等待结束这过程就可置上个消息循环，让等待动画结束过程中可处理用户输入。

```
void play_xxx_anim(...)
{
    ......
    bool finished = false;
    while (!finished) {
        controller->play_slice();
        disp->delay(10);
        finished = anim.animation_finished_potential();
    }
    ......
}
```

一旦动画播放结束，anim.animation_finished_potential()会返回 true。此个消息循环可位于另一个场景消息循环中，由于这两个循环位于同一线程，在执行第二个时线程被阻了，肯定不会进入第一个，要等到退出第二个后才会把控制转权交给第一个。

当前不支持对话框上弹场景，因而不会出现场景循环位于对话框循环内。

场景上弹出对话框

场景、对话框逻辑都是处在同一线程，场景上弹出对话框，控制权将归 tdialog::show，即对话框消息循环。

3.3 资源包

3.3.1 沙盒（sandbox）

在计算机安全领域，沙盒是一种安全机制，为运行中的app提供的隔离环境，通常是作为一些来源不可信、具破坏力或无法判定意图的app提供实验之用。这里把这概念引过来，用它描述一个app对应一个独立目录，这个独立目录称为沙盒。针对当前操作系统，使用上较相似是苹果系统，做到不管是资源还是可执行程序合放一个目录。基于这种方式，安装app就是复制一个目录，卸载app就是删除这一目录。

以上说“一个app对应一个独立目录”，但现实是一些操作系统达不到这个目标，像Windows就须要两个目录。app数据包括两类：资源数据和文档数据。资源数据特点是安装时复制过去，之后app运行时不会增加、减少、修改，即一直保持不变。文档数据则是安装时没有，但伴随运行会不断变化。Windows建议app的使用规则是把资源数据放在“Program Files”，文档数据则放在“用户”目录。以下是Rose在三个操作系统上对这两个目录的处理规则。

	Windows	iOS	Android
资源和文档关系	资源可任意放置，用户放在“用户”目录	用户是资源下的“Documents”子目录	用户是资源下的“Documents”子目录
资源目录如何保证纯英文	强制用户安装时使用纯英文	系统安装会保证是	系统安装会保证是
文档目录是否必定纯英文	不一定	一定	一定

Rose要求资源目录必须是纯英文，而且资源目录中所有子目录名、文件名都是纯英文，原因见“第八章 国际化和gettext”。既然子目录、文件都纯英文了， Windows文档目录不一定是纯英文的原因是目录名本身用了非英文字符，像“C:\Users\Administrator\我的文档”。

综上所述，可以把沙盒理解为是app目录，具体包括资源目录、文档目录。

3.3.2 工作包、app包中的资源包

Rose在设计资源包结构时考虑到了让用户同时开发多个app。按是否属于app私有，资源包中数据分为通用数据、app私有数据，以及关联这两部分的衔接数据。资源包中有四个位置让app出现私有数据，它们分别用于存放多媒体文件、配置、gui和翻译。无论哪部分，入口目录的名称都是“app-<appid>”。

	数据存放位置	内容	app.cfg 目录	app.cfg 内容示范
app 资产	<res>/app-{app}/images/assets	图像		
多媒体文件	和<res>/data 同级目录	图像、声音	不须要	
配置	<res>/data 的子目录	分为两类。一是要融入核心配置的数据，像动画、单元。二是私有bin	<res>/data	{app-aismart/_main.cfg} {app-studio/_main.cfg}

Gui	<res>/data/gui 的子目录	控件、场景、对话框	<res>/data/gui	{gui/app-aismart/_main.cfg} {gui/app-studio/_main.cfg}
翻译	<res>/po 的子目录	和翻译相关的 pot、po	不须要	

用于衔接的资源就两个文件，它们都叫 app.cfg。根据工作包当前有哪些 app，Studio 会自动生成这两个文件。

3.3.3 app 资产

它们是一些图像文件，下表列出了名称、尺寸和作用。

名称	尺寸	作用
ic_launcher-48.png	48x48	Android 的桌面图标(mdpi)
ic_launcher-72.png	72x72	Android 的桌面图标(hdpi)
ic_launcher-96.png	96x96	Android 的桌面图标(xhdpi)
ic_launcher-144.png	144x144	Android 的桌面图标(xxhdpi)
ic_launcher-192.png	192x192	Android 的桌面图标(xxxhdpi)
Icon-76.png	76x76	iOS 的桌面图标
Icon-120.png	120x120	iOS 的桌面图标
Icon-152.png	152x152	iOS 的桌面图标
Icon-1024.png	1024x1024	iOS App Store 上的图标（不能含 alpha 通道）
Default-568h@2x.png	640x1136	iOS 的启动图像
Default-375w-667h@2x.png	750x1334	iOS 的启动图像
Default-414w-736h@3x.png	1242x2208	iOS 的启动图像（同时用于各平台）

开发者必须提供表中加粗的 3 个文件，其它文件则依照“如果有提供同名文件，使用该文件，否则由大分辨率文件缩小生成。

Android、iOS 桌面图标下的名称是 appid 的 msgstr，什么是 msgstr 见“**3.8 国际化**”。

3.3.4 配置、WML、config

让具体深入资源包，看会包括哪些数据，图像、声音、字库、支持国际化的包（po、pot、mo），除去它们还有一种是配置。配置分为静态和动态，Rose 使用 WML 语法描述静态配置。

```
[window]
    id = "browse"
    [resolution]
        definition = "default"
    [/resolution]
[/window]
```

WML 配置的基本单元就两种：块和属性，块下除属性外可再接子块，例子中定义了“window”、“resolution”两个块，其中后者是前者子块。针对 WML 中的块，程序和它对应的是 class config，在 config，子块叫“child”，属性叫“attribute”。Rose 就是通过块级连的方法把配置集中为几个大块，代码就是调用 config::child 等函数去访问具体哪个子块。

因为安全、容量等原因，以 WML 格式描述的文件不适合发布，Rose 提供了工具把它们二进制化，即把以 N 个文本格式写的配置“编译”成比 N 数量小得多的二进制格式文件。二进制化的文件被统一放在资源包的“xwml”目录，以下是该目录中文件。

文件名	格式		功能
data.bin	cfg		应用数据
gui.bin	cfg		窗口系统
language.bin	cfg		支持的语言

tb-hexagonal.dat	构造规则		正六边形构造规则
tb-square.dat	构造规则		正方形构造规则

```
void wml_config_from_file(const std::string &fname, config &cfg, uint32_t* nfiles, uint32_t* sum_size, uint32_t* modified);
void wml_config_to_file(const std::string &fname, config &cfg, uint32_t nfiles, uint32_t sum_size, uint32_t modified);
```

wml_config_from_file 执行从文件 fname 读出配置到 cfg。wml_config_to_file 执行逆操作，把配置 cfg 写入 fname 文件。

3.3.5 独立块：[capabilities]、[settings]

独立块指的是不在任何*.bin 中的块，要用到当中数据时用 get_config 实时获取。

[capabilities]用于存储开发 app 时工程时设置，像是否开启蓝牙。它只和软件开发时有关，和运行时无关。作为独立块的原因是让修改后不必重汇编哪个*.bin。

[capabilities]是 app 私有，像<res>/data/app-kingdom/capabilities.cfg。

[settings]块包括两部分，一是 rose，二是 app 私有。存储的内容是全局设置，是 game_config 名字空间的那些个字段。它们是在 data.bin 之前被读取。作为独立块的原因是一些读 data.bin 之前的一些操作要用到些变量，而这些变量就可放在[settings]块。

gui.bin 也有一个[settings]，它放在[gui]块根下，存储着 gui 系统的全局设置。

3.3.6 app_cfg_、core_cfg_

app_cfg_、core_cfg_是 base_instance 内的 config 类型的成员变量，它们都和 data.bin 有关。初始化阶段，base_instance 执行 load_data_bin 把 data.bin 读入内存形成 app_cfg_，考虑到存储 [terrain_type]块需要不少内存，移除[terrain_type]后形成一个新的 config，它就是 core_cfg_。另外，app 可重载 app_init_load_data_bin 进一步裁剪 core_cfg_。

变量	描述
app_cfg_	load_data_bin 执行后，它的值会赋给 core_cfg_。之后，app 用它做私有目的，像把 core_cfg_和哪个私有 bin 组成新的 app_cfg_，然后把它传给场景。
core_cfg_	对应去除了根下[terrain_type]块的 data.bin。建议之后就不要再改变内容。

app 用到的应该是 app_cfg_，那为什么要存在 core_cfg_？要生成新的 app_cfg_，则需要读出 data.bin 中原始内容，core_cfg_充当了 data_bin.cfg 内容缓存，避免再读 data.bin。

3.3.8 图像、声音

要没意外，图像文件放在“images”目录，声音分为音效、音乐，音效放在“sounds”，音乐放在“music”。“images”、“sounds”、“music”是保留用做固定用途的三个目录名称，app 不要把它们用作其它用途。

app 要用到众多图像、声音，这些文件不是被归到一处，而是分散在多个位置。在不同位置有不同优先级，下表以优先级由高到低列出了所有可搜索位置。

位置	标识	紧接类别目录	备注
文档目录根	<userdata>/	否	文档目录存放“最”新文件，应而把它优先于资源目录
文档目录类别	<userdata>/<type>	是。类别指当中<type>，也就是 images、sounds、music 中一个	
资源目录根	<res>/	否	注 1

会话新增路径	<res>/…/<type>	是	注 2
app 路径	<res>/app-<app>/<type>	是	<app>是 app 标识
Rose 路径	<res>/data/core/<type>	是	最低优先级，rose 内置的图像、音效、音乐

注 1：为什么会存在不须要 type 的目录？这不是说要访问资源目录根下图像文件，往往是要访问省略掉 game_config::path 的“长”文件名，像"data/campaigns/duel/images/image.png"。

注 2：什么是会话新增路径？假设正在写战棋游戏，它有两个剧本，每个剧本对应一个目录，分别是 campaigns/duel、campaigns/siege。游戏运行到 duel 剧本，为方便可以新增 campaigns/duel 作为一个可搜索路径。一旦要改为运行“siege”剧本，这时从可搜索路径中删除 campaigns/duel，改为增加 campaigns/duel/siege。而以上或正运行“duel”、或正运行“siege”称为会话。，

管理会话路径（binary_paths_manager）

一个[binary_path]块添加一条路径。一个会话可定义多个[binary_path]，即增加多条路径。

```
[binary_path]
    path = data/campaigns/duel
[/binary_path]
```

以上 WML 语句实现把“data/campaigns/duel”加入可搜索路径。在目录位置上，path 指定的必须是相对于 game_config::path 的路径。在对系统影响上，此次会话会向系统增加“我”的自有路径，不会清掉之前路径，它退出后只是删除“我”增加的路径。如果一会话增加多条，那优先级是按照路径定义次序。如果出现同时运行两会话，像在会话 A 上弹出会话 B，结果形成的可搜索路径是两会话和，而后出现的 B 优先级要高于 A。

本地化图像文件

它有点类似多国语言翻译，只不过依据系统正使用语言，翻译改变的是文字，本地化图像则是图像。举个例子，要在开始窗口显示 Logo，在这个 Logo 上要显示“王国”图标，在中文时就要让显示画有“王国”文字的 PNG 图像，英文时则改为显示画有“Kingdom”文字的图像。

为支持本地化图像需创建两级目录，第一级是在参照路径下新增“l10n”目录，第二级是“l10n”目录下创建以各语言为名称的目录，像简体中文是 zh_CN，英文 en_US（英文往往会作为通用图像，所以基本不会出现以 en_US 命名的本地化目录）。举个例子，<res>/images/icons/logo.png 指示开始界面要用到图像，这时就可在“icons”（称 logo.png 为参照图像，它所在的目录“icons”称为参照目录）下新建“l10n”，“l10n”下新建“zh_CN”，在“zh_CN”创建“logo.png”。一旦完成，当系统正运行简体中文时会选择这个 logo.png，而不是 icons 根下的那个。

要能让本地图像有效，必须存在参照图像。像以上给的例子中“icons”根下必须存在“logo.png”。

3.4. 用户数据

用户数据包括配置（config 格式）、存档等，config 格式的配置统一存放在 preferences 文件。意外原因会使得 preferences 被破坏。举个例子，**修改 preferences 后，设备立即断电，虽然代码已成功调用 fclose，但断电时间够快，文件还是可能被破坏，像长度变 0。**Rose 已为 preferences 使用 bak 机制，但还是有极低概率失效。

3.4.1 关键参数

app 可能存在几个参数，一旦出错会连 app 都无法启动，对这些参数，Rose 在 bak 上再提供叫关键参数的机制，要使用它，app 要做的是两个，一是在 app_load_settings_config 调用 get_critical_prefs 找回丢了的关键参数，二是重载一个叫“app_critical_prefs”的函数。此机制的过程包括读取时和写入时，

```
virtual config base_instance::app_critical_prefs();
```

读取时。1）app 启动后，加载用户配置时会读 critical_preferences 文件，放在 critical_prefs 这个 config 对象。2）app 在实现 app_load_settings_config 时，发现关键参数丢了，可调用 preferences::get_critical_prefs()得到 critical_prefs，从 critical_prefs 找回丢了的关键参数。

写入时。preferences::write_preferences 会率先调用 app 自实现的 app_critical_prefs。此函数返回值是 config，存放着 app 认为的关键参数。如果非空，它就会被写入 critical_preferences 文件。如果 app 此时还没准备好“所有”关键参数，返回值置空。

关键参数本质是通过减少写入 critical_preferences 次数，从而减少破坏 critical_preferences 的概率。开发者需很清楚哪几个是关键参数，改了这几个参数后，要确保 critical_preferences 不被破坏，像设备不会被立刻重启。

3.5 声音和图像

sound 是个名字空间，集中存放处理声音相关代码，处理声音较清晰分成两个阶段：初始化、运行时调用。图像则可认为没有初始阶段.。

SDL 如何处理声音参考“**2.4.4 播放 API**”，为全面这里抄几句重要结论。SDL_mixer 把声音分成音效、音乐，音乐表现在有较长的持续时间，常见例子像歌曲，程序的背景音乐；音效表现在较短的持续时间，可能就是毫秒级，常见例子像按下按钮时提示音。可以“同时”播放数首音效，但在任一时刻只能播放一首音乐。通道是个和音效相关的概念，和音乐无关。音效播放优先级要高于音乐。

3.5.1 初始化播放声音

初始化声音就一步骤，调用 sound::init_sound。它除调用 Mix_OpenAudio 初始化 SDL_Mixer，还有个功能是初始化音效通道。开发者需要知道 init_sound 如何分配通道，以及各通道别名。

Rose 申请 16 条通道，其中 11 条是保留通道。

位置	别名	备注
0	SOUND_BELL	
1	SOUND_TIMER	
2 到 9	SOUND_SOURCES	它使用较为特殊，可以修改声音发出位置
10	SOUND_UI	用户界面音。像按下按钮
11 到 15	SOUND_FX	非保留通道

3.5.2 运行时播放音效

```
void play_bell(const std::string& files);
void play_timer(const std::string& files, int loop_ticks, int fadein_ticks);
void play_UI_sound(const std::string& files);
void play_sound(const std::string& files, channel_group group = SOUND_FX,
unsigned int repeats = 0);
```

files 是要播放的音效文件。它支持多个文件名，多个时用逗号隔开，系统会随机选一个进行播放。“loop_ticks”指示最长播放多少时间，<=0 表示不限停止时刻，否则在 loop_ticks 毫秒后会强行停止（如果那时这音效还没播放完）。“fadein_ticks”用于指示在 fadein_ticks 毫秒内出现音量逐渐增大效果。“repeats”表示轮回播放次数，0 和 1 一样表示播放一次。

play_bell 限定使用 SOUND_BELL 的通道，play_timer 是 SOUND_TIMER、play_UI_sound 则是 SOUND_UI。play_sound 则可以自个指定哪种通道，默认使用 SOUND_FX。SOUND_FX 有 5 个通道，意味着可以实现"同时"播放 5 个音效，可以缓解播放下一个时前一个没播放完就被强制结束问题，对应用来说，播放音效最常调用的可能就是 play_sound。

补说下为什么要把音效分在多个通道，除去是要让不覆盖之前声效，还有就是要让应用自个去归划音效类别。比如为用户界面音单独划出一通道，对用户界面，基本一次只能按到一处控件，也就是后出来声音可让覆盖掉之前的，而且这才是正确做法。单独划出还可让实现单独设置最大音量、单独使能/关闭，像可实现这么个功能，不播放除用户界面音之外的音效。

```
sound::play_sound("spear.ogg");
```

它让在 SOUND_FX 通道播放“spear.ogg”这个音效文件，只播放一次。

3.5.3 运行时播放音乐

```
void play_music_repeatedly(const std::string& id);

sound::play_music_repeatedly("main_menu.ogg")
```

循环播放一首音乐，id 指示要播放的音乐文件。给出的例子让循环播放“main_menu.ogg”。

play_music_repeatedly 是不断重复播放一首音乐，如果你想让在多个文件中随机播放，那就需要调用 play_music_config、commit_music_changes。

```
void play_music_config(const config& music_node);
void commit_music_changes();

[music]
    name = legends_of_the_north.ogg
[music]
[music]
    name = transience.ogg
    append = yes
[music]
[music]
    name = underground.ogg
    immediate = yes
    append = no
[music]
```

既然要支持在多首音乐间切换，那系统一定定义了表示音乐集合的变量，play_misic_config 指示如何把一首音乐“加入”集合。music_node 指的是一个[music]块，它具体指示如何加入。以下是它当中可以出现的属性。

属性	语义	备注
name	音乐文件名	必须设置
append	yes 时加入已在集合，no 时让集合只有它自个	默认是 no。意味着要先清空集合。
immediate	yes 时立即播放它	默认是 no。
play_once	yes 时立即播放，并且只播放一次	默认是 no。
ms_before	切换到播放它时，ms_before 毫秒内出现逐渐增强效果	默认是 0
ms_after	它要被切换出去时，再播放 ms_after 毫秒时间，这段时间出现逐渐减弱效果	默认是 0

程序可能是一次处理多个[music]块，即多次调用 play_music_config，在调用完它们后记得调用一次 commit_music_changes。

3.5.4 载入、保存图像

在处理图像上，SDL_image 已做了很多，直接调用它的函数就已能实现载入、保存图像。

```
SDL_Surface* IMG_Load(const char* file);
int IMG_SavePNG(SDL_Surface* surface, const char* file);
```

IMG_Load 实现载入 file 文件到图面数据。IMG_SavePNG 执行保存，以 PNG 格式压缩图面数据，并保存到 file 文件。

3.6 动画

简单的动画就是按一定时间间隔显示数张图像，但这的确有点简单了，不少场合要做到在某一时刻“并发”显示数张图像，这时就要事先做出动画素材，素材内容包括图像、声音、时间间隔和更换逻辑。按使用场合，动画类蕈可分为单元模板、单元非模板、地图、图元和窗口。要深入动画细节看“**第七章 动画**”。

动画类型	素材类型	使用场合	更新函数	渲染方式
单元模板	单元模板	大地图中和单元有关、并且要进一步具体化到兵种的动画。参考位移是以主角色为起点、终点到从角色的直线	display::draw_units	累加式
单元非模板	单元非模板	大地图中和有单元有关，所有单元都一样的动画	display::draw_units	累加式
地图(map)	区域	大地图中和单元无关的动画	display::draw_units	累加式
图元(canvas)	区域	控件相关、一定是周期性的动画。包括混排动画	tcanvas::draw	恢复式
窗口(window)	区域	位在窗口顶层，往往用于表示哪事件的动画。只出现在对话框	display::draw_window_anim	恢复式

在渲染方式上，和地图相关的动画都采用累加式。不论哪种动画，要使用它们首先得定义动画素材，如何定义参考“**第七章 动画**”，这里只说代码让如何播放这些动画。

3.6.1 动画素材

按素材不同，动画分为单元模板动画、单元非模板动画和区域动画。

素材类型	id 字段语义	存储位置
单元模板	指示是哪一类动画，它不是全局唯一 ID，可被多个块重复使用。有相同 ID 的素材块表示它们是同一类动画。代码把它称为 tpl_id	模板数据（config）放在 instance::utype_anim_tpls_，扩展（utype_anim_create_cfg）后放在兵种
单元非模板	用它来标识该动画，需全局唯一	instance::anims_
区域	用它来标识该动画，需全局唯一	instance::anims_

单元动画是分到模板还是非模板的依据是该动画是否和特定的兵种数据相关。对一个把部队分为步兵、骑兵、炮兵的游戏，部队移动动画是模板动画，因为在移动时要显示特定兵种的图像。灭掉部队后获得金币的动画则可是非模板动画，动画就是让金币图像从部队位置移到金币显示位置，这和部队特定图像无关。但无论哪种，参考位移都是以主角色为起始，素材类型是基于单元。在内存消耗上，板模动画是“种类数*该模板动画内存”，而非模板动画的“种类

数”肯定是一。

继续说模板动画。假设部队移动动画要分为两种，一种是陆上移动，一种是水上移动，那么这两个动画素材的 id 都可命名为“movement”。一旦扩展开，任一兵种就都会有两类“移动”。任一素材是用一个[anim]块表示具体细节，模板扩展过程会重命名该块名，因为把该兵种所有支持的动画放到同一块下，都叫“anim”会使得无法区分它是什么功能动画，为此在扩展时会根据它是什么功能改为私有块名，像[movement_anim]。注：这个名称可以不是“id”。为什么不是？像步兵部队的攻击动画，它有两种攻击方式，在定义 id 时分别叫“penetrate_attack”、“cyclone_attack”，但在扩展后动画块名上，它们都叫“melee_attack”。

3.6.2 注册、加载素材

注册动画素材目的是告知此次运行可使用哪些动画。一个动画素材对应配置中的一个 WML 块。素材按来源分为两类：Rose 内置的素材、app 私有素材。

anim2 是存放处理动画代码的名字空间，anim2::fill_anims 可认为是准备素材的顶层调用，代码从资源包读出 game_config_后就会调用这函数。

```
void fill_anims(const config& cfg)
{
    fill_tags();

    BOOST_FOREACH (const config &anim, cfg.child_range("animation")) {
        const std::string id = anim["id"].str();
        const config& sub = anim.child("anim");
        bool area = sub["area_mode"].to_bool();
        bool tpl = !area && anim["template"].to_bool();
        int at = NONE;
        if (area || !tpl) {
            at = find(id);
        }
        instance->fill_anim(at, id, area, tpl, sub);
    }
}
```

fill_tags()负责准备可用的动画标识，这个标识是那些需全局唯一的标识，具体包括“单元非模板动画”和“区域动画”，在 WML 中是用字符串（即 id 字段）表示标识，C/C++代码则要让映射到一个整型值。以上有说到可用的标识分为来自 Rose 和 app 私有，fill_tags()分别从这两处形成 tags。

```
void fill_tags()
{
    anim_field_tag::fill_tags();
    if (!tags.empty()) return;
    tags.insert(std::make_pair("operating", OPERATING));
    ......
    instance->app_fill_anim_tags(tags);
}
```

"operating"是内置的一种动画素材，OPERATING 是和它对应的整数值。函数结尾会调用 app_fill_anim_tags，一个 app 须自写的函数，负责把私有标识加入 tags。由于整数标识在代码内必须唯一，app 需要知道 Rose 内定了多少标识，自个标识则从它之后开始定义，MIN_APP_ANIM 就是做这个用的，它指示 app 可使用的最小整数值。

填充可用标识后，注册阶段就结束了，接下是逐个加载动画 WML 块、然后生成对应的动画对象，它就是接下要执行的 fill_anim。在生成动画时，要根据不同素材类型进行不一样操作，Rose 给了 fill_anim 的默认实现，但 app 极可能要重载它。

```
void game_instance::fill_anim(int at, const std::string& id, bool area, bool tpl,
const config& cfg)
{
    if (area) {
        anims_.insert(std::make_pair(at, new animation(cfg)));
    } else {
```

```
        if (tpl) {
            utype_anim_tpls_.insert(std::make_pair(id, cfg));
        } else {
            anims_.insert(std::make_pair(at, new unit_animation(cfg)));
        }
    }
}
```

app 的差别极可能是如何处理非模板单元动画，Rose 定义的动画基类是 animation，app 有可能自个写了派生类，此时就要重载 fill_anim，上面代码用 unit_animation 去代替 animation。

综上所述，在注册、加载阶段 app 要做的是重载“app_fill_anim_tags”、“fill_anim”。

3.6.3 使用图元动画

图元动画一定是周期动画。可用两种方法让控件拥有图元动画，初始时配置和动态增加，由它们产生的分别称为固定和动态图元动画。图 3-2 显示了图元动画在 Z 轴上位置。

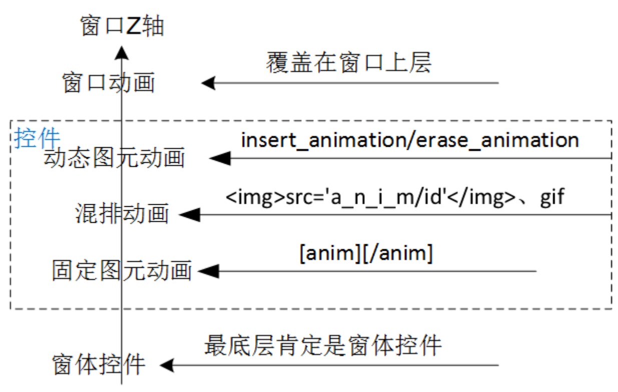


图 3-2 各种动画在 Z 轴位置

类型 1：初始时配置（固定图元动画）

固定图元动画是作为控件的一个图元，在控件相应画布处使用[anim]块，以下是当中可能出现的字段。

字段	值	功能	
id	字符串	动画标识	必须设置
width	整数	标称宽度	按须设置
height	整数	标称高度	按须设置

不能动态删除初始时配置的动画。

类型 2：动态增加（动态图元动画）

```
int tcontrol::insert_animation(const config& cfg);
void tcontrol::erase_animation(int id);
```

参数 cfg 就是[anim]块中内容（见类型 1），位置见图 3-2，总位在控件顶层。insert_animation 向控件增加一个动画，后加的动画会放在之前的上面。函数返回的是整数 id，该值动态分配，和静态分配的动画整数标识值无关。函数执行完后，动画就自动开始播放了。

如果过程中（包含该控件的窗口还在显示时）须要动态删除则调用 erase_animation，参数 id 就是 insert_animation 时返回的整数 id。

混排动画

滚动编辑框中内容采用图文混排，在这两种控件中混排内容可使用动画，此中的动画称为混排动画。要使用它的一个例子是聊天，像要让聊天内容支持插入 GIF。

播放混排动画被限定在一个固定区域，其它混排单元不能进来这区域。

两种途径会让出现混排动画：1）内容中使用<img>标签，src 值以“a_n_i_m/”为前缀，后跟动画标识，像“src='a_n_i_m/perfect'”，它会加入标识是“perfect”的动画；2）碰到 GIF。

在位置上，混排动画叠加在固定图元动画之后，动态图元动画之前，参考图 3-2。

3.6.4 使用窗口动画

窗口底层都有个窗体控件，和其它控件一样，它可使用固定/动态图元动画。见图 3-2，窗体的动态图元、窗口动画都会位在窗口顶部，那根据什么决定要用哪一种？图元动画一定是周期动画，窗口动画的作用往往指示发生哪事件了，是非周期动画。为此可用是否周期性作为判断依据，是周期性的使用图元，否则窗口动画。

app 调用 start_float_anim 开始播放，过了时间后系统会自动删除这动画。编程时，你可能须要知道动画何时结束。举个例子，要用动画表示武将升级事件，为避免误操作，动画开始后要变灰“升级”按钮，这就要求检测到动画何时结束，以便结束时按钮恢复到正常状态。为此须要编写 finish 函数，tgroup2::upgrade_anim_finish。

```
void set_callback_finish(boost::function<void (animation*)> callback)
```

步骤一：编写 finish 函数

```
void tgroup2::upgrade_anim_finish(animation* anim)
{
    twindow& window = static_cast<twindow*>(ctx);
    tbutton& widget = find_widget<tbutton>(window_, “upgrade”, false);
    widget.set_active(true);
}
```

步骤二：编写开始播放函数

编写它有固定逻辑。一开始调用 start_float_anim_uh，如果有 finish 调用 set_callback_finish 执行挂接，接下处理此个动画的私有操作，像修改具体图像，最后调用 start_float_anim_bh 开始播放。

```
void tgroup2::start_upgrade_anim(hero& h, const std::string& description)
{
    float_animation* anim = start_float_anim_th(disp_, anim2::UPGRADE, NULL);
    anim->set_scale(800, 600, true, false);
    anim->set_callback_finish(boost::bind(&tgroup2::upgrade_anim_finish,    this,
_1));

    std::stringstream strstr;
    anim->replace_image_name(null_str,       "__bg.png",       "tactic/bg-single-
increase.png");
    anim->replace_image_name("id", "__id.png", h.image(true));
    anim->replace_static_text("formation", "__formation_name", h.name());
    anim->replace_static_text("description",          "__formation_description",
description);

    start_float_anim_bh(*anim, false);
}
```

步骤三：等待播放完成

此时 app 可以啥都不做，一旦动画结束，系统会调用注册的 finish 函数。

3.7 超文本（Hypertext）

关于超文本的详细内容见网上文档超文本。以下只是当中和 app 较相关部分。

说到超文本，立刻会想到网页（HTML），支持图文混排的同时将各种不同空间的文字信息组织在一起的网状文本。技术实现上，用一串字符让达到超文本效果，具体到实现格式是在字符串中加标记（Markup）。Rose 中超文本类似 HTML，相比网页较多强调超链接，它更关心图文混排，糅合各种文字效果，等等。除了传统只读显示，还支持编辑。

app 的超文本编程分为两部分，分别对应 Rose 提供的两类 API：基础，控件相关。

3.7.1 基础 API

```
surface get_rendered_text(const std::string& text, int maximum_width, int font_size, const SDL_Color& color, bool editable)
```

得到该串字符对应的 surface。参数 maximum_width 指定容许宽度，值是 0 时，将使用 INT32_MAX（这时意味着 text 只会是一行）。

```
void split_integrate_src(const std::string& text, const std::set<std::string>& support_markups, std::map<int, tcfg_string_pair>& ret)
```

分割字符串，分割结果放在 ret。ret 中的每个单元是个“key、value”结构，key 指示了该单元在源字符串中偏移。value 则要分两种情况，不是标记时，text 字段指示这串字符，是标记时，cfg 存储着它被解析成的 WML 块，text 指示着标记名称。

3.7.2 控件相关 API

```
void tcontrol::set_label(const std::string& label)
const std::string& tcontrol::label() const
```

不论是只读的文本控件，还是可编辑的编辑框控件，都用这两函数设置/读取当中的字符串内容。

3.7.3 转义符

标记外部不需要转义符，但在标记内部需额外处理“"”。“"”指示字符串的开始和结束，那如何处理字符串中的“"”，只能通过转义符，即写成“\"”格式。由于使用了转义符，“\”也成为特殊符号了，所以标记内部需同时转义“\”、“"”。

由于在标记内不得不使用转义符，app 在编程时注意两点。1）尽量不要自写标记，而是用超文本模块提供的 generate_format、generate_img 这些标记生成函数。2）不得不自写标记时，要对标记中的“\”、“"”进行转义。

3.8 国际化

国际化的任务是让 app 支持多国语言。在处理国际化上，gettext 已做了很多，代码基本上就是去直接调用它提供的函数。处理国际化分两个阶段：准备素材、编程。

3.8.1 准备素材

准备素材指的是准备好 app 运行时需要的语言包资源。在处理资源上，gettext 依次要收集 msgid 生成*.pot，*.pot 生成*.po，再由*.po 生成*.mo。如何快速生成 pot、po、mo？开源社区使用 gettext 已经好多年，有了较好的辅助软件，以下叙述基于 poedit.exe（https://poedit.net）。

生成*.pot。pot 是要翻译的模板文件，存储着 app 需要翻译的 msgid。app 可能需要翻译大量 msgid，手动汇总太累又容易犯错，为此 gettext 提供了一个叫 xgettext 的程序去实现自动收集。xgettext 实现收集时要注意，它只能提取 msgid、不能提取翻译域。注：poedit 提供自动收集功能，借用的就是 xgettext。

```
dgettext("hello-lib", "hello world!");
```

xgettext 能提取“hello world!”，不能提取作为翻译域的“hello-lib”，但编程时极可能出现一个源文件会同时使用多个翻译域。举个例子，main.cpp 主要使用“hello-lib”，又可能偶尔使用“rose-lib”。为了能正确收集，Rose 要求源码使用约定的翻译函数关键字，只有这些函数中 msgid 才会被收集。

```
_("msgid");
vgettext2("msgid", symbols);
```

“_”、“vgettext2”是会自动收集的两个关键字，“vgettext2”是翻译带参数的 msgid，什么是“带参数的 msgid”见“**8.2.3 API**”。如何设置关键字？运行 poedit 打开一个 pot，像<po>/app-studio/studio-lib/studio-lib.pot，“编目”——“属性”——“源关键字”。设置了能从它收集的关键字后，接下要设置收集路径。还是 poedit，“编目”——“属性”——“源路径”，会看到两个

路径。

```
<apps-src>/apps/studio/
<apps-res>/po/app-studio/cfg-cpp/studio-lib/
```

“studio”目录负责收集在 studio 这个 app 的私有源码目录中出现的 msgid，“studio-lib”则存放着从 cfg 文件收集出、应该放在 studio-lib 的 msgid。那“studio-lib”下的文件是怎么生成的？Studio 在构造哪个 bin 时，会收集过程中遇到的 msgid，结束前会把这些 msgid 写到对应 cfg-cpp 下的某个目录。举个例子，构造 data.bin，过程中发现有存在属于“stido-lib”翻译域的 msgid，就会把它们写到<cfg-cpp>/studio-lib 下的 data.cpp。

app 会遇到这么种情况，要翻译的 msgid 既不出现在源码、也不出现在 cfg。像正在写一个游戏，当中会出现 1000 个武将，每个武将姓名以着“name-0”这样格式下去。对这种情况，解决办法是在“cfg-cpp/studio-lib/”下新建一个 cpp，像_bonus.cpp，然后把额外的 msgid 写在那里，格式遵循使用“_”、“vgettext2”函数。注：新建 app 时已默认生成一个_bonus.cpp，它有一个 msgid 是 app 标识的条目，该条目的 msgstr 将作为 Android、iOS 桌面图标下的名称。

设置好了函数关键字和路径，若要生成 pot 了，就运行 poedit，“编目”——“从源文更新”，就可生成 pot。

Studio 在新建 app 时已准备好初始 pot，即设置好了函数关键字和路径。

生成*.po。Studio 在新建 app 时已准备好初始 po。运行 poedit，打开该 po，“编目”——“从 POT 文件更新”，就可把最新要翻译的 msgid 导入该 po。

生成*.mo。打开 po，翻译结束后，“文件”——“保存”，在该 po 文件的同级目录就会生成对应的 mo 文件。然后把这 mo 文件复制到<apps-src>/translations 的对应目录。

3.8.2 编程

编写 cpp 源码时，遇到要翻译 msgid 了，不需要参数的使用“_”函数，否则使用“vgettext2”。

生成 cfg 时，像用 Studio 进行图形化配置窗口，要选对翻译域。

app 默认可使用两个翻译域，一个是 rose-lib，一个是<app>-lib。如果还想使用更多翻译域，那需要在初始化时把它们加载到内存，加载代码放在“base_instane::app_init_locale”。

3.9 Webrtc

说到 Webrtc，第一反应是网络聊天，好的聊天必须建立在稳定、高效的网络模块和多线程同步基础上。多路事件分离器是 Webrtc 中的基础模块，是网络和多线程同步的一个主要部分。对多路事件分离器中的多路，一方面指的是要同时支持多个 socket，另一方面 Webrtc 在处理 socket 的同时处理了唤醒事件，这是种和 socket 无关的事件，它和多线程相关，为此又涉及到了线程模型。

已有不少分析 Webrtc 线程模型/多路事件分离器的文章，这里不想深入代码，只说和 app 编程相关部分，有点像 Webrtc 向 app 提供的网络 API。在说之前首先要知道多路事件分离器处理的两类事件：唤醒事件和 IO 事件。

唤醒事件。唤醒事件和多线程相关。Invoke（内部其实是 Send）、Send、Post 是产生唤醒事件的三大源头。唤醒事件往往用于同步。举个例子，要把一个目录打包成一个文件，打包须要点时间，为达到好的用户体验就要把打包操作放在后台，为此系统就存在两个线程，主线程 A 和用于打包的工作者线程 B。B 在打包时发现须要密钥，而密钥只能在 A 产生，于是它就调用 a->Invoke(...)。Invoke 会向 A 的消息队列压入此个消息，然后向 A 发唤醒事件，并把自个阻塞。A 收到唤醒事件后，获取和该事件相关的消息数据，执行产生密钥。密钥函数执行结束后 B 解除阻塞，恢复运行。

唤醒事件并不总是其它线程发来，有时是自个发出。尤其 Post，向自个发 Post 可实现延迟调用。举个例子，网络通信时发生网卡突然被禁用，于是网络模块就去调用 app 向它注册的钩子函数。钩子函数知道得释放资源，但此时又不能释放，因为钩子函数执行完后网络模块要继

续获得控制权，还须要依赖这些资源。这时钩子函数就可以在自个结尾处放个发向自己的Post，当网络模块处理禁用结束后，所在线程就会调用Post投递的消息，在那里就可以安全释放资源。

IO 事件。IO 事件和 socket 相关，指的就是在 socket 上发生的事件。在 Windows，socket 会触发十种事件，Webrtc 只处理当中 5 种，分别是 FD_READ、FD_WRITE、FD_ACCEPT、FD_CONNECT、FD_CLOSE。一个多路事件分离器可同时处理多个 socket。

3.9.1 线程模型/多路事件分离器

彩图 7 显示了 Webrtc 的线程模型/事件多路事件分离器。Thread 派生于 MessageQueue，PhysicalSocketServer 派生于 SocketServer。MessageQueue::Get 是多路事件分离器入口，大部分功能是通过调用 PhysicalSocketServer::Wait。SocketServer 虽然有 Socket 字样，但不仅能侦听 socket 相关的 IO 事件，还包括唤醒事件。对 IO 事件，不仅侦听还处理，唤醒事件则只是侦听，至于处理是后面的 Dispatch。

MessageQueue 负责接收、存储 Message 类型消息，即唤醒事件，它决定了所有事件（包括 IO 事件）的执行线程。SocketServer 负责阻塞、侦听、处理（IO）事件。

一个多路事件分离器由一个 MessageQueue 和一个 PhysicalSocketServer 组成，这 2 个组件轮流获得控制权。MessageQueue 最先获得控制权，它会检查自己的消息队列，如果有需要立即处理的消息（唤醒事件）就马上处理，如果没有就把控制权交给 PhysicalSocketServer。PhysicalSocketServer 将侦听所有位于其分发器列表（PhysicalSocketServer::dispatchers_）的 IO、唤醒事件。如果有事件被触发，PhysicalSocketServer 将调用对应的 Dispatcher 的消息响应函数（OnPreEvent、OnEvent）。对 IO 事件，OnPreEvent、OnEvent 就把它们处理了，唤醒事件则要等到后面的 Dispatch。

如果在 PhysicalSocketServer 阻塞侦听时 MessageQueue 接收到唤醒事件，MessageQueue 将会调用 PhysicalSocketServer::WakeUp 触发 PhysicalSocketServer::signal_wakeup_ 以解除 PhysicalSocketServer 的阻塞状态。并将 PhysicalSocketServer::fWait_设置为 false，这将导致 PhysicalSocketServer 退出侦听循环重新将控制权交给 MessageQueue。MessageQueue 获得控制权后将立即处理消息，在完成消息处理后再将控制权交给 PhysicalSocketServer。

传给 Wait 的 cmsWait 参数是 0 时，它实现了轮询事件。即当前有事件时就处理，没有会立即返回。SDLThread 就用了这个功能。

Wait 中的参数 process_io 指示了此次是否要处理 IO 事件，true 时要处理。不管 process_io 是何值，一定会侦听唤醒事件。

一个事件如何同时侦听多个 socket

SocketServer 可以同时侦听多个 socket，可它们在 WSAWaitForMultipleEvents 只占一个事件的位置，这是怎么回事？

```
int WSAEventSelect(
    _In_ SOCKET s,      // desire associated SOCKET
    _In_ WSAEVENT hEventObject,  // desire assoiated event
    _In_  long lNetworkEvents    // interested even, FD_ALL_EVENTS indicate all
event
);
```

socket 事件和 WSAWaitForMultipleEvents 事件不是一回事。以上代码中，参数 hEventObject 是 WSAWaitForMultipleEvents 事件，即那个占了位置的事件，而 s 是我们想侦听的 socket，要同时侦听多少个 socket 就要调用多少次 WSAEventSelect。接下来，一旦这些 socket 上发生事件都会触发到这个 hEventObject。

既然这些 socket 触发的都是位置 0，代码怎么知道是哪个 socket 触发？WSAWaitForMultipleEvents 执行结束后，发现是位置 0 触发，就会调用 WSAEnumNetworkEvents 去判断触发的是哪个 socket。

在 app 使用多路事件分离器

多路事件分离器中有个 Thread，实际使用时这个 Thread 往往就是主线程。也就是说，当 socket 收到数据后，处理数据都放在了主线程，这有助于实现单线程 app，避免潜在的同步问题。

app 使用多路事件分离器分两步，一是定义个从 Thread 派生的类，实现轮询函数；二是把这个类挂向主线程，app 轮询时调用它的轮询函数。

步骤一：定义个从 Thread 派生的类，实现轮询函数

```
namespace rtc {
class SDLThread : public Thread {
public:
    SDLThread() : ss_() { set_socketserver(&ss_); }
    virtual ~SDLThread() { set_socketserver(NULL); }
    void pump();
private:
    PhysicalSocketServer ss_;
};

void SDLThread::pump()
{
    Message msg;
    size_t max_msgs = std::max<size_t>(1, size());
    for (; max_msgs > 0 && Get(&msg, 0, true); --max_msgs) {
        Dispatch(&msg);
    }
}
}
```

SDLThread::pump()是轮询函数。逻辑很简单，以 cmsWait=0、process_io=true 去调用 Get。此刻有 IO 事件时，Get 就给处理了，有唤醒事件时，Get 会返回 TRUE，并把事件相关的消息数据填入 msg，轮询函数就用这 msg 调用 Dispatch，处理这个唤醒事件。

步骤二：把 SDLThread 挂向主线程。app 轮询时调用它的 pump 函数。

```
rtc::SDLThread sdl_thread_;
rtc::ThreadManager::Instance()->SetCurrentThread(&sdl_thread_);

void pump()
{
  ......
  sdl_thread_.pump();
  ......
}
```

在 app 主线程入口处执行 SetCurrentThread(&sdl_thread_)，它把 sdl_thread_挂向当前线程，即主线程。

假设 pump()是 app 的轮询函数，在那调用 sdl_thread_.pump()就可以了。

3.9.2 网络 API

一、初始化 socket

```
rtc::AsyncSocket* socket_;
socket_ = sdl_thread_.socketserver()->CreateAsyncSocket(family, SOCK_STREAM);
socket_->Connect(server_address_);
```

初始化分两步，一是创建，二是连接 ip:port 对。创建 socket 就是调用 PhysicalSocketServer 的 CreateAsyncSocket。CreateAsyncSocket 须要两个参数，分别对应创建 socket 时须要的 af、type。af 可选 ipv4(AF_INET)或 ipv6(AF_INET6)，type 可选 tcp(SOCK_STREAM)还是 udp(SOCK_DGRAM)。

CreateAsyncSocket 会调用 SocketDispatcher::Initialize()，后者会执行 ss_->Add(this)。ss_是要加入到的 PhysicalSocketServer，this 就是这个 socket，Add 会把 socket 加入到事件分离器的

IO 事件集合。ss_所在的 MessageQueue 就是处理该 socket 事件的线程。

Connect 执行连接 ip:port 对。

二、读写 socket

读写就是向 socket 挂接相应的处理函数，然后编写这些函数。

```
socket_->SignalCloseEvent.connect(this, &tchat2::OnClose);
socket_->SignalConnectEvent.connect(this, &tchat2::OnConnect);
socket_->SignalReadEvent.connect(this, &tchat2::OnRead);
```

socket 异常断开（像禁用网卡）会调用 OnClose，连接成功会调用 OnConnect，收到数据则是 OnRead。这些函数的执行线程就是创建 socket_时的 MessageQueue。

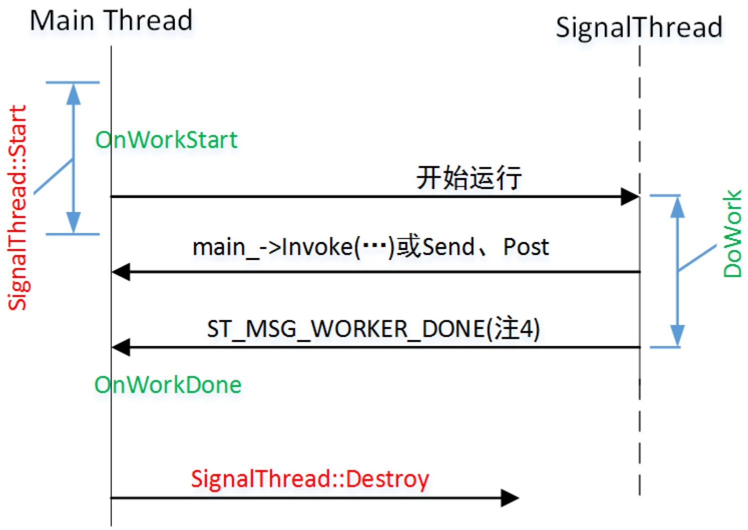
三、关闭 socket

```
socket_->Close();
```

如果 socket_正连接着，Close 会先执行断开。接下会从事件分离器的 IO 事件集合移除该 socket_。

3.9.3 使用 SignalThread 实现双线程协作

用这小节说的双线程协作去要实现的任务中，不包括网络。网络操作虽然也基于 Webrtc，但会有专门方法，见底下的“3.10 网络”。这里双线程目的一般是实现后台任务，像 Studio 在构造 bin 时就用了 SignalThread 机制。



图中绿色是 SignalThread 派生类可重载的函数，所在位置表示了执行时所在的线程。其中 DoWork 是纯虚函数，必须重载。

派生类调用 SignalThread::Start 启动任务。结束前必须要调用过 SignalThread::Destroy，如果工作者线程还在运行，Destroy 会先停止线程。Destroy 不仅仅释放工作者线程，还会析构掉派生类对象，自然也包括基类 SignalThread。也就是说，不能在派生类的析构函数中调用 Destroy。

工作者线程运行期间，SignalThread 没有定义双方通信有什么流程，通信全由双方自个定义。

工作者线程执行完 DoWork 后，向主线程投递 ST_MSG_WORKER_DONE 消息。主线程处理该消息，行为是调用 OnWorkDone，OnWorkDone 执行期间，工作者线程可能还是在运行的，只是已经结束了 DoWork 这个任务。

工作者线程对象是 SignalThread 被构造时就被创建，只是要到 SingalThread::Start 才开始运行线程。执行完 DoWork 后，很快会结束执行，后面可调用 Start 再次执行。一直要等到 SignalThread 析构时才会被销毁。

机制优点。工作者线程运行期间，可发挥 Webrtc 线程模型中强大的同步功能，Send、Post，用得最多的应该是 main_->Invoke。机制会确保被调用的函数是在主线程执行，而且在这执行过程中工作者线程是被阻塞。这让同步变得既清晰又高效，基本可告别同步变量。

机制缺点。析构 SingalThread 不能按通常的析构逻辑，必须通过 Destroy！由于有违常规，一不小心会造成编程时犯错，导致资源泄漏甚至程序非法。更重要的，它会使编程变复杂。app 往往希望用一个类封装此次后台任务，由于不能通过正常析构，类就不能从 SingalThread 派生，而不得不把 SingalThread 作为类中成员变量，这使很多操作变繁锁了，像访问变量。为弥补这缺陷，Rose 提供 tworker、rtc::worker_thread，以下是它们完整代码。

```
namespace rtc {
class worker_thread: public SignalThread {
public:
    worker_thread(tworker& worker)
        : worker_(worker)
    {}
protected:
    void DoWork() override { worker_.DoWork(); }
    void OnWorkStart() override { worker_.OnWorkStart(); }
    void OnWorkDone() override { worker_.OnWorkDone(); }
protected:
    tworker& worker_;
};
}
class tworker {
public:
    virtual void DoWork() = 0;
    virtual void OnWorkStart() = 0;
    virtual void OnWorkDone() = 0;
protected:
    tworker()
        : main_(rtc::Thread::Current())
        , thread_(new rtc::worker_thread(*this))
    {}
    virtual ~tworker() {
        thread_->Destroy(true);
        thread_ = NULL;
    }
protected:
    rtc::Thread* main_;
    rtc::worker_thread* thread_;
};
```

有了 tworker，app 编程大致可按以下流程。

1. 定义个从 tworker 派生的类 A。类实现那三个纯虚函数。
2. 要启动后台线程了，调用 thread_->Start()。
3. 工作者线程运行期间，可使用 Send、Post 向主线程发消息，更普遍的应该是 Invoke。Invoke 可直接调 A 中函数，此函数运行在主线程，而且在这执行过程中工作者线程是被阻塞。
4. DoWork 一次执行完毕后，如果主线程想再次运行任务，只须再次调用 thread_->Start()。
5. A 以着正常逻辑被析构，一旦析构，和此次任务相关的 SignalThread 资源一并被释放。

3.9.4 twebrtc_send_helper

主线程要销毁子线程时会遇到这么种情况，1）主线程进入销毁子线程 A 代码，即在销毁 A 前不会调用 sdl_pump。2）A 向主线程发 Send。——由于主线程不调用 sdl_pmup，A 在等主线程执行 Send 中的操作，主线程则锁在等待 A 销毁，导致死锁。

main thread

```
twebrtc_send_helper send_helper_(this);

要销毁子线程A的函数()

send_helper_.clear_msg();
已能确定子线程A不会有悬挂着、向主线程
的Send，以及接下不会再向主线程发Send
```

```
twebrtc_send_helper::clear_msg()

deconstructed_ = true;
while (senders_ > 0) {
  instance->sdl_thread().clear_msg(phandler_);
  SDL_Delay(20);
}
```

子线程A

```
向主线程触发Send的函数()

twebrtc_send_helper::tlock lock(send_helper_);
if (lock.can_send()) {
  // 此时可以发送，向主线程发Send
  instance->sdl_thread().Send(RTC_FROM_HERE, this, ...);
}
```

```
twebrtc_send_helper::tlock::tlock(helper)
  : helper_(helper)
  , th_can_(!helper.deconstructed_) {
  if (th_can_) helper.senders_ ++;
}

twebrtc_send_helper::tlock::~tlock() {
  if (th_can_) helper.senders_ --;
}

twebrtc_send_helper::tlock::can_send() {
  return th_can_ && !helper_.deconstructed_;
}
```

为解决这问题，Rose 提供了 twebrtc_send_helper， 图中蓝色是 app 如何它的方法。1）主线程调用 send_helper_.clear_msg 后，能保证子线程 A 不会有悬挂着、发向主线程的 Send，以及接下不会向主线程发 Send。2）构造 twebrtc_send_helper 和执行 clear_msg 必须在主线程。3）构造 tlock 必须运行在子线程，构造出 tlock 后，若发现 can_send 是 false，不要有 Send 操作。

tlock::can_send()为何要额外考虑此时的 helper_.deconstructed_？th_can_是 true，表示的是 th_can_被赋值那一刻 helper_.deconstructed_是 false，到 can_send()时，主线程可能在这期间调用 clear_msg。

谁持有 twebrtc_send_helper？属主没有继承上要求，重点是构造 helpler 得填对参数（phandler_），主线程 clear_msg 删除的是子线程 Send 到 phadler_的函数。

3.9.5 Post+app_OnMessage 机制

处理控件事件时，Rose 允许在 LEFT_BUTTON_CLICK、LEFT_BUTTON_DOUBLE_CLICK、RIGHT_BUTTON_UP 弹出新窗口，其它事件要弹出新窗口的，须使用 Post+app_OnMessage 机制。使用 Post+app_OnMessage 分三个步骤。

步骤一：定义消息码

```
enum {MSG_BUILD_FINISHED = POST_MSG_MIN_APP};
```

Rose 内部已经有地方在使用 Post 处理事件，像处理松开拖拽时的 MOUSE_LEAVE，POST_MSG_MIN_APP 是 app 可定义的最小消息码。

步骤二：在须要的地方投递这个消息码

```
rtc::Thread::Current()->Post(RTC_FROM_HERE, this, MSG_BUILD_FINISHED, NULL);
```

总是主线程处理这个投递消息，执行 Post 的线程已是主线程 ，那 rtc::Thread::Current()返回的就是主线程指针。当然，如果你的 tdialog 从 tworker 派生，成员变量 main_就是主线程。

步骤三：实现 app_OnMessage(rtc::Message* msg)

```
void trose::app_OnMessage(rtc::Message* msg)
{
    switch (msg->message_id) {
```

```
    case MSG_BUILD_FINISHED:
        ......
    }
}
```

app_OnMessage 在主线程的 instance->pump()中被执行，参考“**3.2.4 events::pump()**”。

3.9.6 多线程 API

到现在，已介绍了两种场合下的多线程解决方案，一是网络收发，二是后台任务。网络收发时，基于 Webrtc 的多路事件分离器把收发处理放去主线程，后台任务则基于 SingalThread 去实现。对一般 app，它们可能就已经囊括所有多线程场合，但是，如果有 app 想自个创建新线程，那以下代码演示了 1）如何增加线程、2）如何处理新消息。

```
#include <string>
#include <iostream>
#include "webrtc/base/thread.h"

class HelpData : public rtc::MessageData {
public:
    std::string info_;
};

class Police : public rtc::MessageHandler {
public:
    enum {MSG_HELP};
    void Help(rtc::Thread& targetThread, const std::string& info) {
        HelpData* data = new HelpData;
        data->info_ = info;
        targetThread.Post(RTC_FROM_HERE, this, MSG_HELP, data);
    }
    void OnMessage(rtc::Message* msg) override {
        switch (msg->message_id) {
            case MSG_HELP:
                HelpData* data = (HelpData*)msg->pdata;
                std::cout << "MSG_HELP: " << data->info_ << std::endl;
            break;
        }
    }
};

int main(int argc, char** argv) {
    std::cout << "Test Multi-thread is started" << std::endl;
    Police p;
    rtc::Thread thread;
    thread.Start();
    p.Help(thread, "Please help me!");
    std::cout << "Test Multi-thread is completed" << std::endl;
    return 0;
}
```

代码中的 HelpData、Police 有什么用？——但凡要处理消息，须要解决三个问题。

问题	数据类型	对应到示例
消息在哪个线程执行	rtc::Thread 对象	rtc::Thread thread，即调用 Post 的 targetThread
处理消息的是谁	必须从 rtc::MessageHandler 派生	rtc::MessageHandler 的派生类 Police，Post 调用中的 this
能被执行的消息有什么要求	消息须从 rtc::MessageData 派生	rtc::MessageData 派生类 HelpData，Post 调用中的 data

示例中用的是 Post，除了它，多线程通信还有 Send 和 Invoke。

3.10 网络

虽然 Webrtc 提供了结构和同步都简洁的网络 API，但还是要问能不能更简单？想到两点，一是基于协议编程，二是和界面集成。于是基于 Webrtc 再提供网络模块。要注意，这并不是要求 app 一定要用它编程，有时直接用 Webrtc 可能更简单。

3.10.1 基于协议编程

Webrtc 提供的网络 API 是基于 socket，但 app 在网络编程时和协议相关，像 http，irc，或自定义协议。app 更希望把精力花在解析协议和处理收到的数据，不想重复写那些基础的机械式代码，像创建 socket、挂接处理函数，等等。

tlobby 称为大厅，它负责管理网络框架，整系统只存在一个 tlobby 对象。tsock 往往对应一种网络协议，Rose 默认提供了 http、irc。任一时刻 tsock 处在四种状态中一种。

s_none→已创建。socket 执行初始化后就进入已创建状态。

已创建→协商。协商还是就绪由协议自个定。协商往往指握手过程，像 irc 中的登录。至于 http 可能就没协商，直接就入就绪。

协商→就绪。协议的主要执行过程。

就绪→none。意外、或强制断开。

tlobby 要基于 Webrtc，初始化需在 SDLThread 挂接向线程后，销毁则须要在 SDLThread 被删除之前。运行时有个自轮询操作(pump)，系统轮询时(events::pump)会调用这个轮询。

3.10.2 和界面集成

app 有时会遇到要求"阻塞"式应答。举个例子，要登录后台才能继续操作，登录过程是向服务器发请求、接收应答，那在发出请求后，app 进入"阻塞"，一直到收到应答。这时如果网络拥堵，等待应答时最好向用户显示个"通信中"提示，甚至可以让用户取消此次登录。和界面集成指的就是在网络过程中嵌入以上"通信中"提示、取消机制。

```
bool run_with_progress(const std::string& message, const boost::function<bool
(tprogress_&)>& did_first_drawn, bool quantify, int hidden_ms, bool show_cancel)
```

run_with_progress 是 Rose 提供的用于处理需要进度条操作的 API。它可用于执行需要点时间的阻塞式操作，也可用于处理和网络相关的“阻塞”式应答。

run_with_progress 首先会创建带有进度条的对话框，然后 dlg.show。在 show 过程中，执行完第一次 twindow::draw 后会调用 did_first_drawn，系统控制权就交给了 did_first_drawn，也就是说，除非 did_first_drawn 执行结束，控制权不会再交到进度条窗口的 events::pump。当 did_first_drawn 结束，dlg.show 也就结束，同时把 did_first_drawn 返回值做为它的返回值。

进度条窗口的 events::pump 已不会被执行，那 did_first_drawn 是如何刷新进度条?这分两种场景，quantify 是 true 时，表示进度条要定量显示，像 n%递增，刷新方法是通过调用 tprogress_的 set_message/percentage 方法。quantify 是 false 时，进度条只是定性显示，表示操作中，无法通过它知道执行进度，此时的刷新方法是通过在 did_first_drawn 中的某些地方显示调用 twindow::show_slice，后者有 events::pump，在那里执行刷新。

hidden_ms 指示进度条窗口一开始要隐藏的毫秒数。为什么要有这参数?app 要执行网络操作，如果网络通畅，很多在 2、3 秒内就能执行完，用户对这点时间可能是“无察觉”的，但如果一有网络操作就显示进度条，然后很快消失，那界面就不美观了。网络操作超过 hidden_ms 还没完成，意味着发生意外情况，像网络拥堵，就有必要向用户显示进度条，表示正在操作中。

hidden_ms 非 0 时，进度条窗口由隐藏到非隐藏靠的是 events::pump，当 quantify 是 true 时，因为它不执行 events::pump，hidden_ms 必须是 0。

一般情况下，非网络操作，以及网络操作中要上传、下载较多数据的设置 quantify=true、hidden_ms=0。其它的网络操作则把 quantify 设为 false，hidden_ms 不为 0。

3.11 文件和目录

文件 API 分三个部分，操作文件、读写文件和批处理文件。当目录做为参数时，不要加最后的那个目录分隔符。所有文件、目录名用 UTF-8 编码。

3.11.1 操作文件

```
SDL_bool SDL_MakeDirectory(const char* dir);
SDL_bool SDL_GetStat(const char* name, SDL_dirent* stat);
SDL_bool SDL_DeleteFiles(const char* name);
SDL_bool SDL_CopyFiles(const char* src, const char* dst);
SDL_bool SDL_RenameFile(const char* oldname, const char* newname);
typedef boost::function<bool (const std::string& dir, const SDL_dirent2* dirent)> twalk_dir_function;
bool    walk_dir(const    std::string&    rootdir,    bool    subfolders,    const twalk_dir_function& fn);
std::string utils::normalize_path(const std::string& src, bool slash = true);
```

SDL_GetStat 获取文件、目录信息，等同 linux 下的 stat 函数。信息包括是目录还是文件，长度、修改时间、创建时间和访问时间。SDL_DeleteFiles、SDL_CopyFiles、SDL_RenameFile 都既可操作文件也可是目录，更多文件 API 信息见网上帮助“文件 API”。

normalize_path 用于规范化文件、目录名，会执行四个方面的规范化。1）统一当中会出现的分隔符，参数 slash 是 true 时统一用“/”，否则用“\”。2）处理“..”。3）处理“.”。4）处理连续两个分隔符。

3.11.2 读写文件

```
posix_fopen(name, desired_access, create_disposition, file)
posix_fseek(file, offset)
posix_fwrite(file, ptr, size)
posix_fread(file, ptr, size)
posix_fclose(file)
posix_fsize(file)
```

它们被定义成宏，基本就是直接调用操作系统 API。它们都支持 64 位，表示文件长度建议用 int64_t。posix_fseek 用于定位，放弃了传统 C 可选的三种定位点，而是固定从文件头开始(SEEK_SET)。因为直接调用 API，这几个宏的执行效率是高，但使用时会遇到些麻烦，像忘了 close 导致内储泄露，为读写数据要专门申请、释放个缓存，为此提供一个叫 tfile 的类。

设计 tfile 遵循一个原则：简洁、再简洁。因而它是封装“posix_”，但不封装读写，读写还是要求去直接调用 posix_read、posix_write。tfile 主要解决三个问题，一是简化打开、关闭编程，二是提供一个缓存，三是支持截短文件。

tfile 把读写文件的使用场景简化为三类，构造函数中的访问标记、创建标记决定了要使用的是哪一类。

访问标记	创建标记	使用场景	截短
GENERIC_READ	OPEN_EXISTING	只想读文件。文件不存在就失败	不支持
GENERIC_WRITE	OPEN_EXISTING	想读写已存在文件。文件不存在就失败	支持
GENERIC_WRITE	CREATE_ALWAYS	想读写文件，文件如果已存在，会强制先清空	不支持

截短只是截短。为什么要强调这个？当设置比文件已有长度更大的字节数时，一些操作系统的截短 api 能让文件预留出这个字节数，换句话说，让增大文件。但 tfile 只支持截短，tfile 完成截短分两步。

1）app 调用 tfile::truncate 设置文件最后希望的长度 truncate_size_。

2）tfile::close 被调用时，如果文件当前长度大于 truncate_size_，而且是以“GENERIC_WRITE + OPEN_EXISTING”打开，执行截短。

3.11.3 批处理文件

批处理文件指的是一次性处理多个目录/文件。很明显，为方便维护不能把要处理的目录/文件写死在 C/C++代码，须借助配置文件告知要执行的是什么处理，以及要处理哪些目录/文件。

具体到一个文件、目录，内置支持的处理类型包括创建、复制、删除、重命名，以及替换文件中字符串。同时支持 app 按自个要求增加处理类型。Studio 处理工作包、app 包时大量使用了批处理，通过看那些代码可熟悉相关编程。

批处理编程总的来说分三个步骤。1）编写批处理配置。2）编写一个从 ttask 派生的类，这类目的是在一些细节上做必要补充。假设这个类是 tremover。3）调用 tremover 执行具体批处理。编程细节参考网上文档“文件 API”中的“批处理文件”。

3.12 主题

3.12.1 切换主题

主题用一个字符串 id 标识，为支持在开发时不同 app 可使用相同 id，标识由两部分构成，一是 app 标识，二是短标识。像“studio__night”，studio 是 app 标识，night 是短标识。下面代码把“studio__night”设为当主前主题。

```
void base_instance::theme_switch_to(const std::string& app, const std::string& id);

theme_switch_to("studio", "night");
```

app 或许有这么个需要，它只用一个主题，但不是 rose 默认的，希望每次运行都强制使用它，要实现这目的可重载 app_load_app_config，让增加以下语句。

```
preferences::set_theme(utils::generate_app_prefix_id(game_config::app, "night");
```

后续代码加载 data.bin 后，会调用 theme_switch_to 把主题设为 preferences 中的 theme 值。

3.12.2 资源

增加一个主题只需在资源包中增加资源，不涉及任何 C/C++源码。Studio 有提供增加、删除、修改主题命令。

一个新主题由两部分构成，一是脚本，二是图像。新增一个主题也就是包括这两部分。

脚本

脚本只一个 cfg 文件，存放着两种信息，一是主题标识，二是颜色，它们放在一个叫“theme”的块。颜色分为两类，一是文字颜色，二是三个特征色，分别是获得 focus 的 item_focus_color，表示高亮的 item_highlight_color，以及专门用在弹出式菜单项的 menu_focus_color。

文字颜色有三种，通常时 normal_color、灰掉时 disable_color，获得焦点时的 focus_color。这三种颜色组成一套模板，然后存在三套模块，normal、inverse 和 title。

```
[theme]
    id = night
    app = studio
    .....
[/theme]
```

把这个 cfg 放在<res>/data/app-studio/theme，然后在<res>/data/app-studio/_main.cfg 确保有以下的 WML 扩展。

```
{app-studio/theme}
```

图像

在<res>/app-studio/images，涉及到的需要修改的图像增加一个以短标识“id”命名的目录。举个例子，要重载 misc/build.png，需要存在这样路径的文件。

```
<res>/app-studio/images/misc/night/build.png
```

3.13 调试信息、诊断

一些情况需要使用断点外的调试方法，像多线程，这里分两种，输出到控制台和弹出提示对话框。

输出到控制台。Rose 建议使用 SDL 提供的 api 向控制台输出信息，具体是 SDL_Log。为什么说建议，SDL 提供了多个输出到控制台的 api，SDL_Log 只是当中一个。SDL 把调试分为数个级别，SDL_Log 对应的是 SDL_LOG_PRIORITY_INFO，这个级别在 Android 对应的是 ANDROID_LOG_INFO。

弹出提示对话框。app 调用 SDL_SimplerMB 可弹出个模态的提示对话框。SDL_SimplerMB 不是 SDL 提供的 api，是基于 SDL_ShowSimpleMessageBox 封装出的一个函数。它只是供调试用，不在乎界面，app 在正常运行时不要让调用它。

app 可在 main 的一开始就调试 SDL_Log、SDL_SimplerMB，不必等到初始化 SDL 库后。SDL_log.h 声明了 SDL_Log，不过 SDL.h 已包含了 SDL_log.h。global.hpp 声明了 SDL_SimplerMB，cpp 都会包含这个头文件。

诊断指的是 assert，Rose 为避免重名，使用了一个叫 VALIDATE 宏。在 Deubg 和 Release，VALIDATE 有着一样的行为。

3.14 app 后台运行（iOS、Android）

参考“app 后台运行（iOS、Android）”。

Android 下，Home 键导致 app 进入后台、待机导致 app 进入后台，这两种情况的不同是 Home 多了调用 SDLSurface::surfaceDestroyed，于是就在它调用的 onNativeSurfaceDestroyed 中发送出 SDL_WINDOWEVENT_MINIMIZED。意思是 app 被最小化了，同时表示不是前台 app 了。

第四章 WML 和 Lua

本章目标

- 程序处理配置数据逻辑。
- 处理 WML 中的可翻译字符串。
- WML 如何处理宏。
- C 和 Lua 相互调用过程。

不论资源包还是用户数据都存在很多配置。对如何保存配置，Windows 会想到 ini、注册表，Linux 则会让想到 etc 目录，而 iOS、Andorid 则使用配置和二进制代码文件打包成一个 ipa/apk 包。Rose 把配置组织成数个目录，一目录下有多个配置文件，配置文件中以 WML 和 Lua 语法描述每个配置项。WML 描述静态配置，Lua 则描述动态配置。

为什么不使用 Json、xml？——希望以一种“最”快速度、“最”省内存方式处理配置，Json、xml 还是太复杂了，比不上只有块、属性的 WML。当然，因为语法少，WML 的功能不如 Json、xml，但这里把复杂的语法放在了 Lua，像条件判断、循环。

一个文件要被认为是 WML 语法的配置文件，那它的扩展名必须是 cfg。

4.1 WML 和 class config

WML 是编写 cfg 文件使用的配置格式，config 是程序员认为的配置格式，它基本就是 WML 的 C/C++翻译。

4.1.1 WML

WML 是“Wesnoth Markup Language”的缩写，发源于开源项目“韦诺之战”。一个文件要被认为是 WML 语法的配置文件，那它的扩展名必须是 cfg。类似 ini、xml、json，cfg 是种简单的文本文件，而且要没特殊要求，不要让出现非 ASCII 字符。WML 的基本语法就是属性和块。

```
# remark1
[event]
    name=prestart # remark2
    [objectives]
        [objective]
            description= _"Defeat all sides"
            condition=win
        [/objective]
        [objective]
            description= _"No city you holded"
            condition=lose
        [/objective]
    [/objectives]
[/event]
```

以上是一个 WML 例子。[event]是一个块，包含 name 这个属性，也包含一个叫 objectives 的子块。objectives 则没有属性，但包含两个叫 objective 的子块。objective 块下有两个属性：description 和 condition。

一行中，“# ”（后面有一个空格）开始的部分是注释。为什么后面要多一个空格？像“#textdomain”、“#define”是语法意义的。

4.1.2 class config

config 是一个 C++类，基本就是 WML 的 C/C++翻译。

```
class config
{
    static config invalid;
    explicit operator bool() const { return this != &invalid; }
```

```
    std::map<std::string, attribute_value> values;
    std::map<std::string, std::vector<config*> > children;
    std::vector<child_pos> ordered_children;
    ……
}
```

values 存储着属性。map 中的 first 对应属性 key，attribute_value 表示属性值。attribute_value 实现了用以下代码读写属性。

```
// read attribute
const std::string desc = cfg[“description”].str();

// write attribute
cfg[“score”] = 7.2;
```

str()对应的是读 std::string 类型，其它还有 to_bool（true、false）、to_int（整型）、to_long_long（long long）、to_unsigned（unsigned）、to_size_t（size_t）、to_time_t（time_t）、to_double（double）和 t_str（t_string）。对写，attribute_value 会根据右侧值自动判断出要存储到的类型，例子“cfg[“scroe”] = 7.2”会把“score”自动存储为 double。

children、ordered_children 用于表示子块，每个变量都能表示全部子块，只是为不同操作需要用了两种存储形式。children 按块名对子块进行分类存储，size 表示了有多少种子块。ordered_children 则按书写序存储每个子块，size 就是块中子块数。

示例中特别指出“invalid”，它其实是一个空 config，但 config 内部用它表示无效 config。它怎么用，让看以下代码。

```
const config& cfg = core_cfg.child(“app_config”);

if (cfg) { … } // it will execute “operator bool()”
```

core_cfg.child(“app_config”)查找 core_cfg 中是有叫“app_config”的块，当没有这块时，child 返回此个“invalid”，也就是说，存储查询结果的变量 cfg 值就是“invalid”。于是对接下的“if (cfg)”，判断 cfg 是否是有效块将触发“operator bool()”，这重载函数发现 this 等于 invalid，返回 false。“invalid”只在 config 内部使用，app 不要用它，想表示一个空 config 可用 null_cfg。

为方便各种使用场合，config 提供了很多操作，例如 merge_with。

```
void config::merge_with(const config&c)
```

merge_with 把 c 这整个 config 合并到 this，既合并属性也合并块。对属性。this 没有的，增加；有的，替换掉。对块。块名 this 没有的，增加；有的，出现冲突，因为块除了名称还有位置。假设出现冲突的是叫 tag 的块，this 有 tag 块 N 个，c 中有 tag 块 M 个。**N>=M：**M 个 tag 块按索引位置替换。**N<M：**N 个 tag 块按索引位置替换。(M-N)个 tag 块被新加入 this。

换句话说，不论是属性还是块，merge_with 执行的都是 this“没有”的，增加；“有”的，替换掉。只是对属性还是块，这个“有”的概念不一样，对属性，有相同属性名就是“有”；对块，既要有相同块名还有要相同在此种块中位置。

4.2 WML 语法

4.2.1 WML 扩展

如何形成一个块？一种方法是把它在一个文件完整写出来。全要这么做的话，一来不灵活，二来有时就无法做到。这时就可借用 WML 扩展机制去实现把分散在多处的内容形成大块。扩展可归为两类：文件内扩展和目录扩展。

方式一：文件内扩展

文件内扩展指的是在一个 cfg 文件内使用扩展语法让手动实现扩展。文件内扩展是解析到那条语句时执行扩展。

```
[gui]
    {gui/default/macros}
```

```
    {gui/default/window}
    {gui/app.cfg}
[/gui]
```

示例使用了三条扩展，执行“{gui/app.cfg}”后，app.cfg 定义的属性、块就加入[gui]。。扩展语法是“{}”，中间写要嵌入的 WML 路径。“**4.2.2 宏**”会说到“{}”是宏使用语句，解释器没找到和“{}”中字符串相同的宏名时，就认为要执行 WML 扩展。扩展时，“{}”中写的是路径后半部分，前半部分遵循以下规则。

{}中字符串	前半部分	{gui/app.cfg}示例
第一个字符是“~”	<user>/data	<user>/data/gui/app.cfg
第一个字符是“^”	<res>	<res>/gui/app.cfg
第一个字符既不是“~”也不是“^”	<res>/data	<res>/data/gui/app.cfg

完整路径和中间字符串的第一个字符有关，和 cfg 文件所有目录无关。中间字符串是文件也可以是目录，对目录，建议最后一个字符不要是目录分隔符“/”（将来会不支持）。是目录时，要执行下面要说的目录扩展。

方式二：目录扩展

目录扩展是使用创建目录方法让自动实现扩展。它遵循以下两条规则。

1、查找该目录下是否有_main.cfg，如果有，解析该_main.cfg，并结束解析。否则进入 2。

2、依次处理该目录下的所有子目录、文件（会按字符串排序，但会出现文件、“子目录”相互混杂）。枚举到的是文件时，解析该文件。枚举到的是子目录时，如果该目录下有_main.cfg，则解析该_main.cfg，如果没有则忽略此目录。

目录扩展最多检查到该目录下面的一级子目录，而且是该子目录下必须有_main.cfg。

根据以上规则，如果准备写_main.cfg，那就要由_main.cfg 去扩展此目录下所有文件和子目录。_main.cfg 缺点是一旦增加一个 cfg，那就必须在当中加上一条和此 cfg 文件对应的文件内扩展。没了_main.cfg，目录扩展时就会自动解析该目录下所有根文件。

除了_main.cfg，目录扩展可使用另外一个特殊文件：_initial.cfg。如果目录下没有_main.cfg，那么目录扩展会保证首先解析_initial.cfg。

4.2.2 宏

宏用于简化书写配置。WML 中的宏类似 C/C++中的宏，它必须在使用前被定义。宏在解析器预处理时被展开。

定义

```
#define 宏名 [参数]
宏内容
#enddef
```

实例 1

```
#define MAX_TEXT_WIDTH
675
#enddef
```

定义一个名叫 MAX_TEXT_WIDTH 的宏，它的内容就是一个整数值。这在 C/C++中等同#define MAX_TEXT_WIDTH 675。

实例 2

```
#define _GUI_H_SPACER WIDTH GROW_FACTOR
[column]
    grow_factor = "{GROW_FACTOR}"
    [spacer]
        definition = "default"
        width = "{WIDTH}"
    [/spacer]
[/column]
#enddef
```

定义一个名叫_GUI_H_SPACER 的宏，它带两个参数：WIDTH、GROW_FACTOR。宏中使用参数方法是在采用大括号。

使用

```
{宏名 [参数]}
```

使用实例 1 中 MAX_TEXT_WIDTH 例子：

```
[spacer]
    width = "{MAX_TEXT_WIDTH}"
[/spacer]
```

使用实例 2 中_GUI_H_SPACER 例子。

```
{_GUI_H_SPACER 4 1}
```

删除

```
#undef 宏名
```

删除实例 1 的 MAX_TEXT_WIDTH 宏。

```
#undef MAX_TEXT_WIDTH
```

删除实例 2 的_GUI_H_SPACER 宏

```
#undef _GUI_H_SPACER
```

C/C++中没有删除宏这个操作。WML 存在删除宏主要原因是为减少 WML 解析器需要内存、加快 WML 解析速度。可以根据以下要叙述的 copy_map 进一步理解为什么要存在删除宏。

4.2.3 要翻译字符串、t_string

要翻译字符串指的是根据当前语言不同会显示出不一样结果的字符串。代码 3-1 中的_"Defeat all sides"，当前选择是英文时，它会显示“图 3-12 race of hero”，而当前是中文时，它则显示图 3-3 中的“击败所有势力”。

，

图 3-12 race of hero

WML 指定一个字符串是要翻译字符串的方法就是在字符串前加个“_”字符。要翻译字符串只会出现在属性值部分。一个属性值中可以存在多个要翻译字符串。

不同语言下，要翻译字符串是如何实现显示不同语言字符串的？第八章“国际化和 gettext”会详细介绍具体实现，这里只要知道<kingdom-src>/po 目录为每种语言有一组*.po 文件。作为命名习惯，存放英文是 en_GB.po，简体中文是 zh_CN.po，繁体中文则是 zh_TW.po。

po 文件内容就是<key, value>这样的一个映射集合，像 zh_CN.po 中有<Defeat all sides, 击败所有势力>，en_GB.po 中有<Defeat all sides, Defeat all sides>，要翻译时程序知道关键字 key，它就根据当前选择语言定位到特定 po，搜出相应 value。

为什么“每种语言有一组*.po 文件”？这是因为程序把映射项按一定主观规则分开存放在多个文件无疑是有好处的。一来可以维护上模块化，二来程序执行时加快翻译（在那一文件中映射项少，搜索快了）。游戏中全局信息，像“坐标”、“点击鼠标”这样一些操作信息是放在 wesnoth.po 文件中，而像兵种信息放在 wesnoth-units.po，像“骑兵”、“市场”，而“Defeat all sides”则是放在了 wesnoth-race.po 中。

既然存在多个文件，程序在翻译具体某一字符串时就要有“当前文件”概念。WML 中指定

“当前文件”是使用“#textdomain”指令（此处“#”不是注释符)。具体到代码 3-1 中的两个可翻译字符串，01_hero_race.cfg 没有“#textdomain”，这指令是在它的上级<kingdom-src>\data\campaigns\Hero_Race_main.cfg，那里用了#textdomain wesnoth-race，即告诉 WML 解析器，解析到这里时把 wesnoth-race.po 设为当前文件。

“文件”在 gettext 中有对应的专门术语“域”(domain)，“当前文件”往往就称为“当前域”。

代码中实现切换当前域

打开配置文件<kingdom-src>\data\gui\default\window\hero_list.cfg，找到这段代码。

```
#define _GUI_TABLE_HEADER_ADAPTABILITY
    [row]
        [column]
            [label]
                definition = "table"
                label = _ "Name"
                linked_group = "name"
            [/label]
        [/column]

#textdomain wesnoth-hero
        [column]
            grow_factor = 1
            horizontal_grow = "true"
            [label]
                definition = "table"
                label = _ "arms-0"
                linked_group = "arm0"
            [/label]
        [/column]
        ......

#textdomain wesnoth-lib
    [/row]
#enddef
```

hero_list.cfg 的第一条指令#textdomain wesnoth-lib 把 wesnoth-lib 设为当前域，但“arms-0”这个可翻译字符串是写在 wesnoth-hero.po，这时就要把 wesnoth-hero 设为当前域。此处用#textdomain wesnoth-hero 把 wesnoth-hero 切换为当前域，当接下字符串又是写在 wesnoth-lib.po 时，再用#textdomain wesnoth-lib 把当前域切换回 wesnoth-lib。

程序如何表现出这个切换，让进入代码级调试。

1. 在 kingdom 工程中打开<src>/xwml.cpp。
2. wml_config_from_data 函数内增加代码，增加点见图 3-13。
```
if (!strcasecmp((char*)valbuf, "arms-0")) {
    int ii = 0;
}
```
3. 重编译 kingdom.exe。
4. 把断点设在“int ii = 0”。
5. 运行时选“Debug”——“Starting Debug”。

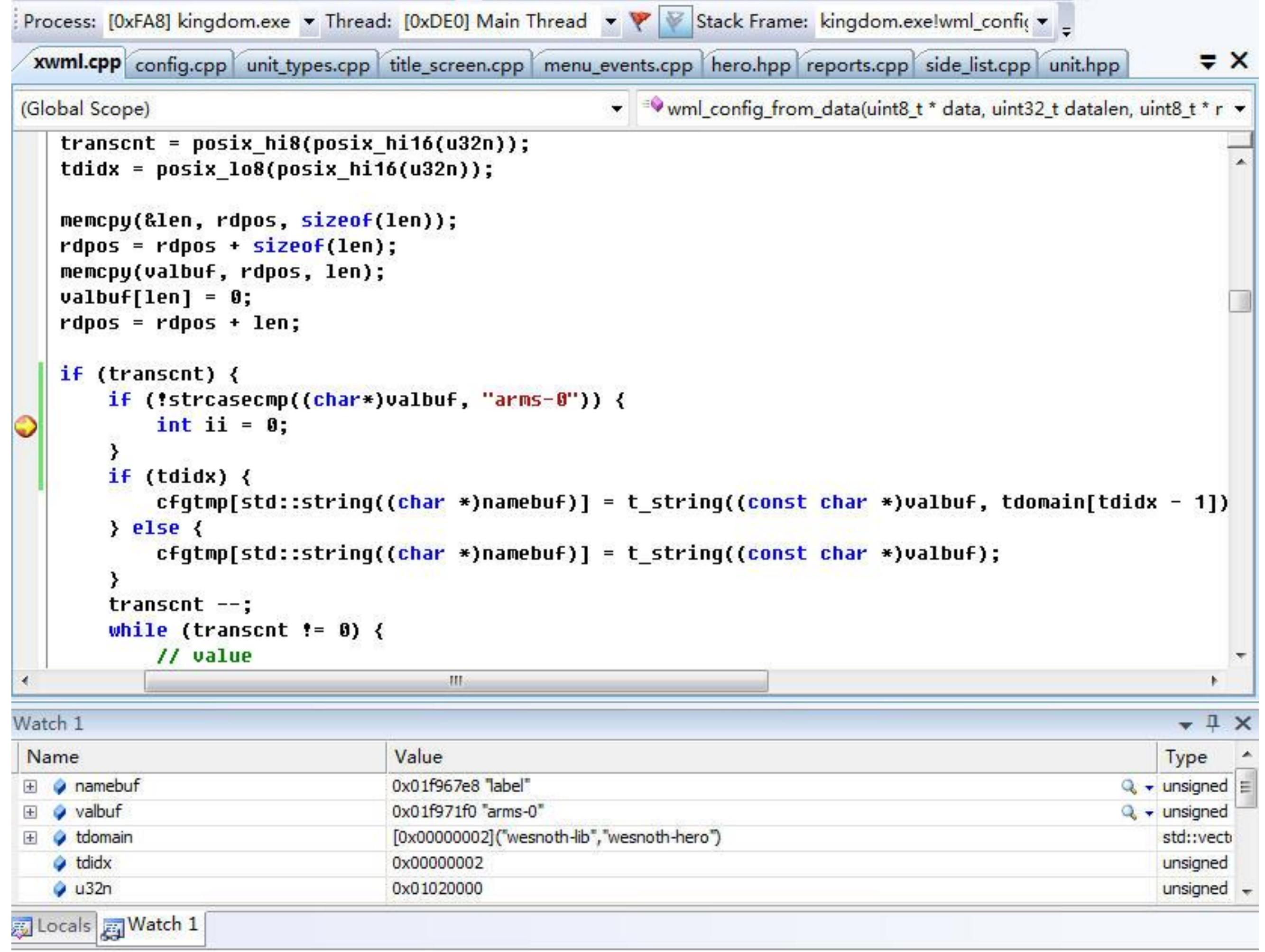


图 3-13 wml_config_from_data

- namebuf：属性名。
- valbuf：属性值。它是可翻译字符串中的关键值。
- tdomain：在该*.bin 中注册了的域。
- tdidx：该可翻译字符串翻译时须要的域在 tdomain 中索引加 1。加 1 是为了不使用 0，此处 0 值有特殊意义，它表示这个子串不是可翻译字符串。当属性值是由 N 个可翻译子串、M 个普通子串混和而成时，tdidx 等于 0 就表示这一子串是普通子串，非 0 表示可翻译子串。

u32n：翻译子串四字节前缀标志。以下是翻译字符串存储规则。

1. 每子串以<四字节前缀标志><四字节长度><字符串值>存储，即整字符串的存储格式：<第 0 子串><第 1 子串>...<第 N-1 子串>。
2. 四字节前缀标志格式

   2.1、如果该子串是整窜第一串，它的 hi8(hi16(u32))值是子串数目,否则置 0

   2.2、如果该子串是可翻译字符串，它的 lo8(hi16(u32))值是 textdomain 索引+1,否则置 0

   2.3、lo16(u32)固定置 0。

   此处前缀标志值是 0x01020000，即可翻译字符串是由一个可翻译子串组成，翻译要用到域在 tdomain 中索引是 1（2 - 1）。

*.bin 如何写入四字节前缀标志参考 wml_config_to_fp 函数。

接下描述 tdomain 变量中的那些个域是如何形成的。

*.bin 在 20 字节后写有该 bin 所需要的域。以下代码摘自 wml_config_to_fp，它会把域写入

*.bin。

```
max_str_len = posix_max(WMLBIN_MARK_CONFIG_LEN, WMLBIN_MARK_VALUE_LEN);
posix_fseek(fp, 16 + sizeof(max_str_len) + sizeof(u32n), 0);

// write [textdomain]
foreach (const config &d, cfg.child_range("textdomain")) {
    if (std::find(tdomain.begin(), tdomain.end(), d.get("name")->str()) != tdomain.end()) {
        continue;
    }
    tdomain.push_back(d.get("name")->str());
    u32n = tdomain.back().size();
    posix_fwrite(fp, &u32n, sizeof(u32n), bytertd);
    posix_fwrite(fp, tdomain.back().c_str(), u32n, bytertd);
}

wml_config_to_fp(fp, cfg, &max_str_len, tdomain, 0);

// update max_str_len/textdomain_count
posix_fseek(fp, 0, 0);

// 0--15
u32n = mmioFOURCC('X', 'W', 'M', 'L');
posix_fwrite(fp, &u32n, 4, bytertd);
posix_fwrite(fp, &nfiles, 4, bytertd);
posix_fwrite(fp, &sum_size, 4, bytertd);
posix_fwrite(fp, &modified, 4, bytertd);
// 16--19(max_str_len)
posix_fwrite(fp, &max_str_len, sizeof(max_str_len), bytertd);
// 20--23(textdomain_count)
u32n = tdomain.size();
posix_fwrite(fp, &u32n, sizeof(u32n), bytertd);
```

编写 MOD 时由于疏忽可能造成[textdomain]中的 name 出现重名，这时要避免重复写入。因而真正写入*.bin 内[textdomain]块数的是 tdomain.size()，而不是 cfg.child_count("textdomain")。

根据代码看出写入什么域根据该 config 的[textdomain]块。打开<kingdom-src>\data\gui\default.cfg。

```
[textdomain]
    name="wesnoth-lib"
[/textdomain]
[textdomain]
    name="wesnoth-hero"
[/textdomain]

#textdomain wesnoth-lib
###
### Defines a gui, all widgets and windows used in a certain 'theme'.
###
..........................
```

以上用于形成两个[textdomin]块。#textdomain wesnoth-lib 指令则是之上描述的告诉 WML 解析器把 wesnoth-lib 设为可翻译字符串的当前域。

4.2.4 公式（formula）

公式类似于数学中函数，它定义函数规则，让自变量 x 取不同值时得到不同的应变量 y。

```
width="(if(screen_width < 800, screen_width, 800))"
height="(if(screen_height < 600, screen_height, 600))"
```

示例中 screen_width、screen_height 是自变量，它实现了这么个宽度计算：屏幕宽度小于 800 时取屏幕宽度，否则取 800。同理，屏幕高度小于 600 时取屏幕高度，否则取 600。怎样在 WML 编写公式？首先要知道代码是如何处理公式。

```
game_logic::formula f("if(screen_width < 800, screen_width, 800))", nullptr);
game_logic::map_formula_callable variables;
variables.add("screen_width", variable(640));
int width = f.evaluate(variables, nullptr).as_int();
```

```
class map_formula_callable {
    .....
    std::map<std::string, variant> values_;
}
```

处理公式分四步。1）以字符串生成一个 formula 对象。2）构建映射（map_formula_callable），它封装可用的自变量信息，具体存储数据的是 values_成员。values_中的“std::string”是自变量名，例子中是“screen_width”。variant 存储自变量值，它可表示多种类型，像整型（TYPE_INT）、浮点（TYPE_DECIMAL）、字符串(TYPE_STRING)，例子中是“640”，variant 会把它存为整型。3）计算表达式值（formula::evaluate），过程中会遇到自变量赋值操作，该操作就是用映射中的 variant 去代替。4）结果转换为指定类型返回到上层，例子中返回的是整型（as_int）。

知道公式怎么被执行，接下深入表达式语法。表达式第一个和最后一个字符必须是“()”。构造 formula 可用不是“()”包含的字符串，这时计算等同简单的类型转换。和这个有关，formula 提供了两个判断函数。1）has_formula。判断字符串是否是表达式。2）has_formula2，只要字符串非空就返回 true，has_formula 是 true 时它一定是 ture，但反过来则不一定。在判断某个 key 是否设置了值时，常用 has_formula2，而不是 has_formula，因为可能只为这 key 设置了常数值，没用表达式。

再次强调，这里说的表达式是第一个和最后一个字符必须是“()”的字符串。

自变量

命名自变量时，字符只能用[A,Z]、[a,z]、或 [-]（不能是第一个字符）。**不能使用数字。**

类型	构造示例	TYPE_NULL(注 1)
TYPE_INT（整型）	variant(640)，varant(true)(注 2)	0
TYPE_DECIMAL（浮点）	variant(3141, DECIMAL_VARIANT) (注 3)	0
TYPE_STRING（字符串）	variant("hello")	不允许

注 1。作为基本原则，代码要确保那些在表达式出现的自变量在映射中应该有对应项。万一没对应项，该自变量的值会被视为 TYPE_NULL。TYPE_NULL 可“安全”转换到 TYPE_INT、TYPE_DECIMAL，结果值是 0。不允许转换到 TYPE_STRING，会触发 assert。

注 2。自变量没有专门布尔类型，但构造 variant 时可直接用 true、false。内部会存储为整型，true 时值是 1、false 时值是 0。虽然内部没有布尔型，但表达式的结果支持布尔类型，见底下的“应变量”。

注 3。构造浮点时，第一个参数也须要整形值，它是希望的浮点值乘上 1000，示例中 3141 表示 3.141。一旦一个自变量使用浮点，此个表达式的结果将就是浮点，必须用 as_decimal，而且它返回的结果也是乘过 1000 了的。

应变量

以下是支持的应变量类型，也就是说，表达式的计算结果可以用以下类型返回给上层调用。

可使用类型	formula 相关函数	
bool	as_bool	一个非 0 的整型或非空的字符串会认为是 true，否则 false
int, unsigned	as_int	
std::string, t_string	as_string	
float, double	as_decimal	结果乘上了 1000

常量

操作数可使用常量，常量可以是整型、浮点和字符串。**表示字符串用单引号，不是双引号。**

运算符

or	逻辑或，等同 C/C++中的“&&”
and	逻辑与，等同 C/C++中的“||”
=，!=，<，>，<=，>=，	对“是否等于”，C/C++是“==”，这里是“=”
+，-	加法可用于字符串，表示合并两字符串
*，/	乘法、除法
%	取模
^	幂运算（C/C++中的 pow 函数），不是异或

从上到下，优先极变高。同一行的运算符有一样的优先级。和 C/C++一样，支持用“()”改变优先级。和 C/C++一个很大区别是“是否等于”，C/C++是“==”，这里是“=”。

控制流语句

```
if (<cond>, <then>, <else>)
```

一个用了“if”的示例。

```
// expression
name="(if(screen_width >= 800, text + '~SCALE(480, 360)', text + '~SCALE(240, 180)'))"

// variables map
screen_width: 480
text: data/campaigns/Hero_Race/images/campaign_image.png

// result
name=data/campaigns/Hero_Race/images/campaign_image.png~SCALE(240, 180)
```

调试公式

```
game_logic::map_formula_callable variables;
variables.add("text_width", variant(120));
variables.add("width", variant(160));
game_logic::formula_debugger debugger;
game_logic::formula f("(if(text_width > width, 0, (width - text_width) / 2))", nullptr);
int v = f.evaluate(variables, &debugger).as_int();
```

内置提供了一个公式调试器。运行上面代码，就可以像调试 C/C++代码一样让公式单步执行，然后看当前的栈、相关值。和不支持调试的代码相比较会发现，它们区别是计算时（evaluate）的第二个参数是否设置了有效的 formula_debugger 对象。

4.2.5 逐进变量（Progressive Variable/Parameter）

在编写 mod 时会遇到这样一类语句。

```
[magic_missile_trail_3_frame]
    duration=350
    halo=misc/blank-hex.png:120,halo/mage-preparation-halo1.png,halo/mage-preparation-halo2.png,halo/mage-preparation-halo3.png,halo/mage-preparation-halo4.png,halo/mage-preparation-halo5.png,halo/mage-preparation-halo6.png,halo/mage-preparation-halo7.png,misc/blank-hex.png:1
    halo_y=-54:120,-54~-45,-45~-27,-27~0
    offset=0.001:120,-0.5~-0.25,-0.25~0.25,0.25~1.0
[/magic_missile_trail_3_frame]
```

像 halo 这样属性，它的值以一连串逗号隔开，把这样变量称为逐进变量，而被逗号隔开部分则称为逐进变量某一值项。

逐进变量取什么值和具体访问时刻相关。**当处于某一时刻，逐进变量只能等于某一值项**。像 halo 变量，它只能或是 halo=misc/blank-hex.png，或是 halo/mage-preparation-halo1.png，或是 halo/mage-preparation-halo2.png，等等。

那么这个时刻是怎么来的？以及它是如何决定变量该取哪个值项？

时刻来自于同标签下的 duration 变量，以上这个 halo 就是 duration=350（和 halo 一样，halo_y 及 offset 也是以 duration 为参考）。

duration 如何发展挥作用？duration 是作为一个过程时间（某一行为从开始到结束，默认单位：毫秒），这个过程时间将被分成 N 个时间片，N 值等于 halo 变量中值项数，以上 halo 这个 N 值等于 9，halo_y 和 offset 的 N 值则等于 4。duration/N 就是时间片长度，halo 变量的时间片长度 38 毫秒（350/9），halo_y 和 offset 则是 87 毫秒。duration 发挥作用时需要借助一个外来变量，可以称为它当前时刻（cur_time），当前时刻被限定在[0, duration-1]，根据这个时刻和时间片就可以定位出某一值项。例如当前时刻是 30 毫秒，halo 值就是在#0 时间片，值 misc/blank-hex.png。

这类变量要得到的值是依着书写次序被逐个赋与，就像时间一样在逐进，于是称之为逐进变量。

时间片长度是不是只能是 duration/N？不是。在一些情形时不希望 duration 被平均分配，例如想让 halo=misc/blank-hex.png 时间更长点，这时就可以使用在字符串后加上以冒号跟的时间值，就像 misc/blank-hex.png:120，这时 misc/blank-hex.png 将执行 120 毫秒，而不是默认计算出的 38 毫秒。这就意味着，当当前时刻是 100 毫秒时，halo 的值还是等于 misc/blank-hex.png。

引入冒号加时间值无疑会改变某值项时间片，这个变动进一步要影响到整个过程时间。举个例子，misc/blank-hex.png 被从 38 改为 120，将使过程时间增加 82 毫秒，那这个增加的时间会不会从后面值得到补偿呢？即后面值要从默认的 38 减去数毫秒，从而让这 9 个值项总和还是 350 毫秒？——会。代码分两步计算各值项时间片，1）计算带冒号时间的项，它们的时间片设为冒号时间，2）duration 减去冒号时间和（total_explicit_time），剩余时间除以不带冒号的时间项数，算出平均值设为那些项的时间片。按这规则，total_explicit_time 必须"小于等于"duration。

到此，要理解变量 halo 是差不多了，但 halo_y 和 offset 还不够，它们在值项中还一个波浪号。波浪号作用是指定一个范围，它对当前时刻和值的关系进一步细分。没有波浪号时，时间片内只能取一个值，像 halo_y=-54:120，在整个 120 毫秒内 halo_y 只能等于-54，而用了波浪号后，像 halo_y=-54~-45，在这 87 毫秒内，它的值随着当前时刻将在[-54, -45]这个范围内取值。

值的计算公式：（中间结果以浮点格式）。

```
cur_val = (cur_time - time) / slice_time * (second - first) + first
```

- cur_time：当前访问时刻。
- time：选中的时间片的起始时刻。
- slice_time：选中的时间片的持续时间。
- first：选中的时间片的起始值。
- second：选中的时间片的结束值。

针对以上 halo_y=-54:120,-54~-45,-45~-27,-27~0，当当前时刻是 130 毫秒时，依据公式可以计算出 halo_y 值将是-53。

```
(130 - 120) / 87 * 9 + (-54) = -53
```

更多代码处理逐进变量细节参考"逐进变量"。

4.3 从 WML 生成 config

4.3.1 两种处理配置数据逻辑

不使用转换的脚本文件编写、生成、使用架构

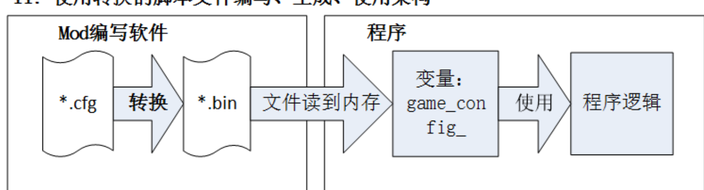


图 4-2 两种“bin”MOD 架构

第二种让在制作 MOD 阶段多了个步骤，转换：由多个 cfg 产生了一个*.bin 文件，让看下多出这 bin 中间形式的好处。

减少游戏启动时间。*.bin 存的是二进制文件，设计它时会尽量做到它和最终可执行程序需要的数据结构 config 之间转换是“零”开销。

让游戏程序模块化。当源代码越来越大时，维护、修改、编译、链接都会变得越来越庞大。转换作为一个较明显的可以和主程序脱开的模块，把它剖离开来至少可以减少一部分工作量。

mod 编写更快捷、简单、高效。编写 mod 说来就是看懂各个*.cfg，然后根据自己要的写*.cfg 文件。到现在，如果写 mod 还是停留在要用文本编辑器，要靠眼睛去严格匹配脚本语法，这是一个极重工作量，而且效率和质量都没程序高。一旦把脚本解析和主程序脱离后，就可以考虑使用像 eclipse 这种集编辑、编译为一体的可视化工具，输入是脚本语法，输出就是要求的*.bin。

编写 mod 可选用语言更灵活。写久了程序的人想来会有个感叹，C/C++，PHP，XML，为啥世界要有这么多种语言！WML 好吗，个人认为是挺简单，但无论怎么简单它毕竟是一种“新”语言，对熟悉 XML 的人来说要是 XML 会更好。使用*.bin，可以让编 mod 人使用自己语言，只要这种语言都够“出来”*.bin 要求的数据。

4.3.2 config_cache::get_config

```
config_cache::get_config(const std::string& path, config& cfg);

config_cache& cache = game_config::config_cache::instance();
```

get_config 的输入参数“path”可以是文件、也可以是目录。它根据 path 中写的配置生成一个 config，结果放在 cfg。要注意，在生成时一定会先清掉之前 cfg 内容。get_config 是 config_cache 的方法，要没特殊需要，可使用系统提供的静态对象，获取方法是上面的 config_cache::instance()。

4.3.3 config_cache_transaction

为什么会有 config_cache_transaction，让先看 get_config 的一个缺陷：它不能记住宏。如果想达到以下这么个目的：要读取“chat.cfg”，这文件要使用“GUI__CHAT_WIDGET”宏，可这宏在“template.cfg”中定义，而且“chat.cfg”中没有写要包含“template.cfg”。这就是说，单独解析“chat.cfg”会因为找不到“GUI__CHAT_WIDGET”宏的定义而失败。为实现这目的须进行两次 get_config，第一次解析“template.cfg”，第二次解析“chat.cfg”，而且第一次解析出的宏要应用到第二次，即要能记忆宏。config_cache_transaction 就是为这目的面出现的，它实现了让前面解析出的宏可用于之后。

```
config cfg;
config_cache_transaction transaction;
cache.get_config("template.cfg", cfg);
cache.get_config("chat.cfg", cfg);
```

正如 config_cache_transaction 中的单词“transaction”，它就是把数个 get_config 组织成为一次事务性操作，其目的就是让后面的 get_config 可共享前面产生的宏。一旦 transaction 对象被析构，事务结束，记忆中的宏消失。另外之前有写过，get_config 在生成时一定会先清空 cfg，这意味着以上代码的第二次 get_config 会先清空第一次生成的内容，但如果 template.cfg 写的全是宏，那它生成的 cfg 其实是空的。

4.3.4 从 WML 格式的字符串生成 config

```
<librose>/serialization/parser.hpp

void read(config &cfg, const std::string &in);
```

read 实现了从一个 WML 格式的字符串生成 config。使用时要注意两点，一是字符串中出现的 WML 语法只能是纯粹的块、属性结构，像不能出现宏，WML 文件、目录扩展。二是它默认使用的翻译域是“rose-lib”，即遇到一个可翻译字符串时，要是没专门指定翻译域，它会由“rose-lib”和该 msgid 联合产生 t_string。要默认到另一个翻译域，像“studio-lib”，那可以在字符串前添加结尾带换行符的“#textdomain studio-lib”，然后把新字符串作为参数调用 read。

4.4 C 和 Lua 相互调用

此处的 C 是 C 和 C++合写，从以下例子可以看出，Lua 既可以被 C++类成员函数调用也可直接调用 C++类中成员函数。

Lua 是种简洁、轻量、可扩展的脚本语言，要学习 Lua 可上 Lua 官方网站（http://www.lua.org）看《Lua 参考手册》，除手册外建议看《Lua 程序设计.第二版》。这里假定读者对 Lua 已有基本概念，以一个例子叙述 C 和 Lua 之间相互调用。不过让首先引用《Lua 程序设计.第二版》的“C API 概述”中两段话。

Lua 使用一个库来扩展应用程序的能力使得 Lua 成为一种“扩展语言（Extension Language）”。而与此同时，一个使用了 Lua 的程序可以在 Lua 环境中注册用 C 语言（或其它语言）实现的新函数，由此就可以向 Lua 添加某些无法直接用 Lua 编写的功能，这便使 Lua 成为一种“可扩展语言（Extensible Language）”。

这两种使用 Lua 的方法对应于 C 语言和 Lua 之间的两种交互形式。在第一种形式中，C 语言拥有控制权，Lua 是一个库。这种形式中的 C 代码称为“应用程序代码”。在第二种形式中，Lua 拥有控制权，C 语言是一个库，因此 C 代码称为“库代码”。应用程序代码和库代码都使用同样的 API 来与 Lua 通信，这些 API 称为 C API。

例子：设置关卡目标

让进入代码级调试。

1、在 kingdom 工程中打开<src>/team.cpp。

2、void team::set_objectives(const t_string& new_objectives, bool silently)函数内增加代码，增加点在一进入函数后。

```
if (!new_objectives.empty()) {
    int ii = 0;
}
```

3、重编译 kingdom.exe。

4、把断点设在“int ii = 0”。

5、运行时选“Debug”——“Starting Debug”。

6、进入主菜单后当前语言设为英文。

7、“Campaign”——“Hero Race”

8、调出“Call Stack”窗口，选择 Luakernel::run_wml_action 函数。

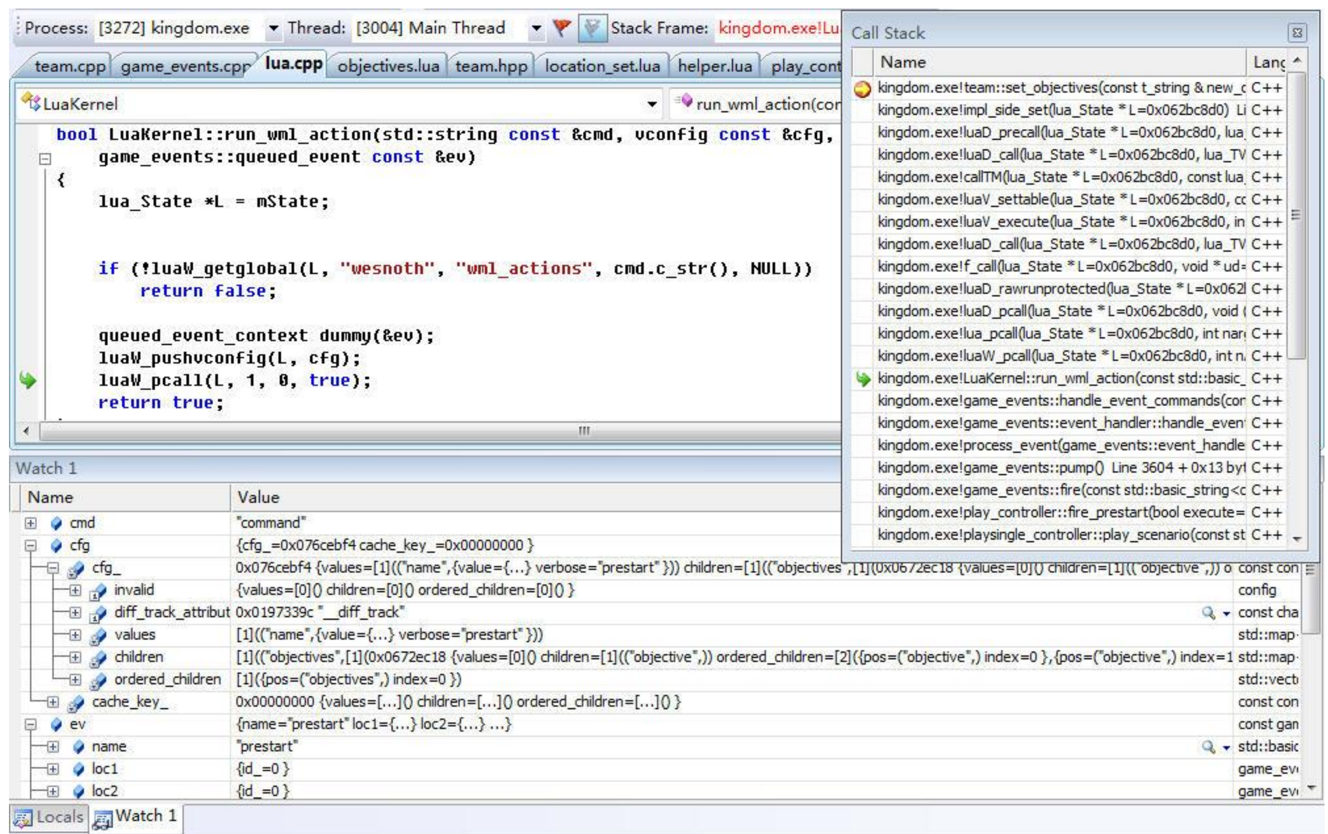


图 3-15 luaW-pcall

程序执行的是把关卡目标设置到势力。

WML 中和关卡目标相关代码就是代码 3-1，显示出来的目标在图 3-12。为对照方便这里重复下代码 3-1。

```
[event]
        name=prestart
        [objectives]
                [objective]
                        description= _"Defeat all sides"
                        condition=lose
                [/objective]
                [objective]
                        description= _"No city you holded"
                        condition=win
                [/objective]
        [/objectives]
[/event]
```

设置关卡目标到势力的总逻辑

1. 程序触发“prestart”事件，game_events::fire 把“prestart”压入要处理事件队列。
2. game_events::pump()从事件队列中抽出事件，prestart 是可处理事件，并找到以上写的 name=prestart 这个 config 对应的事件处理者（game_event::event_handler），这个事件处理者一个很重要私有成员（cfg_）是它的 WML 块，即代码 3-10 的[event]块。
3. 事件处理者调用 game_event::event_handler::handle_event(...)执行具体处理。
4. handle_event 则调用 LuaKernel::run_wml_action 实现处理。注意下传给 run_wml_action 的三个参数。
   @cmd：命令字符串。它用于在“wesnoth”这个全局表变量中搜出可处理函数。
   @cfg：事件处理者的 WML 块，即代码 3-10 的[event]块。
   @ev：处理该事件时上下文。此处就是 name 字段有效。
5. run_wml_action 主要功能是调用一个 lua 函数，这个函数的名称写在全局表“wesnoth”下的“wml_actions”子表，在后者以“command”为关键字的表项中。

注：LuaKernel::run_wml_action 内栈长度变换。（假设原来栈长度 S 是 N，一般 N=0）

- luaW_getglobal(L, "wesnoth", "wml_actions", cmd.c_str(), NULL)会把得到的函数名压栈。S=N+1。
- luaW_pushvconfig(L, cfg)会把自定义类型的 cfg 参数压栈。S=N+2
- luaW_pcall(L, 1, 0, true)会从栈中弹出一个参数和一个函数。S=N。

函数执前后栈中是同样内容。

被执行的函数叫什么？（开始进入 lua 代码）

以上已分析出这函数是在 wml_actions 表的 command，打开<res>/data/lua/wml-tags.lua，会找到“wml_actions.command = handle_event_commands”，即这个要被执行函数是 handle_event_commands，handle_event_commands 就是要被执行的函数。

handle_event_commands 也写在<src>/data/lua/wml-tags.lua，以下是代码。

```
local function handle_event_commands(cfg)
	for i = 1, #cfg do
		local v = cfg[i]
		local cmd = v[1]
		if not string.find(cmd, "^filter") then
			cmd = wml_actions[cmd] or
				helper.wml_error(string.format("[%s] not supported", cmd))
			cmd(v[2])
		end
	end
	-- Apply music alterations once all the commands have been processed.
	wesnoth.set_music()
end
```

handle_event_commands 枚举 cfg 下的每个子块，对每个子块调用 wml_actions.cmd（cmd 是子块名）。针对这个设置关卡目标，它只有一个 objectives 子块，也就是要调用 wml_actions.objectives。

打开<src>/data/lua/wml/objectives.lua，以下是它代码。

```
function wml_actions.objectives(cfg)
	cfg = helper.parsed(cfg)
	local side = cfg.side or 0
	local silent = cfg.silent

	-- Save the objectives in a WML variable in case they have to be regenerated
later.
	cfg.side = nil
	cfg.silent = nil
	scenario_objectives[side] = cfg

	-- Generate objectives for the given sides
	local objectives = generate_objectives(cfg)
	if side == 0 then
		for side, team in ipairs(wesnoth.sides) do
			team.objectives = objectives
			if not silent then team.objectives_changed = true end
		end
	else
		local team = wesnoth.sides[side]
		team.objectives = objectives
		if not silent then team.objectives_changed = true end
	end
end
```

helper.parsed 得到的 cfg 很象 WML 中的块，cfg.side 等同访问 cfg 这个块的 side 属性。

例子没有 side 属性，因而 local side 得到值是 0，local silent 得到值也是 0。if side == 0 条件为真，执行 for 循环，for 循环体内主要是两条语句：

team.objectives = objectives；会调用 C 代码 team::set_objectives。

if not silent then team.objectives_changed = true；会调用 C 代码 team::set_objectives_changed。

为什么 team.objectives = objectives 会调用 C 代码 team::set_objectives？

team 落在 wesnoth.sides 表中，找到 lua.cpp 的 LuaKernel::initialize()。

```
// Create the sides table.
lua_getglobal(L, "wesnoth");
std::vector<team> &teams = *resources::teams;
lua_pushlightuserdata(L, (void *)&getsideKey);
lua_rawget(L, LUA_REGISTRYINDEX);
lua_createtable(L, teams.size(), 0);
for (unsigned i = 0; i != teams.size(); ++i) {
    // Create a full userdata containing a pointer to the team.
    team **p = static_cast<team **>(lua_newuserdata(L, sizeof(team *)));
    *p = &teams[i];
    lua_pushvalue(L, -3);
    lua_setmetatable(L, -2);
    lua_rawseti(L, -2, i + 1);
}
lua_setfield(L, -3, "sides");
lua_pop(L, 2);
```

以上创建了 sides 表，它位在注册表空间，并作为 wesnoth 表中的一个元素。sides 表项数就是当前战役的势力数，以下是 sides 表格式。

key	Value
1	0xp0(team[0]指针)
2	0xp1(team[1]指针)
…	……
N	0xp(N-1)(team[N-1]指针)

team.objectives = objectives 是要对表中 objectives 索引赋值，但表中不存在 objectives 索引，lua 解释器就调用 sides 表的__newindex 方法。

找到 lua.cpp 的 LuaKernel::LuaKernel(const config &cfg)。

```
// Create the getside metatable.
lua_pushlightuserdata(L, (void *)&getsideKey);
lua_createtable(L, 0, 3);
lua_pushcfunction(L, impl_side_get);
lua_setfield(L, -2, "__index");
lua_pushcfunction(L, impl_side_set);
lua_setfield(L, -2, "__newindex");
lua_pushstring(L, "side");
lua_setfield(L, -2, "__metatable");
lua_rawset(L, LUA_REGISTRYINDEX);
```

代码创建 sides 表（它在 wesnoth 表的关键字是(void *)&getsideKey，&getsideKey 正是 sides 表索引），并为 sides 表设置__index、__newindex 元方法。

注：以上代码栈长度变换。（原来栈长度 S 是 N，一般 N=1，1 是由之上的 luaL_register(L, "wesnoth", callbacks)压入）

- lua_pushlightuserdata(L, (void *)&getsideKey)。S=N+1。
- lua_createtable(L, 0, 3)。S=N+2。
- lua_pushcfunction(L, impl_side_get)。S=N+3。
- lua_setfield(L, -2, "__index")。S=N+2。
- lua_pushcfunction(L, impl_side_set)。S=N+3。
- lua_setfield(L, -2, "__newindex")。S=N+2。
- lua_pushstring(L, "side")。S=N+3。
- lua_setfield(L, -2, "__metatable")。S=N+2。
- lua_rawset(L, LUA_REGISTRYINDEX)。S=N。

代码执行前后栈中是同样内容。

到此分析出 team.objectives = objectives 会调用 impl_side_set，并且不存在的关键字是“objectives”，值则是 objectives 这个局部变量。

```
static int impl_side_set(lua_State *L)
{
    // Hidden metamethod, so arg1 has to be a pointer to a team.
    team &t = **static_cast<team **>(lua_touserdata(L, 1));
    char const *m = luaL_checkstring(L, 2);
    // Find the corresponding attribute.
    modify_int_attrib("gold", t.set_gold(value));
    modify_tstring_attrib("objectives", t.set_objectives(value, true));
    modify_int_attrib("village_gold", t.set_village_gold(value));
    ...............
}
```

modify_tstring_attrib("objectives", t.set_objectives(value, true))正好和不存在的关键字匹配，于是会调用 team::set_objectives(value, true)，value 就是 lua 中 objectives 这个局部变量。

代码执行过程从“C 应用程序代码”——“Lua 代码”——“C 库代码”。Lua 实现了它是 C 的扩展库（扩展语言），又实现了把 C 作为它的扩展库（可扩展语言）。

4.5 混合 WML、Lua 编写配置

就以设置关卡目标到势力来说，Lua 除了处理支持 objectives 动作，还有就是给字符串添加显示效果，像胜利字符串显示绿色，失败字符串显示红色。的确，WML 可以实现让不同字符串显示不同颜色，但要实现功能往往就要导致重编译可执行程序，而放在 lua 中实现好处是只要改*.lua 这个文本文件就行。能实现不编译程序就实现灵活配置功能，对编写 MOD 非常重要。

WML 优点是相比 Lua 执行效率高，但改动一些地方可能须要重编译程序；Lua 优点是有灵活逻辑控制、改动后不必重编译。

4.5.1 检查 WML、Lua 语法

WML、Lua 都是种配置语言，它们遵循一定语法，书写总会存在错误，这就须要对它们进行语法检查。

- 编辑器编译时会检查 WML，有错误时就会提示。
- Lua 代码则须在程序运行被检查，有错误时会显示在主界面上。

第五章 GUI：窗口

本章目标

- 理解窗口注册、加载、创建过程。
- 如何布局控件。
- 窗口如何处理用户输入。

第三章“框架逻辑”有说到 Rose 把窗口分为两类：对话框、场景，区分它们的依据是当中是否存在大地图。在这一章叙述窗口通用技术，涉及到窗口类型地方是以对话框为例子，下一章则介绍场景。对话框特殊代码基本可归结为是一处：twindow::show（场景不调用这函数，而是更为简单的 twindow::asyn_show）。为持续渲染，对话框在 twindow::show 不断调用 absolute_draw，场景的触发渲染时机则是通过 display::draw。

术语

- 定义控件：定义某种风格的控件。WML 中，定义控件指控件定义块（“_definition”为后缀的 WML 块）；C/C++中，定义控件对应一个控件定义对象（“tcontrol_definition”派生类）。
- 使用控件：在窗口配置中使用某个控件定义。
- 窗体：窗体是一种控件。窗体是窗口顶层控件，每个窗口有并且只有一个窗体控件。

5.1 app 的对话框编程

5.1.1 基本步骤

第一步：向 app 新建对话框

运行 studio，在左侧树形控件的顶结点按鼠标左键，弹出菜单中选择要增加对话框的 app，子菜单选择“新建对话框”，假设新建的对话框标识是“home”。新建成功会增加三个文件，一个是放入资源包的 home.cfg，它是窗口脚本。另两个放入源码包，分别是 home.cpp、home.hpp，存放着处理该对话框的 C/C++代码。

第二步：修改窗口脚本

运行 studio，单击“窗口”，进入窗口编辑器。打开 home.cfg，通过不断增加、删除、修改控件，改出你希望的样子。

第三步：修改对话框对应的 cpp/hpp

在 Visual Stuido，修改 cpp/hpp，让改出希望达到的效果。对话框逻辑是用户事件驱动，像按下鼠标、触擦、键盘，等等，于是可按运行时刻把代码分两个阶段，第一阶段是第一次显示前，第二阶段是在这之后。

第一阶段的工作是编程 pre_show。它是个重载函数，app 在这函数主要做两件事，一是填充各控件的初始数据，二是向内中控件挂接感兴趣的事件处理函数，像列表改变选中行时，展开树形结点时。第二阶段是编程那些个事件函数，对第一阶段挂接向控件的函数，现在是编程它们是时候了。

不要在对话框的析构函数中使用指向窗口中控件的指针，同样原因，此时 window_成员是 nullptr，要使用的改在 post_show。执行完 post_show，对话框将销毁窗口，那些指针将非法。

第四步：其它模块使用该对话框

```
gui2::thome dlg(...);
dlg.show();
res = dlg.get_retval();
```

逻辑分三个步骤，一是构造对话框，二是消息循环，三是得到返回值。消息循环更多细节参考“**3.2.5 消息循环：对话框**”。

5.1.2 修改控件中字符的颜色

在“控件设置”的“高级”页选择特定颜色模板，Default color、Inverse color 或 Title color。设置操作就这么多，要理解这 3 种模板，首先要知道主题怎么管理颜色。

app 在任一时刻都处于某一种主题，主题把颜色组织成两级架构。第一级是模板，有三种，然后每种模板有四种颜色，即每一种主题预定义了 12 种颜色。

类型	说明	C/C++枚举值	default 主题
默认	最经常使用的一组颜色	default_tpl	偏黑色
反白	和默认有较大不同的一组颜色	inverse_tpl	偏白色
标题	app 可能会为标题使用专门颜色	title_tpl	偏黑色

Rose 默认给出的“default”主题是白天模式，适用于白底黑字。下表是每一种模板下的四种颜色。

C/C++枚举值		default 主题 default_tpl 时的值(A, R, G, B)
normal	通常状态时的颜色	255, 20, 20, 20
disable	表示禁用时颜色	255, 128, 128, 128
focus	表示获得了焦点时颜色	255, 0, 0, 255
placeholder	编辑框占位提示用的颜色	255, 119, 119, 119

C/C++代码调用 text_color_form_index 得到一种颜色，然后就可用它显示文字或绘制简单图形，像直线、矩形。

```
uint32_t theme::text_color_from_index(int tpl, int index);
```

以上是 C/C++代码如何设置颜色，gui 脚本方面则是按之前写在“高级”页选择特定颜色模板。举个例子，default 风格的 label 控件，默认使用“default_tpl”模板（不仅是它，所有控件默认都这模板），如果你想把这控件用作标题，就把它切到“title_tpl”，想把它“反白”就改为“inverse_tpl”。

自写控件

（不自写控件的可略过这部分）。在定义控件时，可使用三种类型的颜色。

类型	格式	示例
预定义	0 开始的数值。0--normal，1--disable，依此类推	0、1
公式	以“()”包含的公式	(cursor_color)，编辑框用它表示光标颜色。控件需定义 cursor_color 这个结果是 uint32_t 的自变量
argb 字符串	以逗号分隔的 4 元组格式，依次表示 A、R、G、B	“128, 255, 0, 0”一表示 50%透明度的红色。“255, 255, 255, 0”表示不透明的黄色。

预定义指的就是主题中预定义的颜色，它的数值对应 C/C++枚举值。为易记定义了三个宏。

宏	C/C++枚举值
GUI__FONT_COLOR_ENABLED__DEFAULT	normal(0)
GUI__FONT_COLOR_DISABLED__DEFAULT	disable(1)
GUI__FONT_COLOR_FOCUS__DEFAULT	focus(2)

5.1.3 修改控件中字符的字号

在“控件设置”的“高级”页选择要使用的字号，smaller、small、default、large 或 larger。设置操作就这么多，接下深入说这 5 种字号。

使用文字的场合分为两种，一是 gui 脚本，二是 C/C++代码，为关联它们，引入个术语叫参

考字号。

```
#define font_cfg_reference_size 1000
#define font_max_cfg_size_diff  2
#define font_min_cfg_size   (font_cfg_reference_size - 2)

int font_hdpi_size_from_cfg_size(int cfg_size) {
    int diff = cfg_size - font_cfg_reference_size;
    if (diff == 0) { return font::SIZE_DEFAULT;   }
    if (diff == -1) { return font::SIZE_SMALL;}
    if (diff == -2) { return font::SIZE_SMALLER; }
    if (diff == 1) { return font::SIZE_LARGE; }
    return font::SIZE_LARGER;
}
```

上面是处理字号相关的 C 代码，font_cfg_reference_size 就是参考字号，值须要是个正常字号不可能出现的数字，1000。font_hdpi_size_from_cfg_size 用于从脚本字号转换为 C 代码字号，SIZE_DEFAULT 等 5 个大写变量就是真实在用的字号。这函数实现了下面映射。

易记名	脚本中字号值	C 代码中字号变量(手机/PC) (注 1)
smaller	-2 (998)	SIZE_SMALLER(12/9)
small	-1 (999)	SIZE_SMALL(14/11)
default	0 (1000)	SIZE_DEFAULT(16/13)
large	+1 (1001)	SIZE_LARGE(18/15)
larger	+2 (1002)	SIZE_LARGER(20/17)

举个例子，gui 脚本填的是 1000 时，手机最后算出的是 16，PC 算出的是 13。填 998 时，手机算出的是 12，PC 是 9。这套机制要实现两个目的，1）统一 gui 脚本和 C/C++代码这两个系统的字号。2）做到了 gui 脚本中字号和设备无关，它只要记住 1000 是默认，至于 1000 对应哪个字号则由 C/C++代码根据具体设备去设定。

gui 脚本只能使用 5 种字号，app 须要其它字号时，像天气 app 要用很大数字显示当前气温，可用 track，或 blits 风格的 image 等控件，然后 C/C++代码自画。

只能修改控件配置中是 default 的字号

为什么有些控件配置中的字号不是 default？让看 vertical_node 风格的按钮。它上面是图像，下面是文本，界面要突出的是图像，于是要没意外，底下文本会要求使用 small。为避免开发者每遇到这种按钮都要修改字号，在 verical_node 配置中就写了让文本是 small。

对非 default 字号，一理改了它，控件极可能就不美观了。

自写控件

自写控件时，除了可使用和上面说的 5 种字号，还可以使用绝对字号，像 13、14。由于绝对字号出来的界面往往依赖特定设备，Rose 库没有一种控件使用了绝对字号。

5.1.4 状态栏（Statusbar）

状态栏是 iOS、Android 出现在顶部、用于显示时间、电量的横条。PC 没有。app 可以把它设置为或隐藏、或显示。是隐藏时不占用空间。是显示时，它的空间将包含在窗口中，为正确显示，窗口应该给它留出高度。

状态栏中字体颜色统一是白色，过程中不能修改颜色。为什么不能设置？因为 Android 无法修改颜色，于是只能用它默认的白色。

编写窗口脚本时可使用 statusbar_height，该自变量指示状态栏高度。状态栏是隐藏时，值总是 0。显示时，PC 总是 0，iOS、Andorid 是状态栏高度除以 hdpi_scale。

5.1.5 定时器（ttimer）

ttimer 提供了两个 reset 方法，带参数的 reset 用于创建定时器，不带参数的则用于销毁。

```
void    ttimer::reset(const    Uint32    interval,    twindow&    window,    const
```

```
boost::function<void(size_t id)>& callback, bool top_only)

interval：定时器间隔。单位毫秒。
window：定时器关联的窗口。
callback：定时器关联的 timer_handler。
top_only：默认是 ture。表示只有该窗口在最顶层时才会调用 timer_handler。
```

使用定时器可归纳为三个步骤。1）在 tdialog 内定义 ttimer 对像，假设 timer_。2）要开始定时器时调用带参数的 timer_.reset。3）app 希望窗口被析构时（post_show）才删除定时器，那等 twindow 自动销毁就行，否则要主动调用不带参数的 timer.reset。4）实现一次性定时器的方法是在 timer_handler 退出前执行不带参数的 timer.reset。

定时器既能用在对话框也能用场景，但一定要和某个窗口挂钩，只有该窗口位在最顶时才会执行 timer_handler（top_only=true）。timer_handler 运行在主线程。

同一种定时器事件(data1、data2 相同)不会让发生重入。重入指嵌套调用 timer_handler。

两次 timer_handler 的调用间隔可能小于设定的定时器间隔，但会保证一次 timer_handler、一次 webrtc 的 post，即不会出现连续多次 timer_handler 然后一次 post 的情况。为什么说它？app 为不在定时器中显示对话框，可能会用一次 post，然后在 post 显示对话框。track 控件内置了定时器，但 timer_hander 极可能被非定时器主动调用，像移动鼠标导致要重画 track，这可能会导致多次 timer_handler，然后一次 webrtc 的 post。而且重画 windows 也会调用 timer_handler。对于 track 情况，建议不要认为一次 timer_handle、一次 webrtc 的 post。

定时器内部实现细节

Ttimer 基于 SDL 定时器。SDL 定时器是在一个专门的定时器线程执行，线程函数是 SDL_TimerThread。因为是新线程，自然会遇到和主线程同步问题。为省掉 app 同步烦恼，rose 会确保 app 提供的 timer_handler 是在主线程执行。那它是怎么实现的？

1、SDL_TimerThread 执行回调函数，执行向 SDL 事件队列增加 TIMER_EVENT。

2、主线程的 pump 提取出 TIMER_EVENT，gui2 的 mini_handle_event 处理 TIMER_EVENT，后者调用 app 注册的 timer_handler。

这是一种二次调用方案，会导致主线程(pump)取出 TIMER_EVENT 时失去预设间隔，严重的一次就取到多个。举个例子，app 在主线程执行一个操作花了不少时间，这时会造成多个一样的 TIMER_EVENT 被放入 SDL 事件队列，因为 SDL_TimerThread 是个独立线程，主线程阻塞了不会影响它定时把 TIMER_EVENT 放入 SDL 事件队列。

除了 TIMER_EVENT 失去预设间隔，还会造成收到无主的 TIMER_EVENT。某一时刻删除定时器 A，它不会同时删除 SDL 事件队列中关联的 TIMER_EVENT。如果删除时正有 A 的 TIMER_EVENT 放在 SDL 事件队列，那下次 pump 就会出来这无主的事件。

5.1.6 建议

尽量不要使用以下事件：松开右键、双击左键。原因是移动平台不支持或不推荐这些操作。

原则上不再支持使用水平滚动条，为此尽量不让出现水平滚动。实在避免不了，1）只让小幅滚动，2）界面让有较好提示，举个例子，要在一行显示 10 个按钮，而每屏最多只能只显示两个，那在每个按钮左上角放上(1/10)字样，让用户很快知道前、后还有多少内容。

5.2 网式布局

为什么首先讨论布局算法？1）布局算法会影响如何定义控件。2）布局算法是总纲，控件是细节，确定总纲后才能更好决定细节。Rose 中窗口分为对话框和场景，它们使用一样的网式布局，Studio 提供了网式布局的图形化界面，示例见图 5-1。

特点

使用通常的屏幕坐标系，左上角是原点，向右 x 递增，向下 y 递增。x、y 不可能是负值。

自适应各种分辨率、DPI。

整个窗口是一个网格，格子中或是控件或是网格。

网格中，同一行内格子必相同高度；同一列内格子必相同宽度。

自动化布局，控件左上角坐标取决于上面、左侧控件。除占位控件，建议不要使用固定像素去设定标称尺寸。

控件只会放大不会缩小。采用一次性布局，一旦太大不能放置，认为布局失败。

布局出的窗口不会出现“空洞”。

支持悬浮控件。

四种尺寸

最大尺寸：窗口最大能覆盖到的尺寸，因为能覆盖整个屏幕，最大尺寸值往往是屏幕尺寸。

容许尺寸：根据最大尺寸和窗口脚本中写的[resolution]块计算出的尺寸。最后窗口实际显示的不可能大于这个尺寸。

标称尺寸：累加窗口内各控件标称尺寸计算出的尺寸。要能显示，它必须小于或等于容许尺寸，否则布局失败。

渲染尺寸：根据标称尺寸，结合窗口模板设置进行放大、但不能超出容许尺寸而计算出的尺寸。最后窗口实际显示的就是这个尺寸。

5.2.1 示例

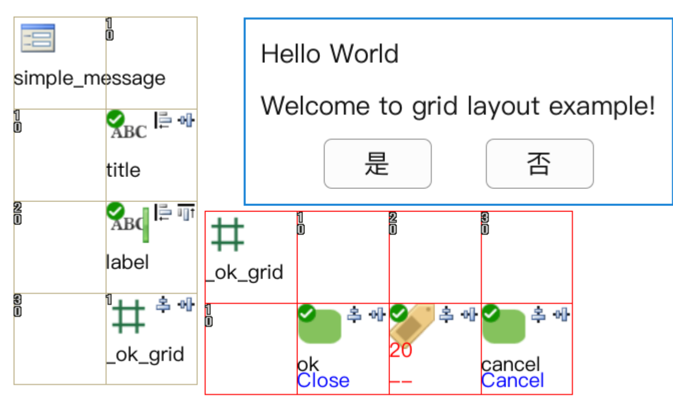


图 5-1 网式布局

示例窗口有两个网格。1x3 网格表示顶层窗口，它分为三行，第一行是文本控件，表示标题。第二行是滚动文本控件，指示要提示的内容。第三行想放两个按钮，于是再用一个网格。_ok_grid 是个 3x1 网格，第一列是按钮控件，表示“确定”，第二列用占位控件，让把两按钮隔开，第三列“取消”按钮。“Hello World”是这窗口的一个运行实例。

接下让看看算法是怎么实现自化布局。1）计算标称尺寸。过程是计算内中所有控件的标称尺寸，然后进行累加，算出是 552x230。2）计算渲染尺寸。容许尺寸是 1136x640，标称尺寸没超过容许尺寸，可以布局。由于此窗口设置的是自动布局，意味着标称尺寸就是渲染尺寸，即渲染尺寸是 552x230。3）放置。放置首先从第一个网格开始，放置顺序是常见的先 x 后 y，即从左到右放置 x，满了一行后 y+1。

让深入分析下_ok_grid 中三个控件的宽度。计算标称尺寸时，三个控件宽度分别是 136、40（标称宽度=配置宽度(20) x hdpi_scale(2)）、136，相加结果是 312，小于 label 控件的 552。根据“同一列内格子必相同宽度”，分给_ok_grid 的宽度是 552。但该网格设置了居中对齐，于是内中三个控件都不会放大，多出的 240(552-312)留给两边。

5.2.2 控件尺寸的弹性

网式布局模型的一个特点在于控件的尺寸是弹性的。网格可以根据本身尺寸的大小来动态地调整控件尺寸。当网格中有空白空间时，控件可以扩展尺寸以占据额外的空白空间。控件尺寸的弹性由一个叫做拉伸系数的属性来确定：grow_factor。在图形化界面中，grow_factor 显示在每行/列最前面的格子，具体是行/列标下面的是那个数值，示例中该值都是 0。

grow_factor 属性的值是一个没有单位的非负数，默认值是 0。它的值表示的是当网格有多余的空间时，这些空间在不同控件之间的分配比例。比如，一个网格中有 3 个控件，其 grow_factor 属性的值分别为 1，2 和 3。每个控件的标称宽度均为 200px。当网格的宽度超过 600px 时，比如变成了 660px 之后，则需要放大的尺寸 60px 由 3 个控件按照比例来分配。3 个控件分别要放大 10px、20px 和 30px，于是渲染尺寸分别变为 210px、220px 和 230px。

控件拉伸多少和拉伸系数有关，和控件标称尺寸无关。

到这里，做过网页布局的人会感觉似曾相识，是的，这里的拉伸系数基本就等同 CSS3 弹性盒布局中的“flex-grow”。不同的是，网式布局不会缩小，因而不存在“flex-shrink”。

5.2.3 控件对齐

当控件的渲染尺寸确定之后，一旦发现渲染高度超过标称高度，这时问题就来了，它在垂直上该怎么放，是上对齐？居中？下对齐？还是两端对齐？水平上也存在同样问题。

控件对齐由两个属性来决定，horizontal_alignment 设置水平对齐、vertical_alignment 设置垂直对齐。它们都有四个值，左/上对齐、居中和右/下对齐、两端对齐。在图形化界面中，控件、网格所在格子的右上角两个小图标分别表示了 horizontal_alignment、vertical_alignment。

5.2.4 通过同列控件计算最大正文宽度

label 控件存放着字符串，字符串需要的空间会随着容许宽度不同而不同。举个例子，有这么个字符串，一行显示时需要 640 像素，给它 480 像素时，须要两行，给它 640 时，只需要一行。根据容许尺寸的不同来源，有三种方法实现多行文本。注：表中 mtwusc 是 calculate Maximum Text Width Using Same Column control 缩写。

使用场景	width	width_is_max	mtwusc	实例
写脚本时就知道该控件的标称宽度，它将作为容许宽度	设置	false	false	
写脚本时知道该控件的最大标称宽度，它将作为容许宽度，最终标称宽度会因为内容而变小	设置	true	false	simple_message
由其它控件算出容许尺寸	留空	false	true	

label 控件的 height 必须留空。前两种都要给控件设置 width，实际使用中更多的是要由其它控件决定容许尺寸，即 mtwuse=yes 时的第三种情况。

“同一列内格子必相同宽度”，网式布局利用这条规则去布局可能多行的文本控件。换句话说，通过同一列的其它格子计算出宽度，然后把这个宽度做为自个容许宽度。基于这个原因，布局逻辑在第一次计算该控件的标称尺寸时，总是返回(0, 0)，它要到后面才真正确定尺寸，像放置阶段。在图形化布局时，用以下步骤放置可能多行的文本控件。

1）放置文本控件。打开“标称尺寸”页，使能“mtwuse”。

2）确保有同列的其它控件去计算宽度。

3）如何设置对齐？水平方向、垂直方向都不能用两端对齐。当内容不足一行时，或左对齐、或中央对齐、或右对齐就会影响字符串位置。当所在行有其它控件的高度大于它时，或上对齐、或中央对齐、或下对齐就会影响字符串位置。

一旦使能 mtwuse，控件会受到些限制（Studio 会执行这些检查）。1）不能出现在树形节点。2）不能设置等尺寸组。3）所在行存在多列时，该列放大系数不能是 0。4）滚动面板的孩子有

多行文本时，必须把 width 留空（height 通常置 0），这是为了一定计算内中 mtwuse 控件的标称尺寸。

一些 gui 有个和多行文本相关的控件叫滚动文本框（scroll_label），它是右侧可能会出现垂直滚动条的多行文本。为减少种类，Rose 没有做 scroll_label，也不做成模板。要实现 scroll_label，通常做法是放一个 scroll_panel，然后内部一个 label。

5.2.5 减少重布局次数

对含有字符串的控件，像 label、button，运行时修改字符串内容后，一旦前后出现尺寸发生变化，极可能导致需要重布局窗口。重布局较耗 cpu。

Rose 判断是否要重布局，依据是**后面标称尺寸和已有渲染尺寸**差值，任何一个方向标称尺寸超过渲染尺寸，或差值的绝对值超过 font_size*2/3，认为要重布局。

满足一些条件时，该方向会不触发重布局。width 方向。1）mtwusc 是 true。2)对齐方式是边沿对齐。3）标称宽度设了固定值或公式。height 方向，除了没 mtwusc，其它类似 width 方向。为什么设条件的原因，设了边沿对齐或固定值的标称宽度时，rose 认为设定置已会使控件美观。

控件是 mtwusc 时，尽量让每次都是相同行数。只要行数相同，就基本不会导致重布局。

控件不是 mtwusc 时。1）对于确信“知道”会出现多少字符情况时，用边沿对齐或固定值的标称尺寸。2）使用自定义最小宽度，让可能出现的宽度不超过这个宽度，这样保证前后算出的标称宽度一样。让不多出现的超长字符时，会触发重布局。

5.2.6 等尺寸组（Linked_group）

图 5-2 等尺寸组

图 5-2 中按钮由于不同的文字内容，要是不做任何处理，按钮宽度会不一样。为美观，如何让这四个按钮同一宽度？这时可使用等尺寸组。具体做法分两步，1)定义一个等宽的等尺寸组，假设 id 是 field。2）把这四个按钮的等尺寸组字段设为“field”。

等尺寸组让组内控件保持同一尺寸，靠的是统一它们的标称尺寸，具体值是它们中最大的标称尺寸。由于界面看到的是渲染尺寸，为保证看到的尺寸一致，同时应该保证组内控件拉伸量一致，最简单做法自然是不让它们有拉伸。

等尺寸组有三类归属者。第一类列表控件（包括表头、表体），每个列表自成一套。第二类是树形控件，每个树形自成一套。除了列表、树形外，其它都属于窗口。

对列表，为让列和列之间有间隔，常会在列控件设置左/右 border。为保证对齐，一旦表体中设了 border，表头相应位置的控件也要做同样 border 设置。等尺寸靠的是统一组内控件的标称尺寸，标称尺寸不包括 border（tgrid 中的 col_width_/row_height_除外）。

5.2.7 悬浮控件

网式布局不会让各控件间出现空洞，同时也不允许重叠。虽然栈层控件内部可让支持重叠，但欠缺灵活。悬浮控件就是让灵活地实现重叠，使用实例像提示，垂直、水平滚动条。

在内容上，悬浮控件只能是原子控件，包括 button、image、label、slider、text_box、toggle_button、track。

悬浮控件的一个重要任务是要确定渲染矩形。计算渲染矩形可使用下表给出的自变量。

自变量	描述
mobile	true，移动平台。false，PC
svga	屏幕尺寸不小于 800x600
vga	屏幕尺寸不小于 640x480
ref_width	参考控件的渲染宽度

ref_height	参考控件的渲染高度
hdpi_scale	hdpi 倍数
width	悬浮控件宽度。计算出尺寸后填的值，只能用在算左上角坐标
height	悬浮控件高度。计算出尺寸后填的值，只能用在算左上角坐标

表示宽度、高度的自变量值都已经考虑了 hdpi_scale。原因是会存在不是 hdpi_scale 整数倍的控件，像 hdpi_scale 是 2 时，也会存在高度是 1px 的图像控件。

确定渲染矩形依次包括计算尺寸和左上角坐标。

尺寸。每个悬浮控件必须有一个参考控件。1）参考控件可以是窗口；2）要能显示这个悬浮控件，参考控件必须是在显示状态。3）有两种方法指示参考控件，一是在代码中调用 set_ref_widget 指定 ref_widget_，二是在窗口脚本用控件 id，当两个都有效时，ref_widget_比 id 优先级高。ref_width、ref_height 分别表示参考矩形的宽度、高度。

```
width/height = (用公式表示的部分) + (用常数表示的部分) * hdpi_scale
```

上面是计算尺寸的公式，算出尺寸后，自变量 width、height 表示了这个尺寸。

左上角坐标。存在两种方法确定左上角坐标，一是用参考矩形，二是用固定点，set_ref_widget 的第二个参数指定这个固定点。不论是哪种方法，都有以下公式计算左上角坐标。

```
x/y = ref_x/ref_y + (用公式表示的部分) + (用常数表示的部分) * hdpi_scale
```

其中的 ref_x/ref_y，用参考矩形指示的参考矩形的左上角，用固定点是当中的 x、y。

5.3 C/C++代码

5.3.1 设计要实现目标

自适应各种分辨率

作为程序一般要求，弹出的窗口不能越出主题界线，由于主题尺寸各式各样，就要求窗口必须跟着缩放尺寸。对主题尺寸，iPhone/iPad 上是 960x640/2048x1536，PC 上可能是 1280x800、800x600，甚至可能是用户随意缩放出的一个尺寸，一个窗口要在这么多种分辨率下显示，即使内容简单，它也要根据那个分辨率下尺寸来调整内部显示。像字号，1280x800 下会比 480x320 中的大；当内容复杂时，调整的不仅仅得是字号，还要调整控件间隔等这些内容。编写出窗口模板必须要能支持各样主题分辨率。

自动布局控件

在窗口配置中会写此个窗口要用到哪些控件，但为“同时”支持各种窗口分辨率，往往不会写控件左上角坐标是多少、尺寸是多少，这时就要求系统能够自动布局窗口中控件。要实现自动布局，它必须要有一定依据，这依据分摊在两种对象，一是布局的管理者：窗口，二是被布局对象：控件。对窗口来说，布局依据可以是要不要把窗口覆盖整主题？当不覆盖时是居中放置还是按坐标放置？对控件来说，它会指示当可放置尺寸超过它基本尺寸时要不要放大？当一“行”/“列”中存在多个可放大控件时，各控件的放大系数分别是多少？

容易扩展控件

控件是组成窗口基本单元，即使现在写的系统再完善，也不能肯定把将来需要都写了，设计窗口系统时就要考虑到能容易扩展控件。扩展的一个要求是新增控件，对新增控件的代码一般分两部分：通用代码和私有代码。通用代码指的是所有控件都会用到的代码，像注册、加载，构建，这部分代码往往已经完备，新增控件这个目的来说只要理解这部分代码；私有代码指的是该控件专属代码，新增控件时要写的主要是这部分代码。要达到容易扩展控件这个目标，就必须定义“好”通用代码和私有代码分工，即通用代码和私有代码之间的 API。

5.3.2 注册控件模板

- 控件模板：控件各式各样，有按钮控件、图像控件、标签控件，等等，由于在实现上一种控件被分为多种风格（一种风格称为一种控件定义），所以它更多成了个讨论中概念，不论是 WML 还是 C/C++代码，它都不对应某个具体存在。

- 控件定义模板：在 WML 中，对应一个控件定义块，例如一个[button_definition]块对应一个按钮定义模板；在 C/C++中，对应从 tcontrol_definition 派生的类对象，例如 tbutton_definition 对应按钮定义模板。

C/C++处理控件，第一步是向 C/C++系统注册它支持的那些个控件模板。注册，和具体哪个控件无关，它注册的是控件模板。注册经常被称为静态注册，“静态”指出注册时间，注册发生在类似 C 语言中静态变量的赋初值时机，它在程序 main 函数之前被执行。具体看图 6-3 中的 Call Stack，register_widget 执行注册控件按钮模板，在它执行时还没到 C/C++主函数 main/WinMain。

知道是什么时候注册，那注册都注册了些什么内容？

注册要注册两类函数，一是控件的加载函数，二是控件的构建函数。不论哪种函数，函数都被加入到射映，射映项名就是控件标识，像 button，drawing。

加载函数

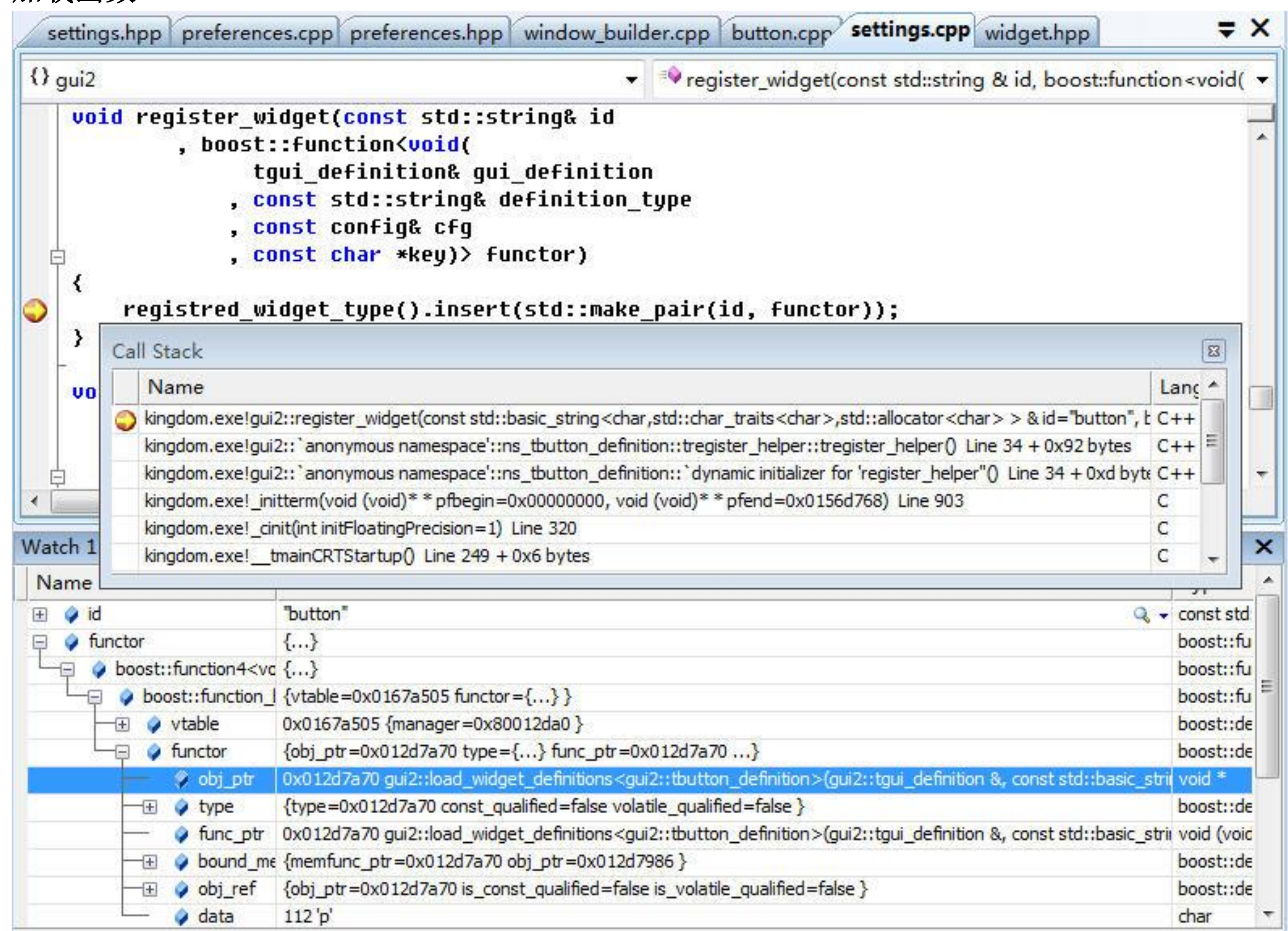


图 6-2 gui2::register_widget

register_widget 用于注册加载函数。

- id：要注册的控件模板标识。
- functor：加载时，用于加载该模板的加载函数。

register_widget 是如何被调用的？让在 Call Stack 中跳到 gui2::anonymous namespace::ns_tbutton_definition::tregister_hepler::tregister_helper()，跳到的是这一个宏定义。

```
REGISTER_WIDGET(button)
```

找到这宏实现，

```
#define REGISTER_WIDGET(id) REGISTER_WIDGET3(t##id##_definition, id, _4)
```

把 button 代入 id，REGISTER_WIDGET(button)执行宏展开。

```
REGISTER_WIDGET3(tbutton_definition, button, _4)
```

这时再进入 REGISTER_WIDGET3 宏定现。

```
    #define REGISTER_WIDGET3(  \
          type \
        , id   \
        , key) \
namespace {    \
               \
    namespace ns_##type {   \
        \
        struct tregister_helper {  \
            tregister_helper()  \
            {  \
                register_widget(#id, boost::bind( \
                        load_widget_definitions<type>   \
                      , _1   \
                      , _2   \
                      , _3   \
                      , key));  \
        \
                register_builder_widget(#id, boost::bind(   \
                        build_widget<implementation::tbuilder_##id>  \
                        , _1));   \
            }  \
        };  \
            \
        static tregister_helper register_helper;\
    }  \
}
```

宏 REGISTER_WIDGET3(tbutton_definition, button, _4)内会执行 gui2::anonymous “static tregister_helper register_helper”语句，后者定义一个静态变量 register_helper（每控件模板都有一个 register_helper，它位在每控件模板专属的名字空间，像按钮模板是“ns_tbutton_definition”)，正是这条语句执行了真正的注册过程。那这条语句执行什么内容？——register_helper 类型是 struct，一旦它的变量被定义，将自动调用 struct 构造函数 tregister_helper::tregister_helper，也就是说，构造函数中执行内容就是 static tregister_helper register_helper 这条语句执行内容。

- register_widget：注册加载函数。
- register_builder_widget：注册构建函数。

register_widget 的两个参数是什么值。第一个参数是#id，对于按钮模板代入 id 后就是字符串“button”，第二个参数是用 boost::bind 定义的函数，函数名是 load_widget_definitions，因为用了<type>修饰，这是个模板函数，模板参数是<type>，具体到“button”控件模板就是 tbutton_definition。boost::bind 的第二、第三、第四、第五参数依次是_1、_2、_3、key，二、三、四参数是占位符，代表这个位置有参数，但具体是什么值得函数被调时才能确定，_1 指示使用被调时参数列表中的第一个位置上的参数值，依此类推。

顺便分析下 Call Stack 中 namespace::ns_tbutton_definition::tregister_hepler::tregister_helper()这字符串是如何形成的。REGISTER_WIDGET3 是在名字空间 gui2 中定义的，第一名字空间是 gui2。“namespace {”定义一个匿名名字空间，第二名字空间是 anonymous namespace。“namespace ns_##type，type 是宏参数，代入值 tbutton_definition 后，第三名字空间是 ns_tbutton_definition。余下 tregister_hepler::tregister_helper()较好理解，指的是 tregister_hepler 这个 struct 构造函数。

到现在已知道针对按钮控件时 register_widget 中的 id 值和 functor 是什么值，以及是如何来的。接下让看 register_widget 干了些什么，register_widget 就是执行一条语句。

```
registred_widget_type().insert(std::make_pair(id, functor));
```

该语句向加载函数映射增加一<id, functor>项。registred_widget_type()则是得到此个映射。

```
static tregistered_widget_type& registred_widget_type()
{
    static tregistered_widget_type result;
    return result;
```

```
}
```

注册是形成一<id, functor>对，然后把它加入 registred_widget_type 映射。但它只是加入，不执行真正的加载过程。执行真正的加载过程是在 gui2::init，它在 do_gameloop 准备好 SDL 库后立即被调用，参考“6.3.2 加载”。

构建函数

tregister_helper::tregister_helper()中执行的 register_builder_widget 就用于注册构建函数。

register_builder_widget 的两个参数是什么值。第一个参数是#id，对于按钮模板代入 id 后就是字符串 button，第二个参数是用 boost::bind 定义的函数，函数名是 build_widget，因为用了<type>修饰，这是个模板函数，模板参数是<type>，具体到“button”控件模板就是 implementation::tbuilder_button。boost::bind 的第二参数是_1，这个参数是占位符，代表这个位置有参数，但具体是什么值得函数被调时才能确定，_1 指示使用被调时参数列表中的第一个位置上的参数值。

注册是形成<id, functor>对，然后把它加入 builder_widget_lookup 映射。和加载函数一样，它只是加入，不执行真正的构建过程。

总结下 REGISTER_WIDGET3 宏中的三个参数。

- type：字符串。用于通用加载函数 load_widget_definitions 的模板参数。实例值：tbutton_definition。
- id：字符串。模板标识。用于 1）registred_widget_type/builder_widget_lookup 标识这个加载/构建是哪个模板；2）用于形成通用构建函数 build_widget 的模板参数。
- key：字符串。用于通用加载函数 load_widget_definitions 的第五个参数，占位符_4。

REGISTER_WIDGET 有个缺陷，就是它的 t<control_id>_definition 和 id 必须相同字符串，要想不一样就须要直接使用 REGISTER_WIDGET3 宏。

```
REGISTER_WIDGET3(tlistbox_definition, horizontal_listbox, _4)，(listbox.cpp)
REGISTER_WIDGET3(ttext_box_definition, password_box, "text_box_definition")，(password_box.cpp)
```

ttext_box_definition 是唯一一个_4 不是 NULL 的类。

注册小结

- 注册在程序 main 函数之前被执行。
- 注册是针对每种控件模板形成一个<id, functor>映射项，然后把这映射项加到加载/构建函数映射。过程中不执行真正的加载/构建操作。
- 所有控件模板使用的是同一个加载函数（load_widget_definitions），同一个构建函数（build_widget）。
- 控件实现类要实现注册只要定义个 REGISTER_WIDGET/REGISTER_WIDGET3 宏。

5.3.3 加载控件模板

加载把具体 WML 控件定义放入内存。

之前在说注册时已提到要注册加载函数，而所有控件的通用加载函数是 load_widget_definitions，很容易联想到加载过程是不是等同调用 load_widget_definitions？——的确，加载过程就是为每种注册过的控件模板调用 load_widget_definitions。仅知道要调用 load_widget_definitions 是不够的，像对这过程得知道调用 load_widget_definitions 时参数从哪来？要知道参数从哪来，让进入 load_widget_definitions 源码级调试，看当时程序运行情况。

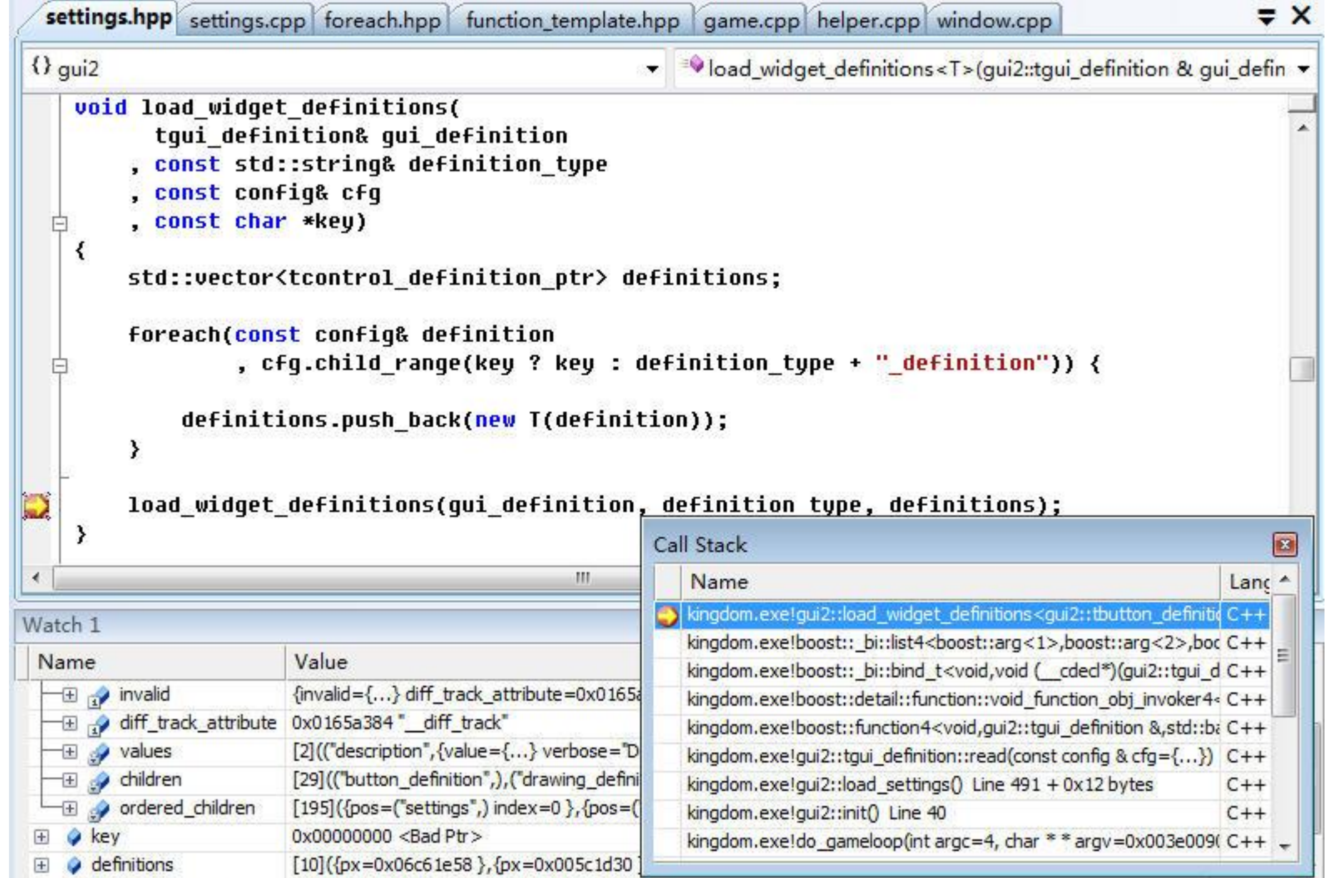


图 6-3 gui2::load_widget_definitions

- gui_definition：[OUT]。C/C++代码负责管理窗口的对象。形成的数据要成为它的成员变量值，因而认为这参数是[OUT]。
- definition_type：[IN]。模板 id，像按钮就是 button。
- cfg：[IN]。加载数据源。来自于资源包<data>/gui 下的那些个资源文件，即[gui]块。对每一次 load_widget_definitions 调用这个值都是一样的。具体一个实例参考图 6-4。
- key：[IN]。当它非 NULL 时，用于在形成控件定义标识时代替 definition_type。当前这个值都是 NULL。

Watch 1

Name	Value	Type
cfg	{invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2](("description",{value={...} verbose="D	const con
invalid	{invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[0]() ...}	config
diff_track_attribute	0x0165a384 "__diff_track"	const cha
values	[2](("description",{value={...} verbose="Default" }),("id",{value={...} verbose="default" }))	std::map·
[0]	("description",{value={...} verbose="Default" })	std::pair<
[1]	("id",{value={...} verbose="default" })	std::pair<
children	[29](("button_definition",),("drawing_definition",[1](0x0695a510 {invalid={...} diff_track_attribute=0x0165a384	std::map
[0]	("button_definition",)	std::pair<
first	"button_definition"	const std
second	[10](0x069144c8 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2](("description",{value=	std::vect
[0]	0x069144c8 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2](("description",{value={...}	config *
[1]	0x0691a288 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2](("description",{value={...}	config *
[2]	0x069202c8 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2](("description",{value={...}	config *
[3]	0x06926320 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2](("description",{value={...}	config *
[4]	0x0692c368 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2](("description",{value={...}	config *
[5]	0x069324b8 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2](("description",{value={...}	config *
[6]	0x069385c0 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2](("description",{value={...}	config *
[7]	0x0693e760 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2](("description",{value={...}	config *
[8]	0x069448c8 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2](("description",{value={...}	config *
[9]	0x0694f680 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2](("description",{value={...}	config *
[1]	("drawing_definition",[1](0x0695a510 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2]((	std::pair<
[2]	("horizontal_listbox_definition",[1](0x069cfae8 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" valu	std::pair<
[3]	("horizontal_scrollbar_definition",)	std::pair<
[4]	("image_definition",[1](0x06982698 {invalid={...} diff_track_attribute=0x0165a384 "__diff_track" values=[2](("d	std::pair<

Locals　Watch 1

图 6-4 gui2::load_widget_definitions-cfg

控件定义标识：用于标识这个定义是属于哪种控件。控件定义标识的值是字符串，它的格式：[key]+_definition，key 往往就是控件标识，像按钮的控件定义标识是 button_definition；标签是 label_definition。控件定义标识用于标识某个控件定义块，后者则是 WML 和 C/C++程序之间桥梁，**一个 WML 中的控件定义块被转换为一个 C/C++中的从 gui2::tcontrol_definition 派生的对象**。把这个转换扩大来说，加载过程就是从 WML 中读出控件定义块，把它转换为相应的 gui2::tcontrol_definition 对象，生成的对象放进窗口管理对象 gui_definition。

既有控件标识，干吗还要控件定义标识？——在命名上，控件定义标识肯定以"_definition"为后缀，这会让 WML 定义看起来更统一化，但这是次要的，主要还是设计时要让每一控件标识下可以存在多个控件定义块，每个控件定义块则定义了该控件的一种风格！让举个按钮控件例子，常见按钮背景是不透明图像，如果我们想再定义一种背景是透明图像按钮，这时在 WML 中就可定义两个[button_definition]块，以当中 id 值来区分是常见按钮（id=default）还是透明风格按钮（id=transparent），同样的如果要增加滚动条上端按钮，它没有字符只显示一个向上箭头，则可定义一个 id=up_arrow 的[button_definition]块。

```
[button_definition]
    id = "default"
    description = "Default button"
    [resolution]
    ......
    [/resolution]

[/button_definition]
[button_definition]
    id = "transparent"
    description = "Default transparent button"
    [resolution]
    ......
    [/resolution]
[/button_definition]
[button_definition]
    id =  "up_arrow"
    description = "Up arrow button for a scrollbar."
    [resolution]
```

```
    ......
    [/resolution]
[/button_definition]
```

有了控件标识和控件定义标识这种上下级关系，加载时执行“一个 WML 中的控件定义块被转换为一个 C/C++中的从 gui2::tcontrol_definition 派生的对象”，但对同一个控件标识下的控件定义块，它转换到的是同一种 gui2::tcontrol_definition 对象！它减少了 C/C++要实现的 gui2::tcontrol_definition 类数目。

图 6-4 是一次[gui]块截图，图中指示按钮控件底下有 10 个按钮控件定义块[button_definition]，也就是说编写 WML 代码时可使用 10 种风格的按钮。图像控件则只有一个控件定义块[image_definition]，也就是说在编写 WML 代码时可使用一种风格的图像控件，以下会说到当一控件下只有一个控件定义块时，定义块 id 必须叫“default”。

区分控件标识和控件定义标识，知道一控件下可存在多个控件定义块，就能容易理解图 6-3 中 load_widget_definitions 中的那个 foreach 循环，它的作用就是针对每个控件定义块生成相应的 tcontrol_definition 对象。对这 foreach 循环，还须要明确模板参数 T 是怎么来的，虽然从图 6-4 中已能猜到它应该是 tcontrol_definition 对象的指针类型。

叙述注册加载函数时说过所有控件的加载函数统一是 load_widget_definitions，那对每个控件它们除了不同标识，还有什么是不同的？——REGISTER_WIDGET3 中，参数 type 表示的传给 load_widget_definitions 的模板参数，也就是这个 T！按钮就是 tbutton_defintion，标签就是 tlabel_definition。

到此已知道 load_widget_definitions 把一个控件标识下的所有控件定义块逐个转成为 tcontrol_definition 对象，但这些对象是放在 definitions 这个栈变量，只要函数一退出，栈变量就要被释放，因而必须把这变量内容赋到一个“静态”变量中去，这就是 load_widget_definitions 下半部 load_widget_definitions(gui_definition, definition_type, definitions)要干的事，虽然它也叫 load_widget_definitions，但的确是两个函数。

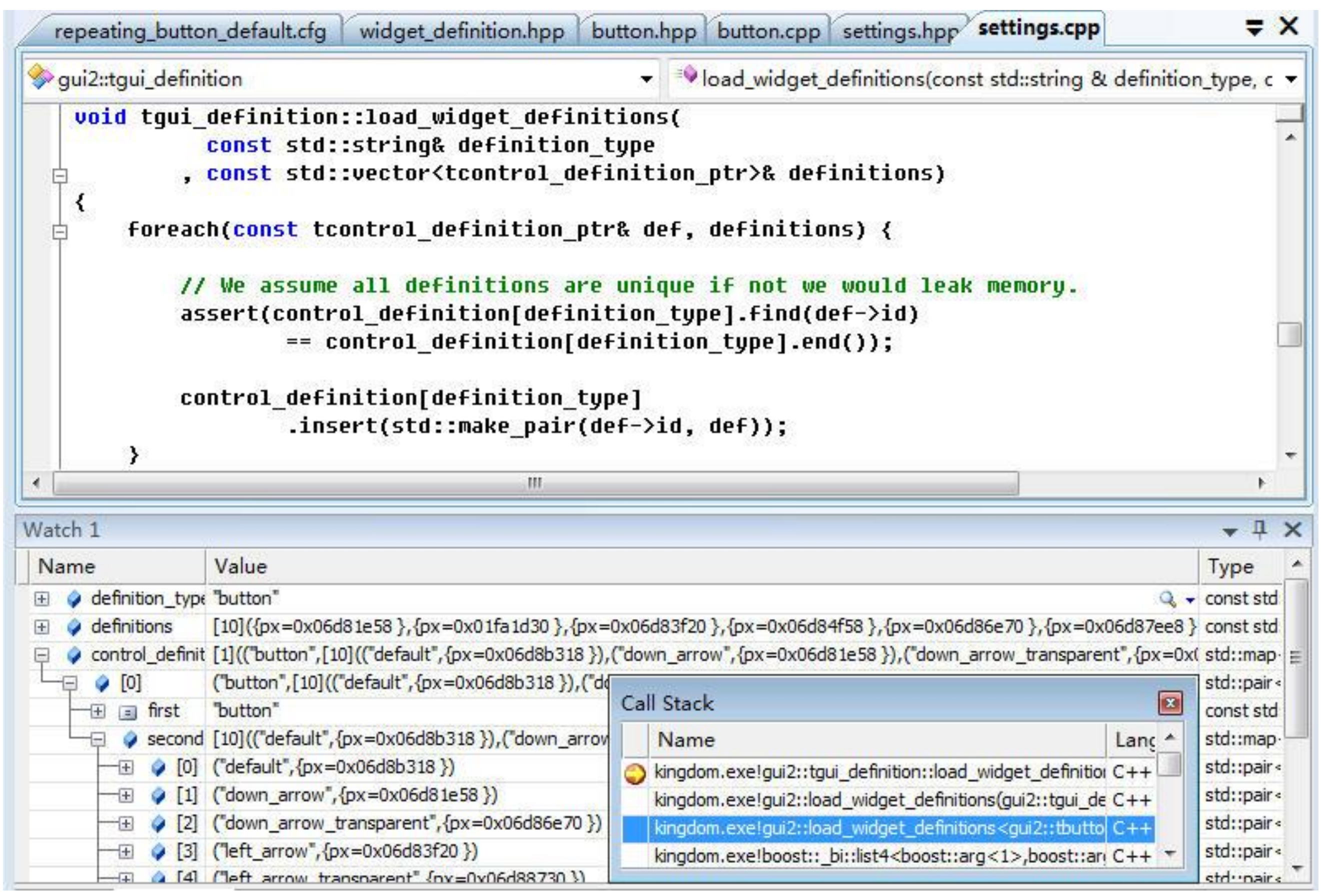


图 6-5 gui2::load_widget_definitions2

它是第三个名叫 load_widget_definitions 的函数，它是 tgui_definition 的成员函数，tgui_definition 对象就是图 6-3 中第一个参数 gui_definition。此个 load_widget_definitions 中两个参数 defintion_type、definitions 则直接来自上层函数，以下是按钮控件时的值。

- definition_type：button；
- definitions：把 [gui] 中 [button_definition] 控件定义块逐个转成的 std::vector<tbutton_definition_ptr>。

图 6-5 中 foreach 循环执行就是把栈变量 definitions 中数据赋给类型是双 std::map 的“静态”变量 tgui_defintion::control_definition。

```
struct tgui_definition
{
   ......
   typedef std::map <std::string /*control type*/,
      std::map<std::string /*id*/, tcontrol_definition_ptr> >
      tcontrol_definition_map;
   tcontrol_definition_map control_definition;
   ......
};
```

图 6-5 显示的只是 load_widget_definitions 上半部代码，该函数后续代码会检查在此个控件的控件定义块集合中是否有个叫 id=default 的，没有就报错，也就是说当一控件下只有一个控件定义块时，定义块 id 必须叫“default”。

控件配置：[resolution]块、gui2::tresolution_definition_

书写 WML 时一个控件定义块（[<id>+_definition]）下肯定有一个或多个[resolution]块，这些个[resolution]指示该控件定义块在外界不同分辨率时如何显示自己。针对当前可能存在设备（PC、平板、手机等等），控件一般需支持 800x600、640x480、320x480 三种“标准”[resolution]。也就是说，当外界分辨率是 800x600 或更大时，它采用 800x600 的那套去显示；当外界分辨率是 800x480，即界于 640x480 和 800x600 之间时，用 640x480 那一套；当小于 640x480 时，则采用 320x480。在具体显示上，针对要显示的同一内容，为让更小的面积却依旧能显示，不同[resolution]往往会有不同显示效果，像要显示一字符串，800x600 会使用 12 号字，640x480 会使用 10 号字，480x320 则是 8 号字。**多个[resolution]保证了该控件定义块在不同外界分辨时都能显示出控件需要表达的内容。**

[resolution]是 WML 中概念，它对应到 C/C++代码是“struct gui2::tresolution_definition_”，即 **gui2::tresolution_definition_对应控件定义块（[<id>+_definition]）中的一个[resolution]块。**

通过分析图 6-3 代码可得出个结论：C/C++中的一个 t<id>+_definition 对象封装一个 WML 中的控件定义块（[<id>+_definition]）。既然“封装了该块”，那应该存在个和[<id>+_definition]块对应的成员变量。基于[<id>+_definition]是个标准 WML 块，很容易让想到这个成员变量 C/C++类型是 config， 但<id>+_definition 没有用 config，而是用了处理[<id>+_definition]块后形成的 std::vector<tresolution_definition_ptr>，即单元类型是“gui2::tresolution_definition_”的 std::vector，该 std::vector 成员变量名则是 resolutions。

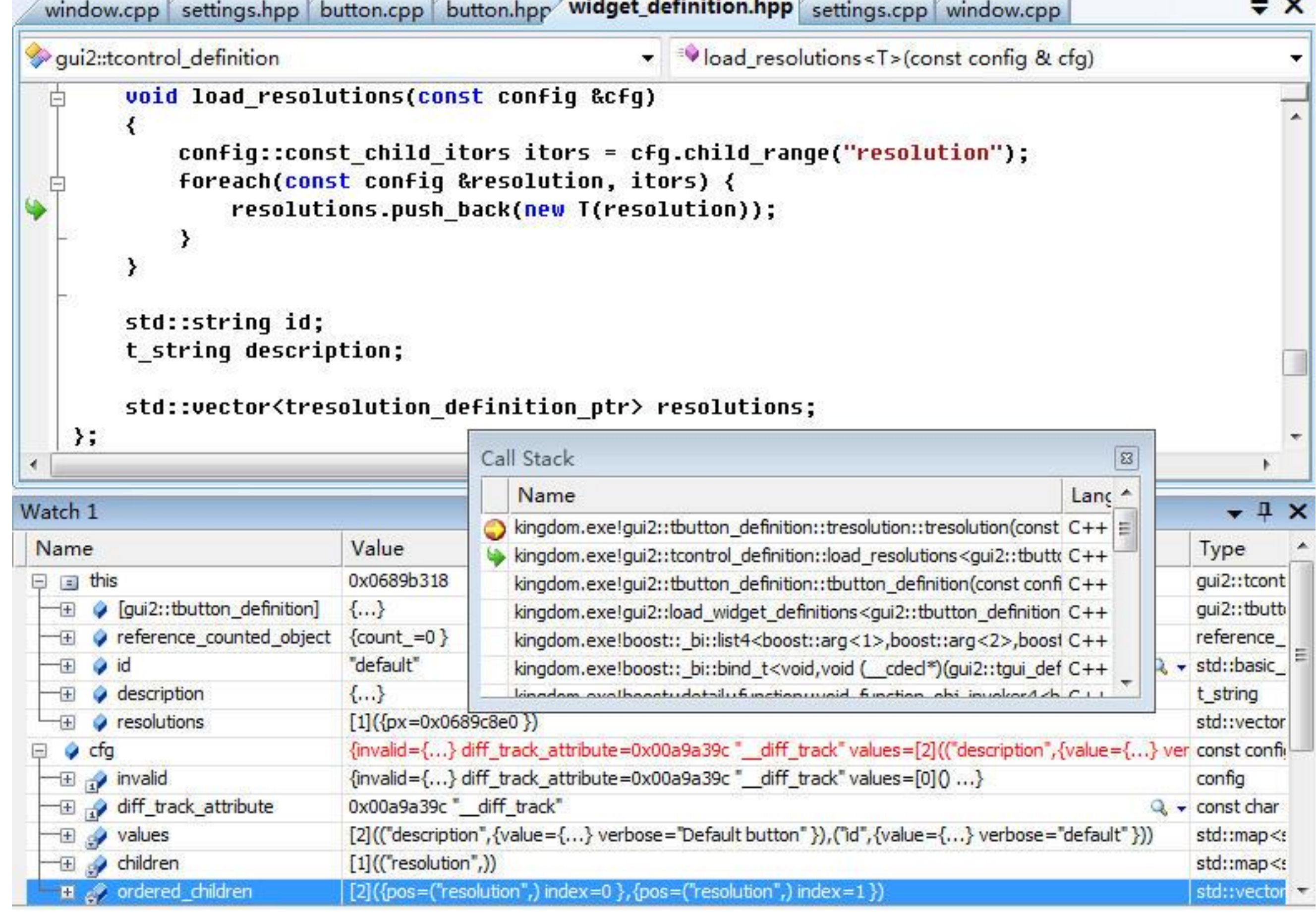


图 6-6 tbutton_definition::tresolution

- cfg：控件定义块[<id>+_definition]。

load_resolutions 是 gui2::tcontrol_definition 成员函数，执行把[<id>+_definition]块转换为 std::vector<gui2::tresolution_definition_>，生成的 gui2::tresolution_definition_ 放在成员变量 resolutions。gui2::tcontrol_definition 是所有控件定义类 gui2::<id>+_definition 的基类，因而控件定义类对象都有 resolutions 变量，作为一般做法，**控件定义类在它的构造函数中调用 load_resolutions 来“存储”对应的 WML 块。**

控件定义块下可能存在多个[resolution]，但具体到要用时只会选择当前分辨率下最合适的那一个，以代码要实用和规范化考虑，整系统最好能提供统一的查找最合适分辨率操作，即根据参数<控件标识，控件定义标识，当前分辨率尺寸>找出一个 gui2::tcontrol_definition。gui2::get_control 实现这个查找。

```
tresolution_definition_ptr get_control(
        const std::string& control_type, const std::string& definition)
{
    const tgui_definition::tcontrol_definition_map::const_iterator
    control_definition =
        current_gui->second.control_definition.find(control_type);

    std::map<std::string, tcontrol_definition_ptr>::const_iterator
        control  = control_definition->second.find(definition);

    for (std::vector<tresolution_definition_ptr>::const_iterator
        itor = (*control->second).resolutions.begin(),
        end = (*control->second).resolutions.end();
        itor != end;
        ++itor) {

        if (settings::screen_width <= (**itor).window_width ||
```

```
            settings::screen_height <= (**itor).window_height) {

            return *itor;
        } else if (itor == end - 1) {
            return *itor;
        }
    }
}
```

查找过程首先根据控件标识（control_type）在 control_defintion 的一级映射中找，找到<控件定义标识, 控件定义对象>。接下根据控件定义标识（definition）在得出的<控件定义标识, 控件定义对象>中搜出控件定义对象，最后根据当前分辨率找到“最合适”tresolution_definition_，并把它返回给调用程序。由以上代码可看出最合适[resolution]的判断标准。

顺序扫描[resolution]块预定义的分辨率（window_width、window_heigh 字段），找到**第一个大于或等于当前窗口尺寸**的分辨率。认为这个分辨率就是“最合适”分辨率。

如果第一步没找到，选择最后一个分辨率。

基于以上的判断标准，WML 书写[<id>+_definition]时它的[resolution]块必须按升序排列。

以下是两个调用实例。

```
get_control("window", "title_screen");
```

在当前分辨率下查找风格是 tile_screen 的窗体控件。

```
get_control("stack", "default");
```

在当前分辨率下查找风格是 default 的栈控件。

gui2::tresolution_definition_是静态的。静态指的是在运行过程中不管基于该模板构造了多少控件，那些控件具体数值是如何不同，gui2::tresolution_definition_中的值不会变。基于静态特点，加上 gui2::tresolution_definition_来自于 WML 中的标准配置块，它往往又被称作配置。控件实现类的基类 gui2::tcontrol 就把设置/读取 gui2::tresolution_definition_操作取名为 set_config(...)/config()，两函数使用参考“**图 6-21 load_config**”。

加载控件小结

- 加载过程就是从 WML 中读出控件定义块，把它转换为相应的 gui2::tcontrol_definition 对象，生成的结果放进窗口管理对象 tgui_defintion 中的成员变量 control_definition。
- tgui_defintion::control_definition 是个二级 std::map，一级 map 的 first 是控件标识，像 button、label，二级 map 的 first 是该控件标识下的控件定义标识，像 default、transparent、up_arrow；二级 map 的 second 则是由该控件标识块生成的 gui2::tcontrol_definition 对象。
- 当一控件下只有一个控件定义块时，定义块 id 必须叫“default”。

5.3.4 注册窗口模板

注：窗口模板不是窗体，窗体是控件，只是相比于按钮、图像，它有点特殊而已。

和注册控件模板一样，注册窗口模板也是发生在类似 C 语言中静态变量的赋初值时机，即在程序 main 函数之前被执行。具体看图 5-7 中的 Call Stack，register_window 执行注册 id=transient_message 的窗口，在它执行时还没到 C/C++主函数 main/WinMain。

注册主要执行，1：实现各个窗口实现类的 window_id 方法；2：把窗口标识加入到可支持窗口模板映射。

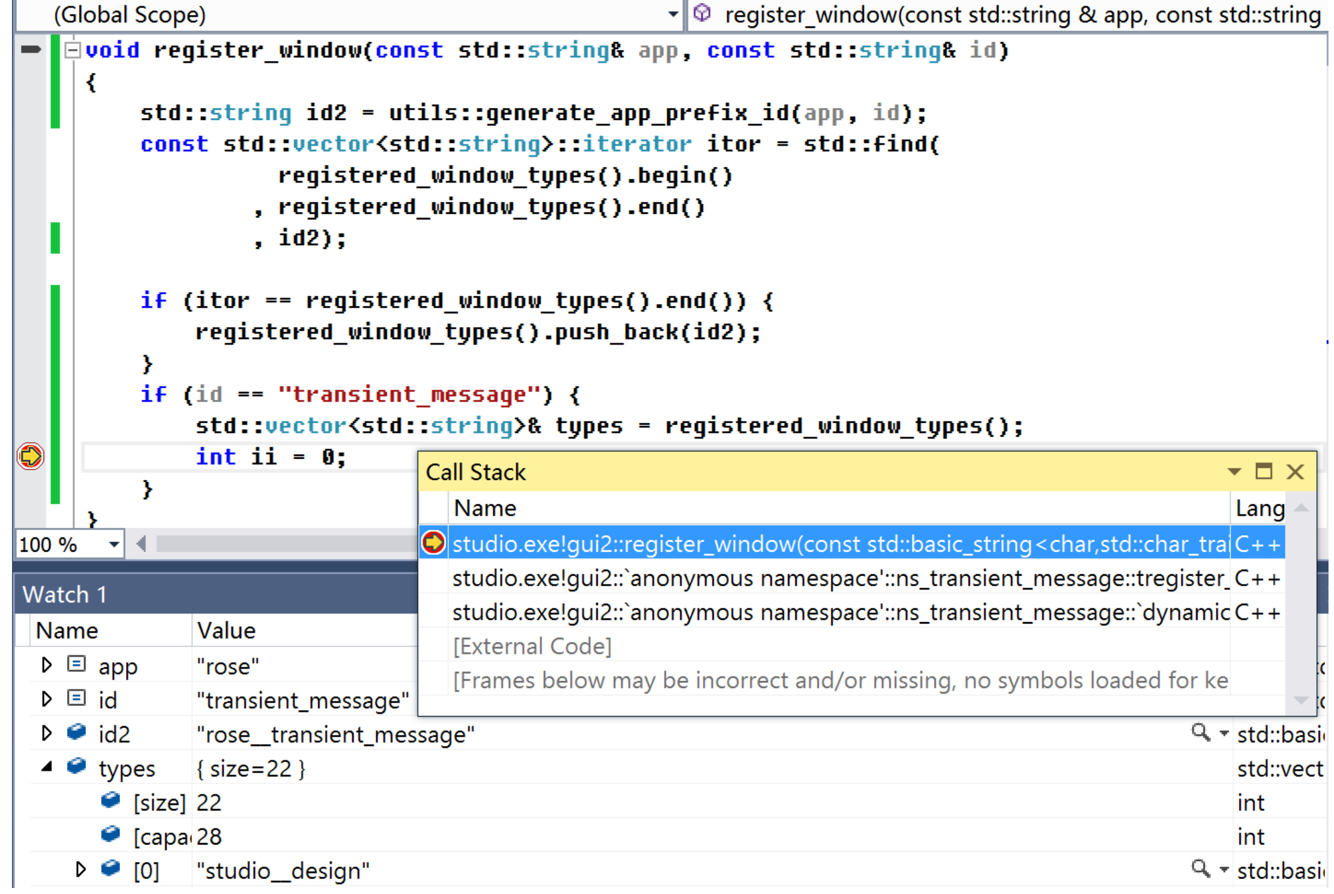


图 5-7 gui2::register_window

先说图中三个变量，“app”、“id”和“id2”。参数变量“app”、“id”共同指定了要注册哪个窗口。为什么要两个参数？因为 Rose 允许不同 app 可使用相同窗口 id。举个例子，你正在开发两个 app，它们都有主页窗口，这时你可以把两窗口的“id”都设为“home”。对 Rose 来说，它需要区分这两个“home”，这时靠的就是“app”。第三个变量“id2”指示了 Rose 最后标识窗口的值，格式上是用两个下横符连接“app”、“id”。

register_window 是如何被调用的？让在 Call Stack 中跳到 gui2::anonymous namespace::ns_transient_message::tregister_hepler::tregister_helper()，跳到的是这一个宏定义。

```
REGISTER_DIALOG(rose, transient_message)
```

找到 REGISTER_DIALOG 宏实现，

```
REGISTER_DIALOG(app,    window_id)    REGISTER_DIALOG2(t##window_id,    app,
window_id)
```

把 transient_message 代入 id，REGISTER_DIALOG 执行宏展开。

```
REGISTER_DIALOG2(ttransient_message, rose, transient_message)
```

找到 REGISTER_DIALOG2 宏定现。

```
#define REGISTER_DIALOG2(    \
         type   \
        , app   \
        , id)   \
    \
REGISTER_WINDOW(app, id) \
    \
const std::string&    \
    type::window_id() const  \
{   \
    static const std::string result(utils::generate_app_prefix_id(#app, #id); \
    return result;\
}
```

结合 REGISTER_DIALOG2(ttransient_message, rose, transient_message)，这段宏主要执行两个任务：1）调用宏 REGISTER_WINDOW(rose, transient_message)；2）实现函数 ttransient_message::window_id()。ttransient_message 被称为窗口实现类，在定义窗口实现类时只是声明 window_id()，REGISTER_DIALOG2 宏以统一方法实现了 window_id 操作。

让继续深入 REGISTER_WINDOW(rose, transient_message)。

```
#define REGISTER_WINDOW( \
        app
        , id) \
namespace {    \
    \
    namespace ns_##id {   \
    \
        struct tregister_helper {    \
            tregister_helper()    \
            {    \
                register_window(#app, #id); \
            }    \
        }; \
    \
        tregister_helper register_helper;   \
    }  \
}
```

这个定义看似有点熟悉，它在层次上很像用于注册控件模板的 REGISTER_WIDGET3，但它比 REGISTER_WIDGET3 简单多了，它只是调用 register_window，register_window 不存在 register_widget 中的加载函数！注册窗口模板不存在什么加载函数，注册过程只是把这窗口标识加入到一个指示系统可支持的窗口标识集。

为什么说 register_window 实现的只是把窗口标识加入到一个指示系统可支持的窗口标识集，回看图 6-6 register_window 实现，它说来就一条语句：

```
registered_window_types().push_back(id);
```

registered_window_types()和 registred_widget_type()一样返回的是一个全局数据结构，但它比 registred_widget_type 简单，它是一个单元项是字符串的 std::vector，后者可是个 std::map。为直观，图 6-7 加了些调试代码，让看到当系统注册了 transient_message 这个窗口模板后，已注册的窗口模板增至 47 个。

总结下 REGISTER_DIALOG2 宏中的三个参数。

- type：窗口实现类类名，它的命名格式是在“id”前缀上字符“t”。
- app：app 标识。它用于对窗口进行归类，从而让不同 app 可定义同 id 的窗口。值是“rose”时，它们是 Rose 内置窗口。
- id：窗口（模板）标识。它前缀上 t 字符就是对应的模板实现类类名。

注册小结

- 注册在程序 main 函数之前被执行。
- 注册主要执行两个任务，实现各个窗口模板实现类的 window_id 方法；把窗口模板标识加入到可支持窗口模板集。
- 窗口实现类要实现注册的操作就是定义个 REGISTER_DIALOG 宏。

5.3.5 加载窗口模板

加载控件模板时说到，加载控件模板过程就是从 WML 中读出控件定义块，把它转换为相应的 gui2::tcontrol_definition 对象，生成的结果放进窗口管理对象 tgui_defintion 中的成员变量 control_definition。加载窗口模板类似以上过程，只不过它在 WML 中加载源从控件定义块换成窗口（模板）块[window]，加载生成的 C/C++对象从 gui2::tcontrol_definition 换成 gui2::twindow_builder，用于存放生成结果的 tgui_defintion 成员变量从 control_definition 换成 window_types。要看加载具体过程让在 twindow_builder::read 内设断点。

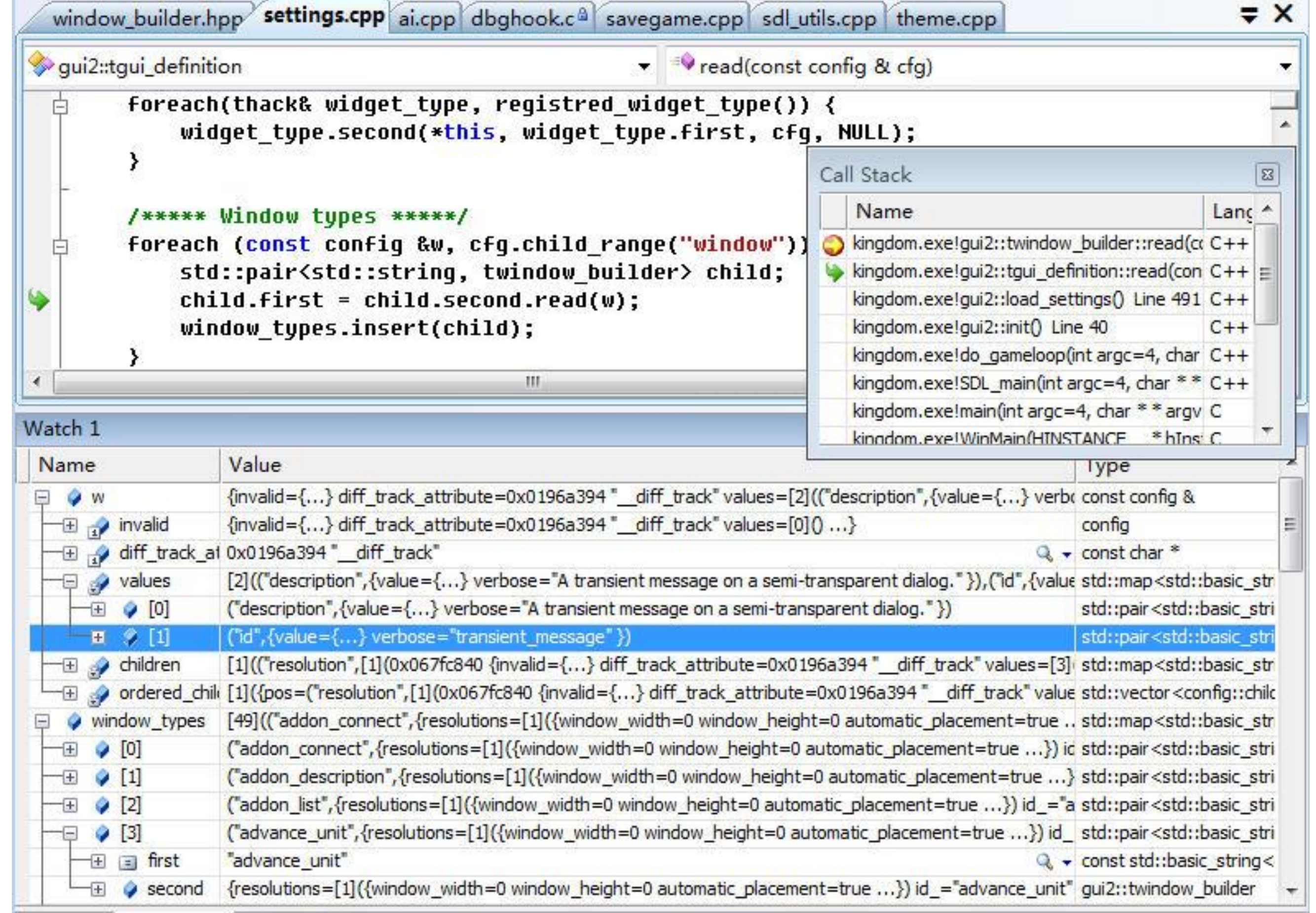


图 6-8 tgui_definition::read

图 6-8 显示 tgui_definition 加载窗口模板代码片断。cfg 就是 WML 中的[gui]块，对 foreach 每一次执行，它从[gui]块中取出一个窗口块[window]，使用 twindow_builder::read 操作把[window]转换为 twindow_builder 的内部数值（它等同把 WML 的[window]块转为 C/C++的 twindow_builder 对象），然后把生成的<id, twindow_builder>加入到 tgui_definition 成员变量 window_types。

具体图 6-8 中 Watch 窗口，它指示当 read 解析到 id=transient_message 这个窗口模板时情况，window_types 是个 std::map<std::string, twindow_builder>，first 就是窗口 id（[window]块中 id 字段）。C/C++代码解析到 id=transient_message 这个窗口模板时已从 WML 解析出 49 个[window]块，再加上 id=transient_message 则增至 50 个。回看图 6-7，“当系统注册了 transient_message 这个窗口模板后，已注册的窗口模板增至 47 个”，一个 50 一个 47，50 是已经从 WML 中解析出的[window]块数，这是由 WML 代码的编写顺序决定的；47 是 C/C++代码已注册的窗口模板数，这是由 C/C++代码编写顺序决定的。

此时会有个疑问，以上描述加载过程就没提到注册时生成的、用于描述 C/C++可支持的窗口标识集 registered_window_types，是不是加载过程不需要 registered_window_types？

registered_window_types 用在之后的有效性判断。由[window]块生成 twindow_builder，生成过程不须要借助相应的窗口实现类，但在具体窗口要被构造、被显示时则须要窗口实现类。可以想象，如果 C/C++中没有针对该 id 的窗口实现类，一旦程序要操作基于该标识的窗口，程序肯定非法。同样原因，定义了 C/C++实现类却没有在 WML 中相应[window]块（像忘了把该[window]块汇编进[gui]），一样会导致程序非法。因此加载过程要判断之前加载进的 [window]块是否有相应窗口实现类，对于代码，即执行图 6-8 的 foreach 循环后就去判断窗口有效性，它以 registered_window_types 中的 id 为源，逐个在 WML 中的[window]中找，找不到就报错，也就是说 WML 中[window]须是 registered_window_types 中 id 的父集。

窗口配置：[resolution]块、gui2::twindow_builder::tresolution

书写 WML 时，窗口模板[window]下肯定有一个或多个[resolution]块，这些个[resolution]指示该窗口模板在外界不同分辨率时如何显示自己。看到这里，让回看控件[resolution]块，当中也有类似内容，在功能上，此处的[resolution]定义的是窗口级别显示，控件[resolution]定义的是控件级别显示。对窗口级别来说，有像指定 800x600 时两内中控件垂直间距是 12 像素，640x480 时 10 像素，480x320 则 8 像素，等等。

让出现多个[resolution]是为该模板在不同外界分辨时都能显示要表达内容，多[resolution]机制有个副作用：**要求每[resolution]下都必须写上"全部"控件**，这意味着一旦多使用一个控件，那所有[resolution]都要"重抄"一遍。基于不希望重复写控件，实际使用中很少用多[resolution]来适应分辨率，而是采用公式来表示参数，用公式来计算某一分辨率时最后是什么值。

[resolution]是 WML 中概念，对应到 C/C++代码是"struct gui2::twindow_builder::tresolution"，即 **gui2::twindow_builder::tresolution 对应窗口模板[window]中的一个[resolution]块。**

类似于控件配置中存在查找函数"get_control"，系统也提供了根据标识查找窗口配置的函数"get_window_builder"，它在 window_types 这个映射找，找到 gui2::twindow_builder::tresolution 对象，并把它返回给调用程序。查找过程中分辨率判断也类似"get_control"。

gui2::twindow_builder::tresolution 是静态的。静态指的是在运行过程中不管基于该模板构造了多少窗口，那些窗口具体数值是如何不同，gui2::twindow_builder::tresolution 中的值不会变。基于静态特点，加上 gui2::twindow_builder::tresolution 来自于 WML 中的标准配置块，它往往又被称作配置。窗口实现类 twindow 中的"definition_"成员变量存储了该窗口基于的配置。顺便说下，twindow 基类 twidget 也有一个成员变量"definition_"，该变量用于指示控件风格，对窗口来说它存储的就是窗体风格。虽然同一个 twindow 对象内出现了两个"definition_"，但 twidget 把"definition_"声明为私有，所以两者不存在冲突。

网格（grid）

一直到现在都没讨论窗口如何组织控件。

控件按是否能包含其它控件分为**原子控件**和**容器控件**，原子控件如文本、按钮，容器控件如滚动面板、列表，在编写 WML 时，区分是原子控件还是容器控件的方法是该控件定义块中是否有[grid]块。以下是窗口组织控件规则。

- 必然而且只能包含一个窗体控件。
- 所有控件被组织成控件树状结构，树根是窗口模板定义的网格。树中容器控件是枝，原子控件是叶。除了控件，树状结构还会存在网格，网格肯定是枝。
- 网格不是控件，窗口模板、容器控件使用它来组织内中控件。网格组织控件方式是把所属区域以行、列拆分，形成一个内分好多矩形格子的网状结构，每个格子存放一个控件/网格。

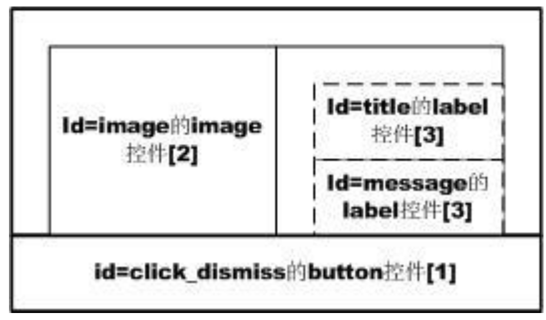


图 6-9 ttransient_message 窗口的网状结构

WML 中[grid]块表示一网格，那 C/C++代码是如何处理网格？在加载阶段，C/C++代码是把网格转换成 tbuilder_grid 对象。为清晰看到[grid]是如何变成 tbuilder_grid 让在 tbuilder_grid::tbuilder_grid 设断点，看 C/C++如何处理代码 6-1 那个顶层[grid]。

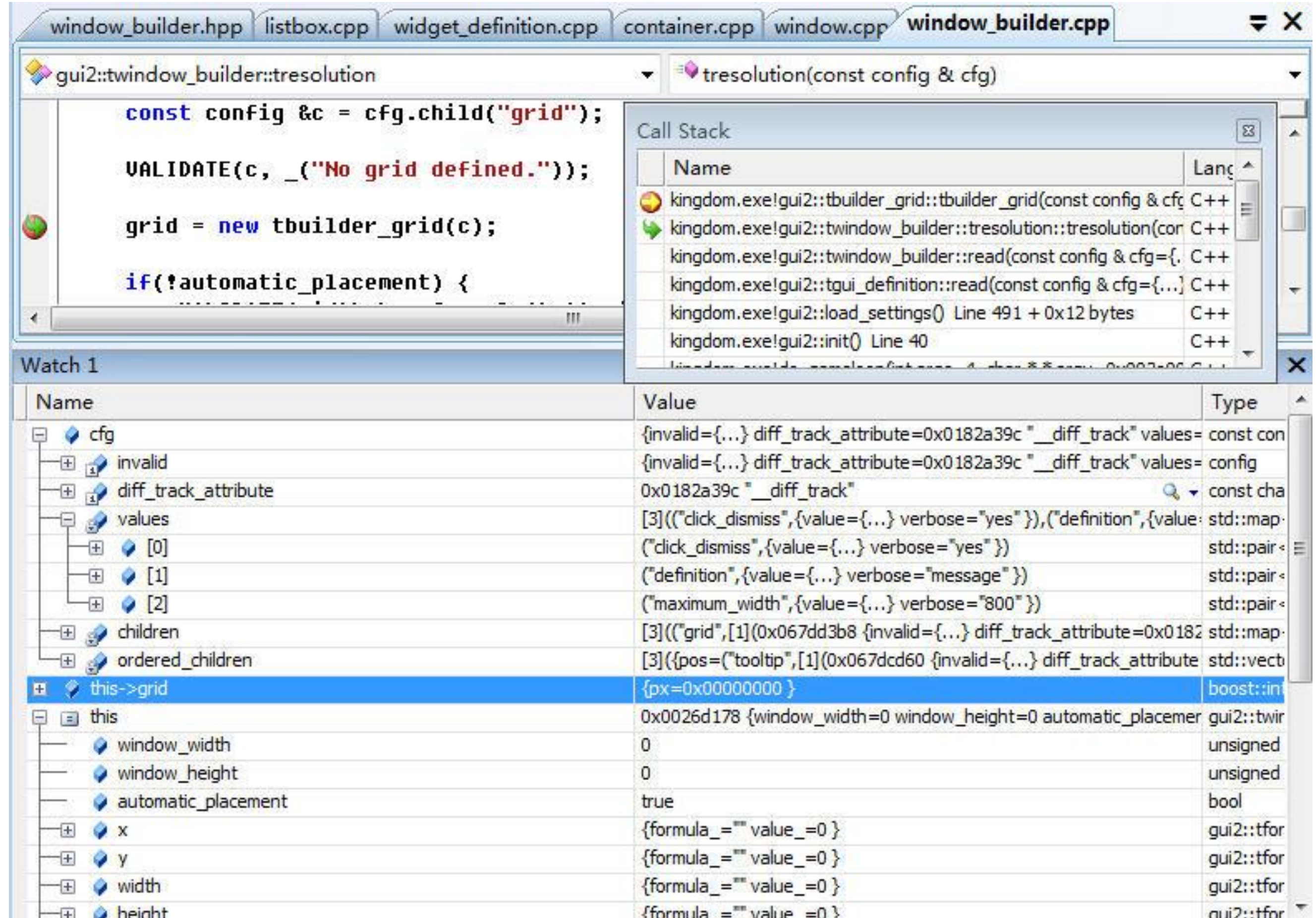


图 6-10 tbuilder_grid::tbuilder_grid

- cfg：窗口模板块[window]下的[resolution]块。

tresolution::tresoultion 解析[resolution]下的[grid]块，调用 tbuilder_grid 执行转换，生成的结果放在成员变量 grid 中。图 6-11 则是由代码 6-1 生成的 tbuilder_grid 对象各成员变量。

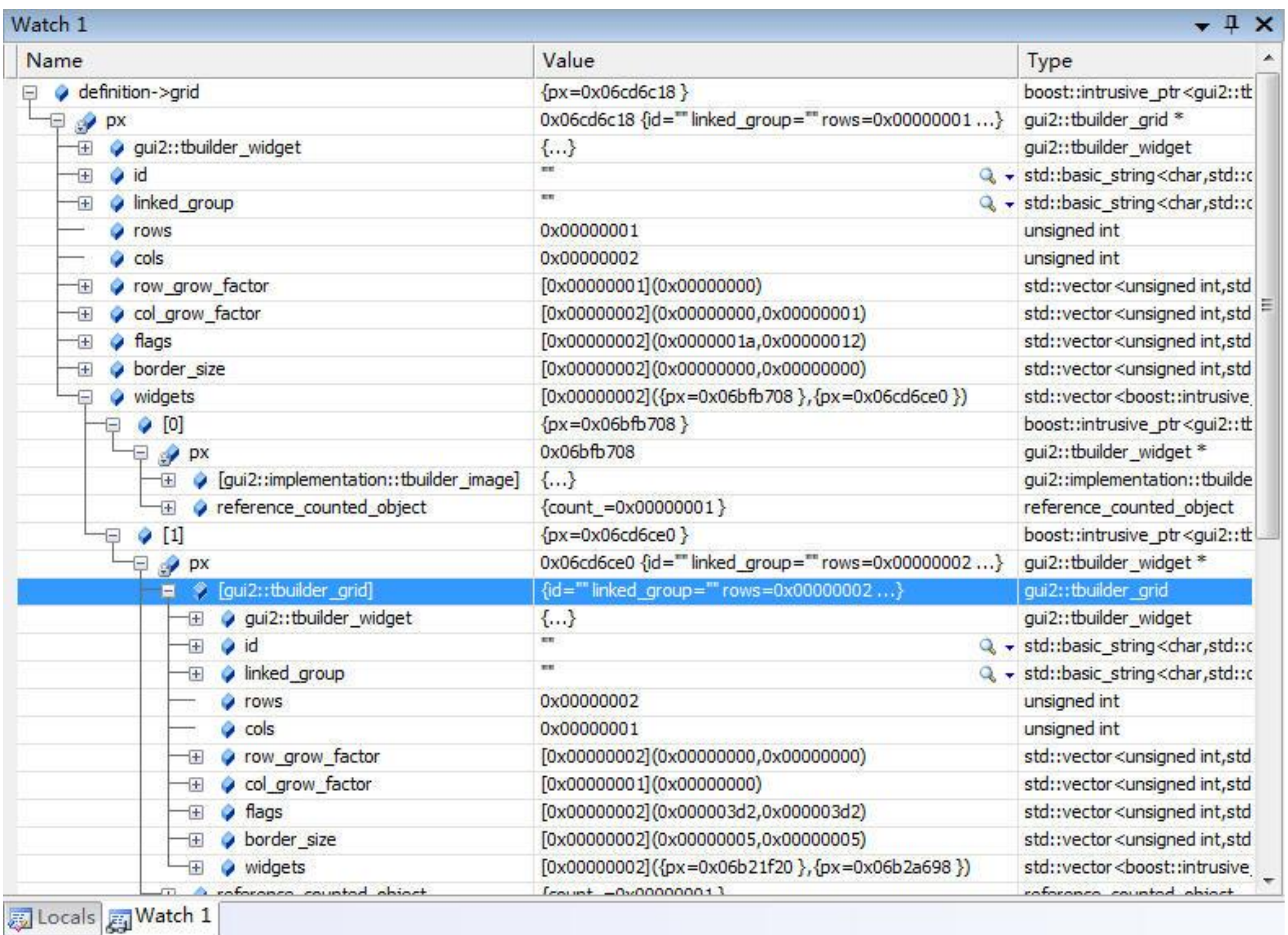


图 6-11 vairable-definition_grid

第一个网格被拆成 1x2（rows=1，cols=2），其中(0, 0)是 tbuilder_image；(0, 1)是 tbuilder_grid。由于(0, 1)格子中又是[grid]，它又要被解析，这个网格被拆成 2x1（rows=2, cols=1），当中内容存在 widgets 变量，限于区域不再显示。

以上是 C/C++如何处理窗口模板中网格，网格还可能存在容器控件，像滚动面板（scroll_panel_definition），scroll_panel_definition 中的[grid]也是被逐个生成 tbuilder_grid 对象。

网格不是控件，窗口模板、容器控件使用它来组织内中控件。加载过程把遇到的[grid]块生成 tbuilder_grid 对象。

tbuilder_widget

```
struct tbuilder_grid
    : public tbuilder_widget
{
public:
    ......
    std::vector<tbuilder_widget_ptr> widgets
    ......
};
```

widgets 用于存储该网格下所有格子，为查看 tbuilder_grid 中 widgets 生成过程，让进入 create_builder_widget 源码调试。

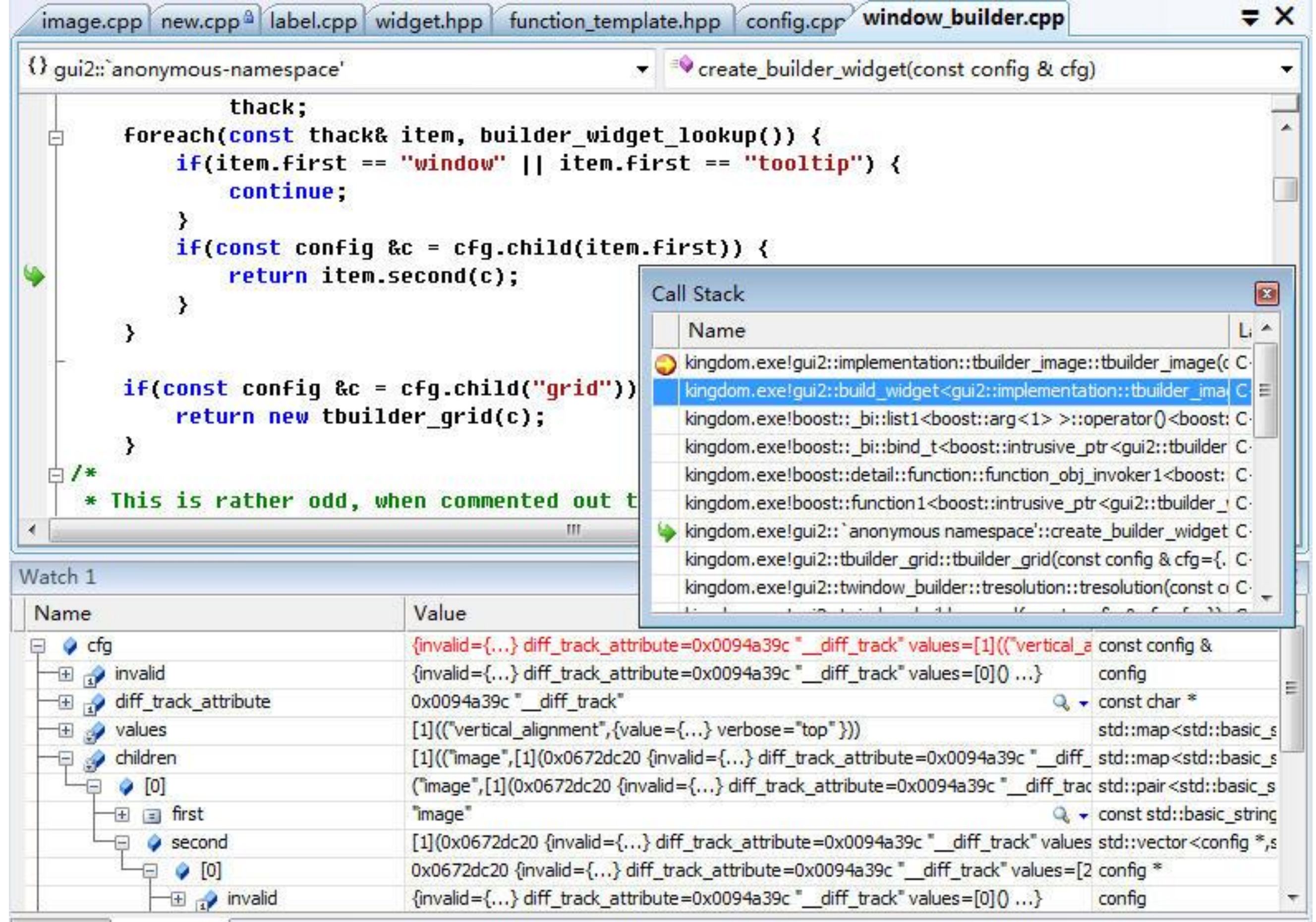


图 6-12 create_builder_widget

- cfg：[column]下子块，表示一控件块或网格[grid]。

图中 cfg 是[image]块。foreach 中的循环源来自 builder_widget_lookup，它就是注册控件形成的构建函数集！注册时说到所有控件模板使用的是同一个构建函数（build_widget，那对每个控件它们除了标识不同还有什么是不同的？——REGISTER_WIDGET3 中，由参数 id 形成的 implementation::tbuilder_##id，这个传给 build_widget 的模板参数，也就是这个 T！按钮就是 implementation::tbuilder_button，图像就是 implementation::tbuilder_image。

```
template<class T>
tbuilder_widget_ptr build_widget(const config& cfg)
{
    return new T(cfg);
}
```

通用构建函数 build_widget 执行的就一条语句，构建模板参数 T 指定的对象，像图像控件就是 new implementation::tbuilder_image(cfg)。图 6-12 代码还要注意当 cfg 是网格[grid]时，网格不是控件，foreach 项中没有一个是会和它匹配，构建网格会“跳过”foreach 调用之下“new tbuilder_grid(c)”，因而当[grid]下又有[grid]时 create_builder 将出现递归。

图 6-10 处理的[grid]块是位在窗口模板，窗口模板通过调用 create_builer 把[grid]块内所有模板定义块生成模板构建对象，同时把底下[grid]也生成 tbuilder_grid 这个构建对象。容器控件中也可能出现[grid]，那里的[grid]也被和以上一样过程被处理，即当中所有控件定义块将被生成控件构建对象，同时底下[grid]也生成 tbuilder_grid 这个构建对象，代码表现就是进入图 6-12 中的“new tbuilder_grid(c)”语句。

加载窗口模板小结

- 加载过程是从 WML 中读出窗口模板块[window]，把它转换为相应的 gui2::twindow_builder

对象，生成的结果放进窗口管理对象 tgui_defintion 中的成员变量 window_types。

- tgui_defintion::window_types 是个 std::map，map 的 first 是模板标识，像 transient_message，second 则是由该模板块生成的 gui2::twindow_builder 对象。
- WML 中[window]块必须包含 registered_window_types 中的 id 集。
- 加载窗口模板（只要是除去窗体控件的容器控件模板）时会执行构建内中控件。

控件定义对象、控件构建对象之间关系

每种控件存在三种封装，控件定义对象、控件构建对象和控件实现对象。在三者关系上，定义对象提供了配置信息，构建对象提供了使用信息，结合这两者生成了用于编程具体窗口的实现对象。定义和构建对象是窗口了系统加载过程被构造，要说它们关系，除去容器控件的定义对象会包含一些原子控件的构建对象，它们其实就没啥了。

	定义对象	构建对象
场合	配置控件。控件最大尺寸、有多少画布，画布中图元等	使用控件。如何对齐、如何缩放等。
WML 块	[xxxxx_definitionn]块（xxxxx 是控件标识）	[xxxxx]块（xxxxx 是控件标识）
创建时机	加载控件（main 外静态）	加载控件（该控件是容器控件，main 外静态）、加载窗口（main 内）
归属变量	tgui_defintion 中的 control_definition	定义对象（该控件是容器控件）、tgui_defintion 中的 window_types
实例数	唯一。一风格只创建一个	多个。视被使用次数而定

常驻内存

常驻内存指程序启动后，窗口子系统为实现功能而立即占用、并且直到程序退出才释放掉的内存。和常驻内存相关的概念是临时内存，临时内存是窗口子系统为实现某个任务而再申请的内存，像要显示某个窗口，而一旦任务结束后，像用户关闭该窗口，这些内存就会被释放。常驻内存和临时内存区别在于在内存的驻留时间。

讨论常驻内存原因在于提高窗口子系统效率。常驻内存可说是整个程序生命期都占用内存，这部分内存越大，其它子系统可用到内存就越小，为让其它子系统设计可以更灵活，常驻内存是越小越好。对于临时内存，它因任务而生，任务开始而申请，任务一结束就释放，它可说不影响设计其它子系统灵活性（肯定有影响，但它只要运行任务那一刻总内存足够就行）。以设备可用内存来说，相比于 PC，移动设备较为紧张，程序设计时内存要能省则省，具体到此处设计窗口子系统则表现为要尽量减少常驻内存。

注册控件/窗口模板发生在 main 函数之前静态变量赋初值时机，加载控件/窗口模板发生在程序一进入之后，而且这两个过程一旦占用哪些内存了，这些内存要等到程序退出才会被释放。注册、加载形成了窗口子系统的常驻内存。根据以上对两过程分析，可得出常驻内存分为两部分：控件配置、窗口配置。

- 控件配置：加载控件模板时形成。来自 WML 中的控件定义块，C/C++中存储格式是 tcontrol_definition 派生对象，具体存储在窗口管理对象 tgui_defintion 中的成员变量 control_definition。
- 窗口配置：加载窗口模板时形成。来自 WML 中的窗口定义块，C/C++中存储格式是 twindow_builder 和 tbuilder_control 的派生对象，具体存储在窗口管理对象 tgui_defintion 中的成员变量 window_types。

5.3.6 构造窗口

模板+具体数值=实例

对窗口，它也符合以上这个逻辑，窗口模板+具体数值=具体窗口。构造指的是根据窗口标识从已注册窗口模板集中找出模板，并把当前数据结合到模板，然后生成具体窗口的过程。

构造窗口需构造出窗口中的所有控件。要把加载过程形成的控件构建对象（tbuilder_<id>+）生成相应的控件实现对象（t<id>+）。至于如何计算控件在窗口中位置则等到“6.3.6 布局”。

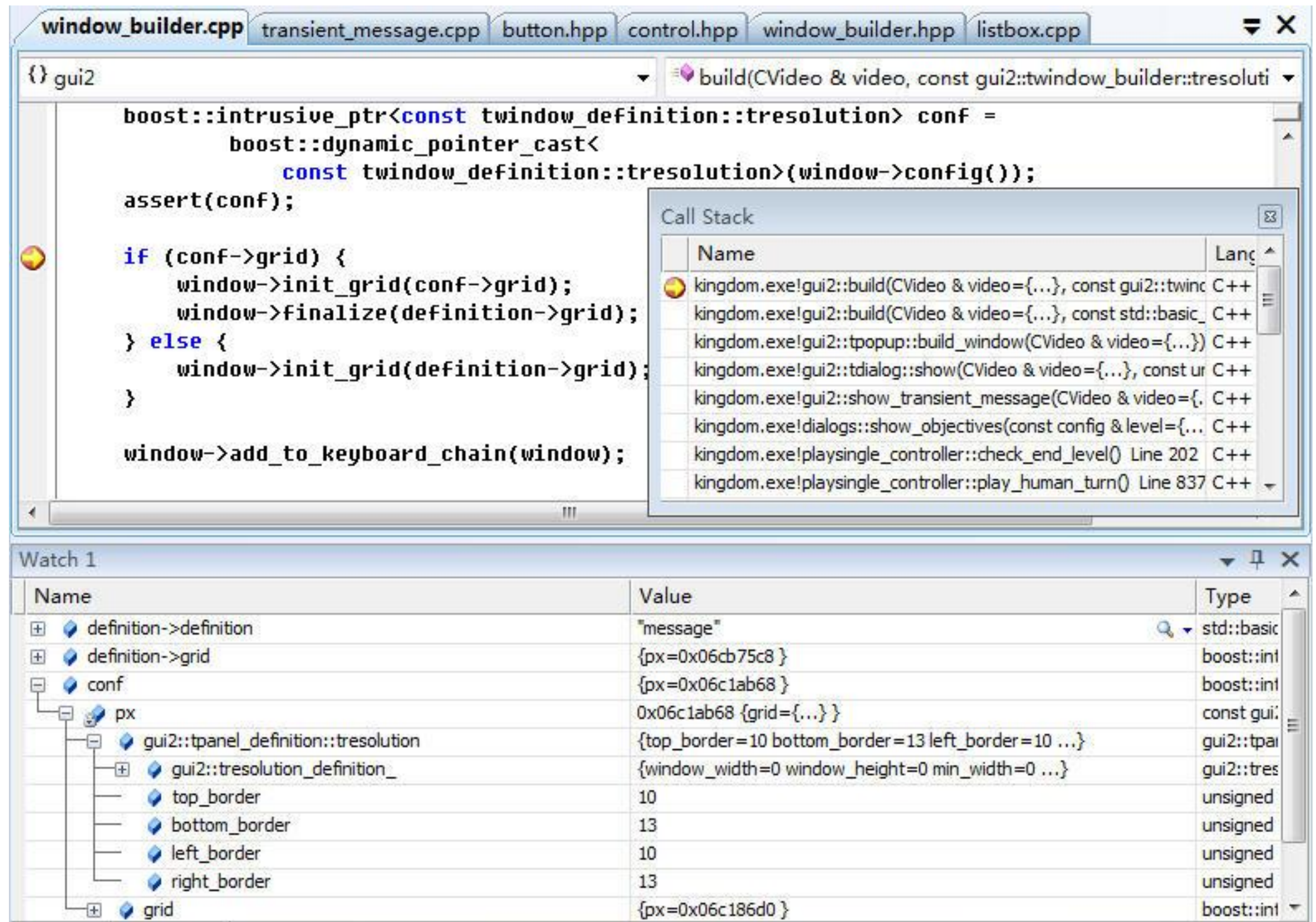


图 6-13 gui2::build

gui2::build 根据参数 definition 指示的窗口配置构造一个窗口。要理解这函数首先需要知道 definition 从哪来。让在 Call Stack 中跳到上一级的 build 函数。

```
twindow *build(CVideo &video, const std::string &type)
{
    std::vector<twindow_builder::tresolution>::const_iterator
    definition = get_window_builder(type);
    twindow *window = build(video, &*definition);
    window->set_id(type);
    return window;
}
```

它是个同名不同操作的 gui2::build，参数 type 指示要使用的窗口标识，它来自于窗口实现类，针对“关卡任务目标”窗口它的实现类是 ttransient_message，对应窗口标识是 transient_message。build 以这个模板标识为参数调用 get_window_builder 得到当前分辨率下窗口配置 definition。但以内容上说，至少到此处 definition 存储是该模板共通值，没有加载“任务目标窗口”的私有数据。

到此已根据窗口标识从已注册窗口模板集中找出配置 definition，这 definition 作为参数传给 gui2::build。gui2::build 根据 definition 构造 twindow 对象 window，注意下 definition 没有加载具体窗口数据，此个 window 存储的也是该模板共通值。

接下 gui2::build 要把加载过程形成的控件构造对象（tbuilder_<id>+）生成相应的控件实现

对象（t<id>+），这些控件构造对象中首个要处理的是位在顶层的窗体模板构造对象。

```
boost::intrusive_ptr<const twindow_definition::tresolution> conf =
        boost::dynamic_pointer_cast<
            const twindow_definition::tresolution>(window->config());
```

此条语句得到的 conf 正是窗体控件的分辨率配置（调用 tcontrol::config()前必须先调用过 tcontrol::set_config(...)，之前 twindow::twindow 会调用 set_config）。init_grid 执行对象转换，为看转换是怎么个过程，进入 init_grid 进行源码级跟踪。

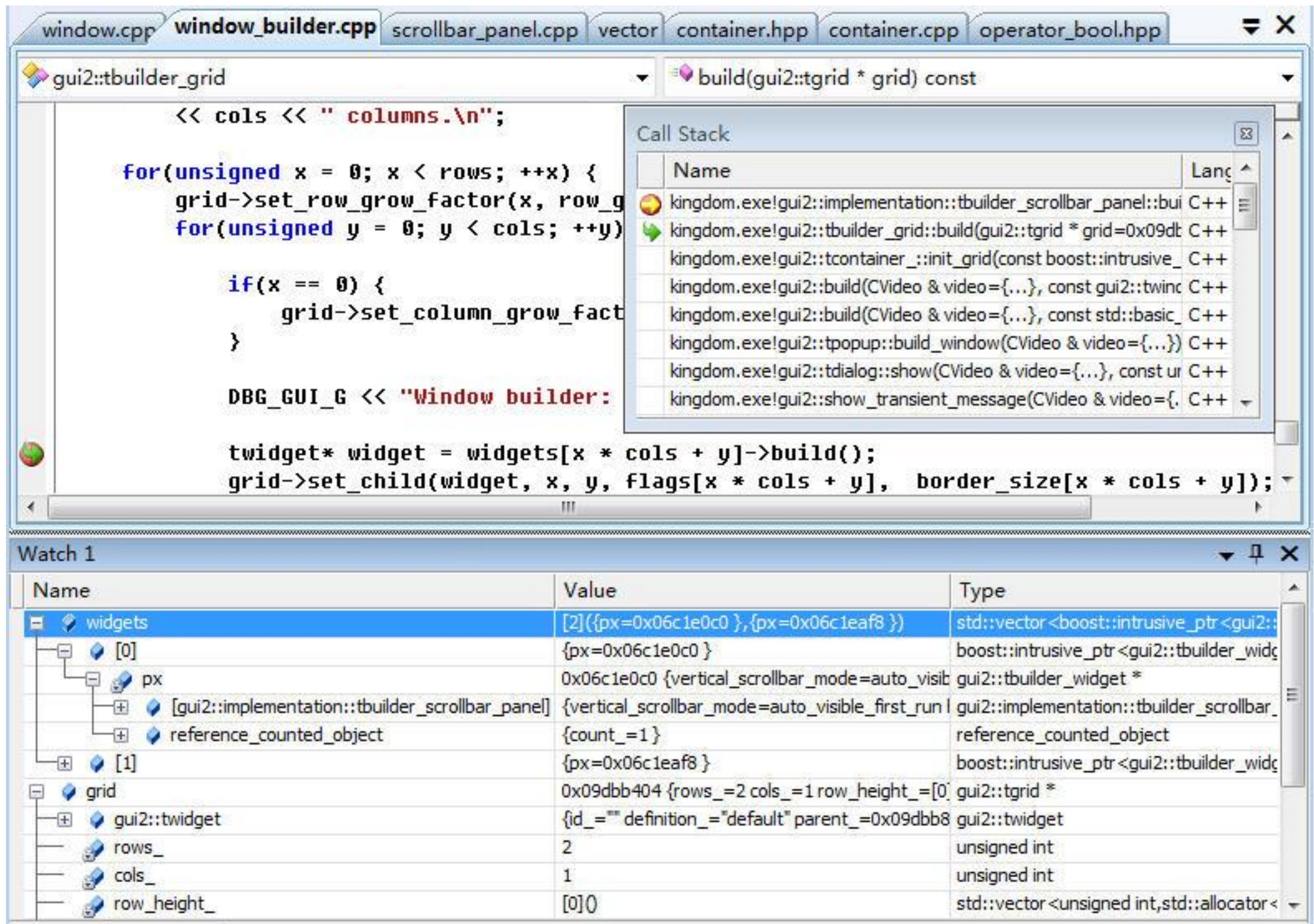


图 6-14 tcontainer_::init_grid

正要被转换的控件构建对象是 tbuilder_scroll_panel，转换到的控件实现对象是 tscroll_panel。init_grid 执行的是转换网格，即 tbuilder_grid 到 tgrid。网格转换过程是逐个转换它包含的格子中对象，如果格子中是网格，函数会发生递归。当转换完格子中所有对象后，它们的容器网格转换也就结束了。

让看下“if (conf->grid)”，在说到网格时曾写过“窗口模板顶层是一网格。当窗体控件是容器控件时，这个网格会替换掉窗体控件中的“_window_content_grid”网格，从而实现两者合一。否则窗口模板顶层网格就作为这个窗口的顶层网格”。conf->grid 正是指示了窗体控件中的[grid]块，当窗体控件是容器控件时，conf->grid 非 NULL，这时窗口模板顶层网格就会替换掉窗体控件中“_window_content_grid”网格（finalize 就用于实现转换窗口模板顶层网格及替换）。当窗体控件是原子控件时，只要转换窗口模板顶层网格就行了。

到此已大致构造出窗口模板实现类通用数据，但到现在也没看到私有数据被应用于这个 twindow。这就是接下 pre_show 函数要做的事。

pre_show 是 tdialog 的虚函数，它用于提供机会给窗口实现类实现私有操作，像设置标题，向菜单填充数据。对调用时机， pre_show 是在窗口要被显示时被窗口管理者调用。

构造小结

- 各个控件构造对象须提供 build()方法（包括网格），从而把自已转成控件实现对象。
- 构造要把加载过程形成的控件构建对象（tbuilder_<id>+）生成相应的控件实现对象（t<id>+）。窗口最顶端是网格构造对象，因而只要把该网格转换了就可以逐个转出所有控件实现对象。
- gui2::build 返回的 twindow 是窗口模板实现类的通用数据，不含私有数据。
- pre_show 用于向 twindow 填充私有数据。

5.4 布局

构造出窗口以及当中所有控件、把私有数据填充入窗口，接下工作是渲染该窗口。渲染过程肯定要确定各控件在窗口中位置，渲染时发现没有布局过窗口、或需要重新布局窗口，它就会调用窗口的布局逻辑。

在此有个疑问，为什么不在构造时执行布局？布局不仅和当前屏幕分辩率相关、还和窗口当前内容相关。举个窗口中滚动面板控件例子，滚动面板用于显示一段文字，这段文字伴随用户切换菜单项而变动，一旦滚动面板出现显示/隐藏滚动条切换，这时就要重布局窗口。布局放在渲染中有利于统一处理这些变动，对窗口系统来说，构造执行的是“一次性”的动作。

5.4.1 何时布局

gui2::twindow 内有个 need_layout_成员变量，当渲染逻辑（twindow::draw()）检测到该变量是 true 执行布局。总的来说布局出现在两种情况。

1）第一次渲染窗口时。

2）调用过 twindow::invalidate_layout()。后者会把 need_layout_置为 true。

5.4.2 步骤一：计算窗口的容许尺寸

根据最大尺寸和 WML 中写的窗口[resolution]字段计算出容许尺寸。

```
const int maximum_width = automatic_placement_
       ?  maximum_width_
           ? std::min(maximum_width_, settings::screen_width)
           : settings::screen_width
       : w_(variables_);

const int maximum_height = automatic_placement_
       ? maximum_height_
           ? std::min(maximum_height_, settings::screen_height)
           : settings::screen_height
       : h_(variables_);
```

计算出的 maximum_width、maximum_height 就是容许尺寸。settings::screen_width、settings::screen_height 是最大尺寸。automatic_palcement、maximum_width/maximum_height、w_/h_都是窗口配置中字段。为更清晰容许尺寸计算过程让看两个例子，假设最大尺寸(settings::screen_width, settings::screen_height)=(800, 600)。

transient_message（关卡任务目标）窗口配置中和容许尺寸相关的字段。

```
maximum_width = 800
根据字段得出
automatic_placement_=true（默认值是 true）。
maximum_width_=800
maximum_height=0
w_=""
h_=""
计算尺寸
maximum_width=std::min(maximum_width_, settings::screen_width)=800
maximum_height=settings::screen_height=600
```

title screen（标题屏幕）窗口配置中和容许尺寸相关的字段。

```
automatic_placement = "false"
width = "(screen_width)"
height = "(screen_height)"
```

```
根据字段得出
automatic_placement_=false
maximum_width_=0
maximum_height_=0
w_="(screen_width)"
h_="(screen_height)"
计算尺寸
maximum_width=w_=800
maximum_height=h_=600
```

容许尺寸不一定是渲染尺寸。

5.4.3 步骤二：计算各控件和窗口的标称尺寸

要能显示窗口，它的标称尺寸必须小于或等于容许尺寸，否则布局失败。窗口标称尺寸是累加内部各控件标称尺寸计算出的尺寸，为此这问题就变成如何计算控件的标称尺寸。控件标称尺寸是控件宣布此刻我应该占据的最小尺寸，twidget::get_best_size()执行这个计算，get_best_size则借助 calculate_best_size。

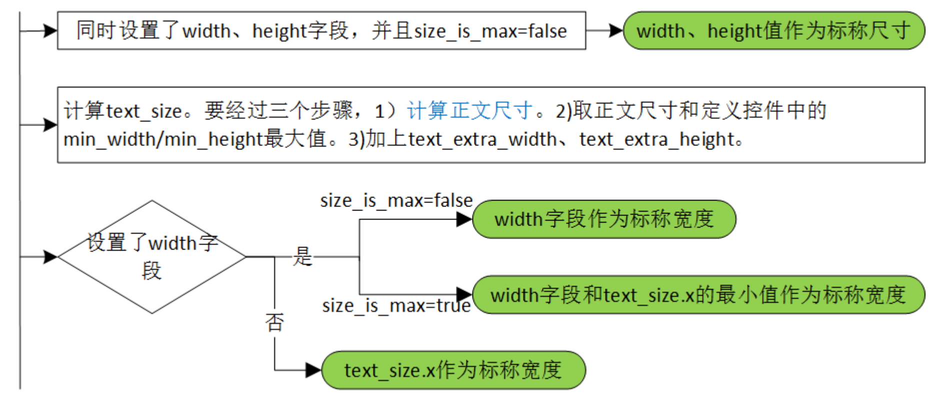


图 5-15 tcontrol::calculate_best_size

tcontrol::calculate_best_size 是计算控件标称尺寸的入口，图 5-15 显示了它的执行逻辑。图中没画计算 height 的步骤 4，它类似步骤 3，系统是相互独立计算 width、height。对不同控件，不同的只是步骤 2 中的“计算正文尺寸”。

窗口脚本中和计算标称尺寸相关的有三个字段，width、height 和 size_is_max。width、height留空表示未设置，设为 0 那也是设置了的。所有控件都能设置 width、height，但只有可滚动控件能把 size_is_max 设为 true（该字段默认是 false）。

总的来说，计算出的标称尺寸可能会是两种值，以宽度为例，按优先级依次是“设置的 width 字段”，“计算出的 text_size.x”。对 text_size.x，可概括为以下公式。

```
text_size.x = max(正文宽度, min_width) + text_extra_width;
```

要理解这个公式，让看图 5-16 的三个控件。

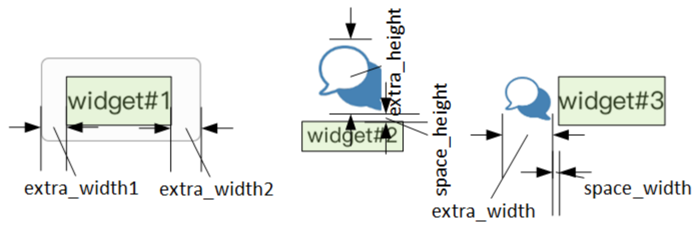


图 5-16 正文尺寸

三个控件中淡绿色表示的是正文区域，其它则是额外部分。控件 1 是个 default 风格按钮，正文的左右两侧长度是 extra_width/2，上下则是 extra_height/2，extra 部分使得正文四个方向总留有空隙。控件 2 是 vertical_button 风格按钮，extra_width 是 0，extra_height 是位在上面的图标高度，那它底下为什么会有 space_height？space_height 是个根据正文是否存在内容而取不同值的字段，当正文区域是(0, 0)时，值是 0，否则是配置中的值。控件 3 类似控件 2，只不过换为

extra_height 是 0，space_width 意义等同 space_height。

对控件 1，字符串可能是一个字符，也可能很长，为美观会希望正文至少保留一定宽度，使得一个字符时不致于很窄，对此可设置 min_width。同样，在垂直方向可设置 min_height。

不管有没有正文，控件 text_size.x 至少会有 extra_width，text_size.y 则至少有 extra_height。min_width、min_height 比较的是正文，不包括 extra 部分。space_width、space_height 则是和正文有没有内容相关的字段。它们都在控件的定义脚本被赋值。

字段	PC 是否缩小	描述
min_width	是	最小正文宽度
min_height	是	最小正文高度
extra_width	是	额外宽度。text_size.x 至少会有这宽度
extra_height	是	额外高度。text_size.y 至少会有这高度
text_space_width	否	有正文时再增加的宽度
text_space_height	否	有正文时再增加的高度
label_is_text		label_字段是否是字符串，而且它就是正文

表中除了 6 个字段，还出现了 label_is_text，这也是个和计算正文尺寸相关的字段。在顶层，calculate_best_size 根据 label_is_text 这个布尔变量把正文分为两类。label_is_text 是 true 时，认为 label_字段中存的是字符串，正文尺寸就是字符串占用的尺寸，图 5-16 三个控件都属这个类型。label_is_text 是 false 时，将调用 mini_get_best_text_size，它是个虚函数，通过重载它，各种控件计算出自个的私有正文尺寸。举个例子，图像控件中 label_字段存储的是图像文件路径，它的 mini_get_best_text_size 执行读取图像，读出的图像尺寸就作为正文尺寸。

由于只能在一个方向进行扩展，space_width、space_height 不可能同时是非零。另外，这两个值只在 label_is_text 是 true 时生效。

表中“PC 是否缩小”表示换到 PC 时是否要把值缩小。相比移动设备，PC 有鼠标，命中率不是问题，而缩小有助于美观。当前缩小就是取配置值的 3/4，为避免出现小数，需要缩小字段的值必须是 4 的整数倍。

下表汇总各控件正文尺寸，具体控件的更多细节参考“**第七章 控件**”，那里写的标称尺寸准确说是正文尺寸。

控件	正文	编写窗口脚本时建议
button	label_是字符串，正文尺寸是字符串尺寸	不要设置 width、height 字段
label	label_是字符串，正文尺寸是字符串尺寸	不要设置 width、height 字段
image	label_是图像文件路径，正文尺寸是图像尺寸	
text_box	总是(0, 0)	控件定义有 min_width、min_height
toggle_button	label_是字符串，正文尺寸是字符串尺寸	
track	总是(0, 0)	或设置 width、height，或通过其它控件保留出一个区域
panel	内中各控件的标称尺寸和	
toggle_panel	内中各控件的标称尺寸和	
stack	不同 mode 用不计算方法，见“7.5 栈层”	
report	见“7.7 报表”	
listbox	见“7.8 列表”	
tree	总是(0, 0)	
scroll_panel	内中各控件的标称尺寸和	
scroll_text_box	总是(0, 0)	

5.4.4 步骤三：计算窗口的渲染尺寸及左上角坐标

```
tpoint size = get_best_size();
tpoint origin(0, 0);
if (automatic_placement_) {
    switch (horizontal_placement_) {
        case tgrid::HORIZONTAL_ALIGN_LEFT :
            // Do nothing
            break;
        case tgrid::HORIZONTAL_ALIGN_CENTER :
            origin.x = (settings::screen_width - size.x) / 2;
            break;
        case tgrid::HORIZONTAL_ALIGN_RIGHT :
            origin.x = settings::screen_width - size.x;
            break;
        default :
            assert(false);
    }
    switch (vertical_placement_) {
        ......
    }
} else {
    size.x = w_(variables_);
    size.y = h_(variables_);

    variables_.add("width", variant(size.x));
    variables_.add("height", variant(size.y));
    origin.x = x_(variables_);
    origin.y = y_(variables_);
}
```

首先调用 get_best_size()得到的窗口的标称尺寸（size.x/size.y）。automatic_placement=true 表示自动放置，此时渲染尺寸将等于标称尺寸。计算左上角坐标则涉及到[resolution]中如何对齐字段，即 horizontal_placement（水平）和 vertical_placement（垂直）。

非自动方置时，用公式依次计算渲染尺寸、左上角坐标。算左上角坐标时可使用已算出的渲染尺寸，自变量“width”、“height”表示了这个尺寸。在编写 cfg 时，建议这两自变量仅限用于计算 x_、y_。正是这原因，自动放置时不设置这两个变量。

5.4.5 步骤四：各控件的标称尺寸拉伸到渲染尺寸，放置所有控件

控件的标称尺寸小于渲染尺寸时，会被放大。具体函数上，tgrid::place 执行了放置的主要工作，而 tgrid 中的成员变量 row_height_、col_width_在不同阶段分别存储着标称、渲染尺寸值。

	std::vector<unsigned> row_height_	std::vector<unsigned> col_width_
语义	网格中各行高度，包括了 border。可能标称、可能渲染。	网格中各列宽度，包括了 border。可能标称、可能渲染
calculate_best_size 后	网格中各行标称高度	网格中各列标称宽度
place 后	网格中各行渲染高度	网格中各列渲染宽度

tgrid::place 有两个任务，一是根据 calculate_best_size 算出的标称尺寸，结合拉伸系数算出渲染尺寸。二是调用 tgrid::layout，放置该网格中的所有控件。

第一步，根据标称尺寸算出渲染尺寸。以下是执行计算宽度的代码，高度类推。

```
const unsigned w = size.x - best_size.x;
unsigned        w_size        =        std::accumulate(col_grow_factor_.begin(),
col_grow_factor_.end(), 0);
if (w_size == 0) {
    // If all sizes are 0 reset them to 1
    BOOST_FOREACH(unsigned& val, col_grow_factor_) {
        val = 1;
    }
    w_size = cols_;
}
const unsigned w_normal = w / w_size;
for (unsigned i = 0; i < cols_; ++i) {
```

```
    col_width_[i] += w_normal * col_grow_factor_[i];
}
```

结合“**5.2.2 控件尺寸的弹性**”中的示例历遍这代码。size.x 是容许宽度，值 660，best_size.x 是标称宽度，值 600，w 指示需要放大的 60px。w_size 是该行中 3 列的放大系数和，三个单元的值是 1、2、3，算出 w_size 值是 6。根据 w、w_size，算出 w_normal 是 10。接下是计算各列渲染宽度，“**col_width_[i] += w_normal * col_grow_factor_[i]**”是具体拉伸公式，于是 3 列分别要放大 10px、20px 和 30px，相应地渲染尺寸分别变为 210px、220px 和 230px。

顺便说下，当 col_grow_factor_中所有单元都是 0 时，会被全改到 1。

第二步，调用 tgrid::layout，放置该网格中的所有控件。算出各栅格的渲染尺寸后，就要调用 tgrid::tchild::place(tpoint origin, tpoint size)在栅格内放置控件/网格。“同一行内格子必相同高度，同一列内格子必相同宽度”，这个高度(row_height_[i])是同一行内最高格子的高度，这时问题就来了，对于其它高度小于 row_height_[i]的格子，在垂直上该怎么放，是拉伸填满？是上对齐？是居中？还是下对齐？水平上也存在同样问题。

通过在栅格对应的 [column]块设置 vertical_alignment 属性来决定那些块的垂直对齐方式，horizontal_alignment 则对应水平方向。下表是各种垂直对齐值。

vertical_alignment	描述
edge	两端对齐。执行拉伸，控件的渲染尺寸将等于最大尺寸
top	上对齐。不执行拉伸，控件的渲染尺将等于标称尺寸
center（默认值）	居中对齐。不执行拉伸，控件的渲染尺将等于标称尺寸
bottom	下对齐。不执行拉伸，控件的渲染尺将等于标称尺寸

twindow::layout()执行布局全部操作，用流程图表示四个步骤。

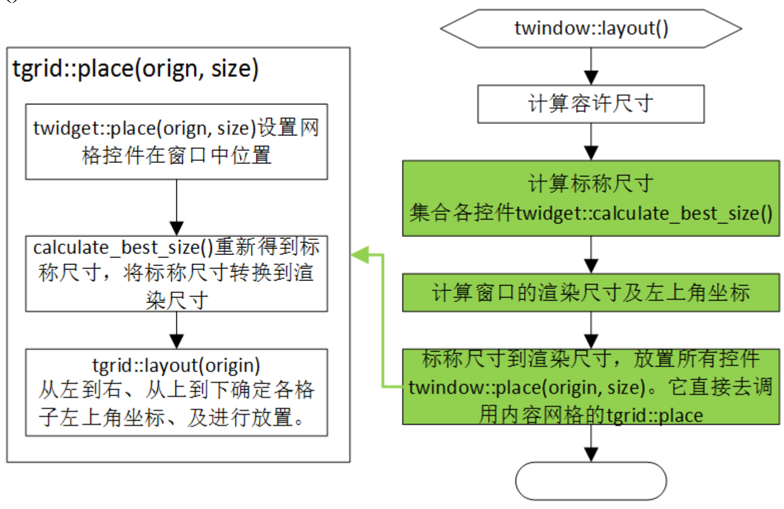


图 5-17 twindow::layout

格子块[column]中和布局相关字段

- grow_factor：指定该格子宽度拉伸系数。“同一列内格子必相同宽度”，为保证这条规则要求同一列内格子必须相同宽度拉伸系数，因为拉伸相同的值才能保证新宽度是一致的，而要拉伸相同值则要求相同拉伸系数，如何计算拉伸值见“步骤四：标称尺寸拉伸到渲染尺寸，实现放置所有控件”。当网格下有多行时，只取第一行中那些个[column]的 grow_factor，只有第一行写的 grow_factor 才是有效的宽度拉伸系数。顺便说下 grow_factor 可以出现在[row]根下，那时它用于指定该行中所有格子的高度拉伸系数。可取值：自然数。
- vertical_alignment：当该行最大高度大于该格子高度时，指示如何在垂直方向上放置控件。可取值：edge、top、center、bottom。
- horizontal_alignment：当该列最大宽度大于该格子宽度时，指示如何在水平方向上放置控件。可取值：edge、left、center、right。

布局小结

- 同一行内格子必相同高度；同一列内格子必相同宽度。
- 标称尺寸不能大于容许尺寸，否则布局失败。
- 标称尺寸小于渲染尺寸时要执行拉伸。
- 一次布局过程中某个控件可能拉伸，不可能缩小

5.5 渲染

控件拼凑出窗口，实现渲染“所有”控件就等于实现了渲染窗口，渲染窗口问题就具体到如何渲染控件。广义上说布局也属于渲染，它解决了控件画在哪里，知道画在哪后就要解决怎么画、画什么。

5.5.1 恢复式和累加式

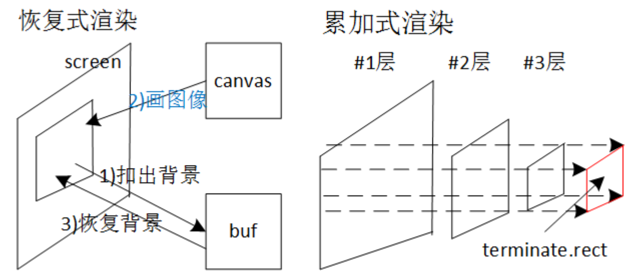


图 5-27 恢复式渲染和累加式渲染

恢复式渲染。时刻 t1 要画图像 canvas1 了，它从 screen 上先扣出背景到纹理 buf，然后把 canvas1 渲染到 screen。时刻 t2 要画图像 canvas2 了，先把背景 buf 渲染回 screen，即恢复背景，然后把 canvas2 渲染到 screen。

步骤	常用函数名	描述
扣出背景	redraw#1	screen-->buf 记住要画位置处旧有矩形图像，这矩形图像称为背景图像，buf 存储着背景纹理。
画图像	redraw#2	canvas-->screen。在位置处画图像
恢复背景	undraw	buf-->screen。当图像要被擦除时，位置处恢复背景图像

redraw 函数先后执行两个操作，一是扣出背景，二是画出此次内容。恢复背景时，作为贴图的 buf 要以着“只取贴图”的 alpha 混合方式画到 screen。

要同时渲染多个 canvas、而且 canvas 间有可能出现重叠时，在 redraw 阶段，各单元以着顺序去操作，undraw 阶段则以着逆序去操作。同样原因，多种要被依次渲染时，redraw 时顺序，undraw 是逆序。具体到渲染窗口时，同时要渲染悬浮控件、浮动动画、光标，flip 是个分界。

```
window->draw_float_widgets();
disp->draw_float_anim();
cursor::draw();

video.flip();

cursor::undraw();
disp->undraw_float_anim();
window->undraw_float_widgets();
```

累加式渲染。从#1、#2、#3……#N，逐层渲染。为降低渲染消耗，会预先设置裁剪矩形，矩形尺寸是末尾层要渲染的尺寸，terminate.rect。

怎么画

在<src>/gui/auxiliary/cavas.cpp 中的 ttext::draw 内设断点，运行程序让触发这个断点（只要显示窗口就行，哪个窗口无所谓），看下当时函数栈。

ttext::draw 用于画一串字符。怎么画的流程可分为两步骤，twindow::draw 是分界点，把这步

骤分别称为 twindow::draw 外、twindow::draw 内。draw 是 twindow 私有操作，gui2::twidget 中只有 twindow 提供了这操作。

twindow::draw 外

twindow::draw 主要是完成接收 SDL 事件，从而让触发 twindow::draw。

图 6-18 会找到 events::pump，“如何处理 SDL 事件”时说到这函数用于从 SDL 库提取事件，然后广播给 SDL 事件处理者。gui2 系统实现如何画窗口利用了这个机制，在构造时把自个注册为 SDL 事件处理者，当要画窗口时 SDL 会发出 DRAW_EVENT 事件，于是 gui2 就能收到 DRAW_EVENT，针对 DRAW_EVENT 这个事件码去调用 twindow::draw 操作。大概逻辑就这样，具体到细节还须要说得更清楚。

问题一：把谁注册为 SDL 事件处理者

整个 gui2 系统就一个 SDL 事件处理者。这里为什么不用控件对应事件处理者？gui2 可以把所有控件都注册为 SDL 事件理者，也就是说一个窗口有 N 个控件就意味着系统至少将存在 N 个事件处理者。可以想象下窗口中存在列表（列表是容器控件，每一项对应一个原子控件），当屏幕要显示 15 行，每行有 5 项，窗口光列表就至少产生 75 个事件处理者，由于 events::pump 是广播事件，每处理一个事件都要群发，现实中无法忍受这样执行效率。一个控件对应一个处理者效率太低，那为什么不用一个窗口对应一个处理者？窗口对应处理者在效率上是可接受了，可对于多个窗口，虽然具体界面上表现出不一样，它们内里很多逻辑是共通的，在处理 SDL 事件上也会表现出很多共通逻辑，统一在一个函数好过分散。

gui2 如何定义处理者，如何关联处理者和窗口？它定义一个从 event::handler 派生的类：gui2::event::thandler，它重载 handle_event 操作。gui2 另外又定义类：gui2::event::tdispatcher，它被使用为 gui2::twidget 基类，gui2::twidget 又是 gui2::twindow 基类，即 gui2::event::tdispatcher 是 gui2::twindow 基类，对于 thandler、tdispatcher 这两个新定义类，它们之间通过 thandler 成员变量 std::vector<tdispatcher*> dispatchers_进行关联，这变量用于存储当前正连接着分发者（tdispatcher）对象，分发者就是窗口。

接下让叙述用户操作窗口时的 thandler、tdispatcher。当用户创建窗口 A，窗口 A 被创建那一刻它被加入 thandler::dispatchers_。当用户在窗口 A 按鼠标、按键盘，代码进入 thandler::handle_event，该函数执行初步事件过滤，然后发去 dispatchers_中窗口。一旦用户消毁窗口 A，该窗口 A 对应的 tdispatcher 就被从 thandler::dispatchers_删除。如果用户没销毁窗口 A 情况下弹出窗口 B，这时 thandler::dispatchers_长度将是 2，thandler::handle_event 就会把事件依次发去两个窗口，一旦 B、A 都被销毁，thandler::dispatchers_将归为空。

gui2::event::tdispatcher::connect()：用于把窗口 A 加入 thandler::dispatchers_。它在 twidnow 构造函数中被调用。

gui2::event::disconnect_dispatcher(tdispatcher* dispatcher)：用于从 thandler::dispatchers_销毁窗口 A。它在 twindow 析构函数中被调用，具体是在基类 tdispatcher::~tdispatcher()。

顺便说下 gui2::event::thandler 对象加入 SDL 处理者栈时机。系统中不存在 gui2 窗口时就不该有 gui2::event::thandler 对象，因而代码是用一个指针表示 gui2::event::thandler 对象。

```
<src>/gui/auxiliary/event/handler.cpp
static thandler* handler = NULL;
```

免得出现内存问题，用 gui2::event::tmanager 来管理 handler 指针，即 tmanager 构造函数中创建一个 gui2::event::thandler 对象赋给 handler，析构时删除 handler 对象。tmanager 是在程序一进去时就构建，这时不该让 gui2 中的这个 thandler 成员 SDL 事件处理者（因为它就没窗口任务），因而 gui2::event::thandler 在构造时选择是不加入处理者栈，要等到确实要创建窗口才去加入，而一旦所有窗口都销毁它要退出处理者栈。

问题二：DRAW_EVENT 事件

DRAW_EVENT 是程序自定义事件，不是 SDL 标准事件。

```
#define DRAW_EVENT (SDL_USEREVENT + 3)
```

自定义事件须要解决如何被触发。既然不是 SDL 标准事件，要是不做其它处理 SDL 系统自然不会触发 DRAW_EVENT，events::pump 也就不会收到这个事件。程序是通过向 SDL 挂接一个 30 毫秒定时器，定时器处理函数（draw_timer）执行把 DRAW_EVENT 压入 SDL 事件栈。这样一来，挂接动作的 30 毫秒后，draw_timer 被调用，draw_timer 把 DRAW_EVENT 压入 SDL 事件栈，紧接下的 events::pump 从 SDL 事件栈提取事件时取出 DRAW_EVENT，DRAW_EVENT 被发向 gui2::event::thandler 这个事件处理者，于是就实现一次“自创”DRAW_EVENT 事件产生、处理过程。为让窗口保持“最新”界面，draw_timer 不是只被调用一次就够，它除了把 DRAW_EVENT 压入事件栈还要告知 SDL 在下一个 30 毫秒后再次执行这逻辑，实现这个告知只要把返回值置为 30 就行了。

```
static Uint32 draw_timer(Uint32, void*)
{
    SDL_Event event;
    SDL_UserEvent data;

    data.type = DRAW_EVENT;
    data.code = 0;
    data.data1 = NULL;
    data.data2 = NULL;

    event.type = DRAW_EVENT;
    event.user = data;

    SDL_PushEvent(&event);
    return draw_interval;
}
```

draw_timer 是定时器处理函数，主要执行两个操作：把 DRAW_EVENT 压入事件栈；告知 SDL 多少时间后再次执行。SDL 根据返回值来决定下一次是何时再次调用这定时器函数，像 30 就指示 30 毫秒后再执行，如果是 0 则会销毁这个定时器。draw_interval 是个文件域变量，它一般情况下置为 30，即每隔 30 毫秒让 gui2::event::thandler 收到一个 DRAW_EVENT。当 gui2 要自动销毁窗口，它只要把 draw_interval 置为 0。

问题三：thandler::draw 到 twindow::draw

gui2 唯一事件处理者 gui2::event::thandler 收到事件后调用 handle_event，针对 DRAW_EVENT 事件 handle_event 调用 thandler::draw。

thandler 内有个 dispatchers_成员变量，用于存储当前正连接着窗口对象，thandler::draw 要实现画当前所有窗口只要用个 for 循环逐次调用 dispatchers_[index]->draw。的确，这样已能实现从 thandler::draw 到 twindow::draw，但图 6-18 在 thandler::draw 到 twindow::draw 之间还经历若干函数，像 tdispatcher::fire。

fire_event、implementation::fire_event，这些函数说明见之下“6.3.8 处理输入事件”。此处使用它们目的是为了获得更好的扩展性。另外注意下 thandler::draw 转换事件码，它在向底下挂接的窗口发 DRAW_EVENT 时把事件码变成 DRAW，即对底下 twindow 来说 DRAW 指示“画”事件。

twindow::draw 内

到此应该是向界面显示该窗口中所有控件的时候。为连贯 SDL 渲染时采用双图面，即开辟两图面，一图面用于显示在屏幕（即前台的帧缓存 framebuffer），一图面用作后台缓存，后台图面准备就绪后用 flip 命令切换两图面位置，不断这样在后台作图、切换过程。在这种机制下，如何画控件也就变成如何把控件画向后台图面。作为一般流程，要把一图像画向某一矩形区域分三个步骤。

- 记住要画位置处旧有矩形图像，这矩形图像称为背景图像。

- 在位置处画图像。
- 当图像要被擦除时，位置处恢复背景图像。

图 6-18 的 Call Stack 跳转到 twindow::draw()。

```
void twindow::draw()
{
    ......
    surface frame_buffer = video_.getSurface();
    ......
    foreach (std::vector<twidget*>& item, dirty_list_) {
        const SDL_Rect dirty_rect = item.back()->get_dirty_rect();
        clip_rect_setter clip(frame_buffer, &dirty_rect);
        ......
        // Restore.
        SDL_Rect rect = get_rect();
        sdl_blit(restorer_, 0, frame_buffer, &rect);
        // Background.
        for(std::vector<twidget*>::iterator itor = item.begin();
                itor != item.end(); ++itor) {

            (**itor).draw_background(frame_buffer);
        }
        // Children.
        if(!item.empty()) {
            item.back()->draw_children(frame_buffer);
        }
        // Foreground.
        for(std::vector<twidget*>::reverse_iterator ritor = item.rbegin();
                ritor != item.rend(); ++ritor) {

            (**ritor).set_dirty(false);
        }
        update_rect(dirty_rect);
    }
    ......
}
```

dirty_list_存储了此次 twindow::draw 时画的那些个控件，对于第一次调用 twindow::draw，dirty_list_是全部控件，后续则是那些“脏”了的。foreach 执行逐个画“脏”控件，对每次画它依次执行先恢复背景矩形然后画控件具体内容。控件要画的内容分三个方面：背景（Background），孩子（Children）和前景（Foreground）。dirty_list_，背景、孩子、前景更多细节参考**“6.3.9 刷新：‘脏’控件集”。**

画什么

一个控件要画的内容分三类：背景、孩子、前景。三部分在坐标上可覆盖，后画的会覆盖之前画的。三部分没有明确分类界线，比较来说孩子较清晰，它是容器控件特有，指那些包含的子控件。例如图 6-19 窗口中菜单控件，在自已那一层要画的内容只有孩子，包括各菜单项、右侧的垂直滚动条、底下的水平滚动条；针对菜单项这个孩子，它每一行是一个开关面板控件（tollge panel），开关面板控件当中是什么内容要按该开关是否被选中，如果没被选中，它背景是个“暗淡”颜色矩形，没有前景；否则背景是 9 个图像，其中 8 个是边框，最后一个背景（看去是“明亮”颜色矩形）；不论有没有选中孩子都是内中的 7 个标签控件。

图 6-19 菜单

```
[toggle_panel_definition]
    id = "default"
    description = "Default panel"

    [resolution]
        [state_enabled]
            # 使能，没获得输入焦点，没被选中
            [draw]
                [rectangle]
                    # “暗淡”颜色矩形
                    fill_color=21,19,19,255
                    h=(height)
                    w=(width)
                    x=0
                    y=0
                [/rectangle]
            [/draw]
        [/state_enabled]
        [state_disabled]
            ......
        [/state_disabled]
        [state_focussed]
            ......
        [/state_disabled]
        [state_enabled_selected]
            # 使能，没获得输入焦点，被选中
            [draw]
                [image]
                    # 右上角拐角图像
                    name=misc/selection-3-border-topleft.png
                    x=0
                    y=0
                [/image]
                ......
                [image]
                    # 背景图像（看去是“明亮”颜色矩形）
                    name=misc/selection-3-background.png
                    h=(height-4)
                    w=(width-4)
                    x=2
                    y=2
                [/image]
```

```
            [/draw]
        [/state_enabled_selected]
        [state_disabled_selected]
            ......
        [/state_disabled_selected]
        [state_focussed_selected]
            ......
        [/state_disabled_selected]
    [/resolution]
[/toggle_panel_definition]
```

开关面板控件模板 WML 代码（<data>/gui/default/widget/toggle_panel_default.cfg）。

背景、孩子、前景是种层次分类，它没指出要画的具体元素（图元）是什么，图元拼凑出层中内容。图元有线条（[line]）、矩形（[rectangle]），圆圈（circle），图像（[image]），文本（[text]）。开关面板控件 WML 代码中有[rectangle]、[image]范例。

若干图元组成一层，三层（背景、孩子、前景）组成一控件要画的内容。

控件模板 WML 代码中和内容相关部分怎么写，这和具体控件相关，更确切说是对应该控件模板的 C/C++实现决定了要什么写。那 C/C++是如何决定的，WML 内容代码是如何被转换成 C/C++代码，WML 写的是模板、具体数值如何挂接到模板从而最终显示出“独有”界面？让以“关卡目标”窗口中的 id=message 这个标签控件来分析。

步骤一：[draw]块生成 tstate_definition 对象

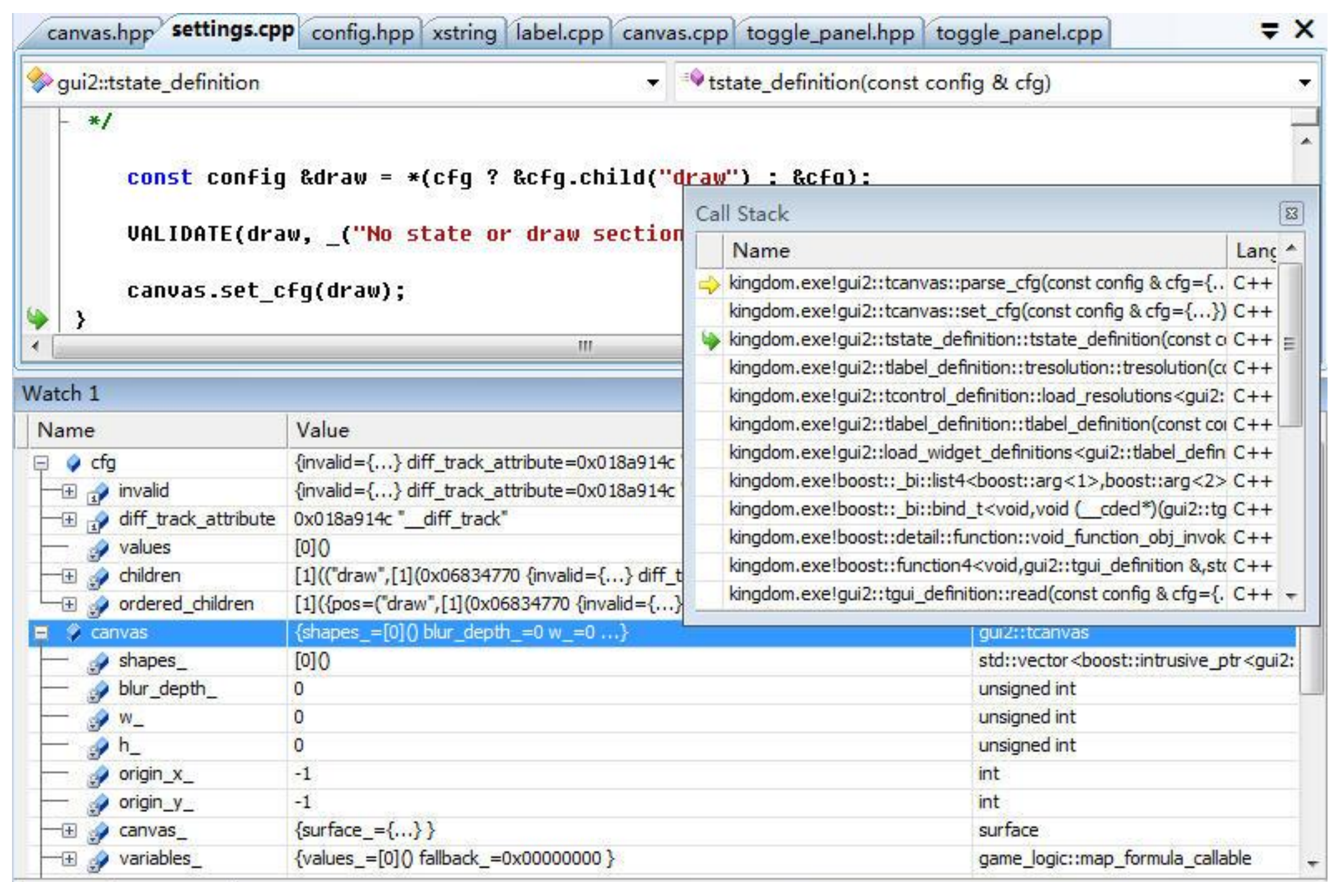


图 6-20 render-tstate_definition

[draw]块生成 tstate_definition 是在加载控件模板时执行的。一个 tstate_definition 对象封装一个画布对象（gui2::canvas），一个画布对象对应 WML 中一个[draw]块。通过 Call Stack 可追溯到[draw]块生成的 tstate_definition 它是归属于 tresolution_definition_。

```
struct tresolution_definition_
    : public reference_counted_object
{
    ......
```

```
    std::vector<tstate_definition> state;
}
```

进入 tcanvas::parse_cfg，在那里可看到 C/C++代码能支持的图元，同时决定了编写 WML 时[draw]块中能出现的图元。

Call Stack 跳到 tresolution::tresolution，函数代码决定 label 控件只支持两块画布：state_enabled、state_disabled，也就意味着 WML 中只会支持[state_enabled]、[state_disabled]块。

```
tlabel_definition::tresolution::tresolution(const config& cfg)
    : tresolution_definition_(cfg)
{
    state.push_back(tstate_definition(cfg.child("state_enabled")));
    state.push_back(tstate_definition(cfg.child("state_disabled")));
}
```

步骤二：画布从 tstate_definiton（控件配置）复制到 gui2::tcontrol 对象

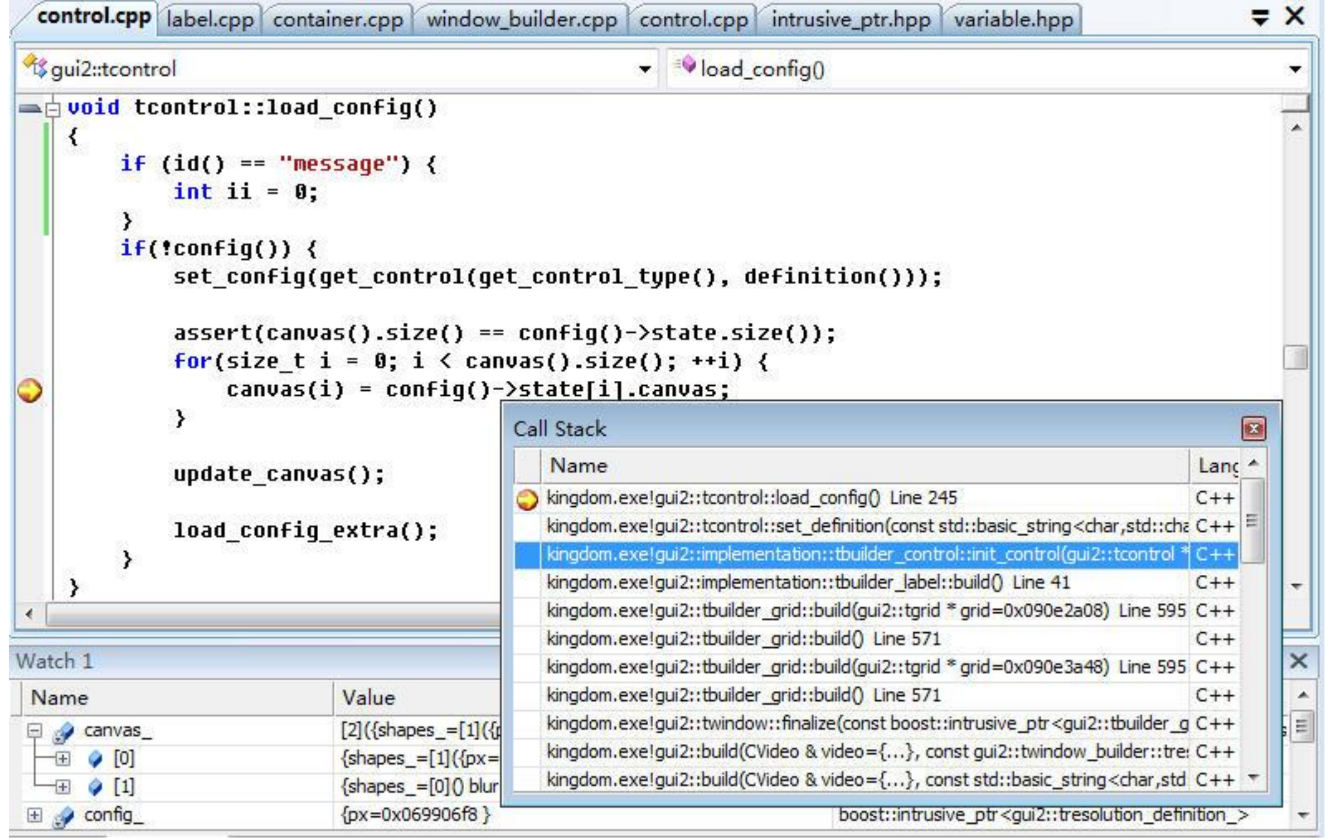


图 6-21 load_config

断点指向的 for 循环执行把画布从 gui2::tresolution_definition 中的 tstate_definiton 复制到 gui2::tcontrol 对象的 canvas_变量。

从 load_config()没看到设置 gui2::tcontrol 用于存放画布 canvas_变量项数代码，也就是在 load_config 之前已经为 canvas_设置了项数（虽然都是空项），这个设置是在 tcontrol 构造函数。

```
tcontrol::tcontrol(const unsigned canvas_count)
    : label_()
    ......
    , canvas_(canvas_count)
{
    ......
}
```

对 tlabel，canvas_count 值是 2。

看 Call Stack 能知道这个复制过程是在构造窗口时执行的，具体是在由控件构建对象（tbuilder_label）生成控件实现对象（tlabel）的辅助函数 tbuilder_control::init_control。在执行时间上有个特例，就是窗体控件，窗体控件是在 twidnow 构造函数而不是 tbuilder_control::init_cotrol，但追溯到更上层可以认为是在构建窗口过程时执行复制（gui2::build）。

步骤三：具体数值如何挂接到模板

图元（tline/tcircle/trectangle/ttext/timage）的 draw 操作实现把具体数值挂接到模板，同时把内容显示到界面。

text_对应的 WML 语句 text=(text)使用了公式（参考“1.3.3 公式”），公式只使用一个自变量，名称是 text，只要从当前可用自变量映射中取出变量名 text 对应的值就是 text_值。variables 是当前可用自变量映射。

从 variables 可看到名称是 text 的自变量它的值类型是 TYPE_STRING，也就是 string_变量存储了这个自变量的值，那个值就是“*Victory:\n<0,255,0>……”也就是图 6-22 中 Watch 窗口显示的 text 变量值。

模板中以公式去定义某一显示项，在要显示时刻提供公式中要求的自变量的具体数值，两者结合实现了数值挂接到模板。

已知道挂接是从当前可用自变量映射中搜索 first 方法，显示项要求的自变量是何时加入当前可用变量映射的？针对到此个 id=message 标签控件，自变量 text 是何时加入的。——tcontrol::update_canvas()。

```
void tcontrol::update_canvas()
{
    const int max_width = get_text_maximum_width();
    const int max_height = get_text_maximum_height();

    // set label in canvases
    foreach(tcanvas& canvas, canvas_) {
        canvas.set_variable("text", variant(label_));
        canvas.set_variable("text_markup", variant(use_markup_));
        canvas.set_variable("text_alignment"
                , variant(encode_text_alignment(text_alignment_)));
        canvas.set_variable("text_maximum_width", variant(max_width));
        canvas.set_variable("text_maximum_height", variant(max_height));
        canvas.set_variable("text_wrap_mode", variant(can_wrap()
            ? PANGO_ELLIPSIZE_NONE : PANGO_ELLIPSIZE_END));
    }
}
```

“canvas.set_variable("text", variant(label_))”中 label_从哪来？——构建窗口时在 pre_show 以“独有”字符串为参数调用 tcontrol::set_label 设置 label_，再向上追溯“独有”字符串则来自构造对话框 ttransient_message 时的参数 message。注意 tbuilder_control::init_control 也会调用 tcontrol::set_label 设置 label_，但那里设置的 label_是空值（[text]块中没有 label 字段）。另外 tcontrol::set_label 修改了 label_后需要调用 update_canvas()把最新 y 值更新到可用自变量映射。

update_canvas()形成自变量主要是和文本相关，它只是形成可用自变量映射的一个情况而已，系统中还存在其它的填充自变量映射函数，像 gui2::get_screen_size_variables，这些函数按需调用，它们协力形成 tline/tcircle/trectangle/ttext/timage 执行 draw 操前有个“完整”的可用自变量映射。

ttext::draw 中参数 canvas 是个临时创建的中性图面，draw 把自已内容“画”向这个图面，上层函数 tcontrol::impl_draw_background 会把 canvas 图面“画”向后台图面，从而实现控件内容渲染到界面。

5.5.2 刷新：“脏”控件集

在图 6-19，当鼠标落按钮上时，就要显示相应按钮的浮动状态；切换武将列表中行时，就要显示切换前、后两行状态，——要显示这些操作后状态就要刷新窗口界面，为提高效率，这个刷新不要刷新整个窗口，只需重画必须画的那些控件，这些必须重画的控件就称为“脏”控件集。

dirty_list_

twindow::draw 内如何渲染时有说到“dirty_list_存储了此次 twindow::draw 时画的那些个控件，对于第一次调用 twindow::draw，dirty_list_是全部控件，后续则是那些‘脏’了的。foreach 执行逐个画‘脏’控件，对每次画它依次执行先恢复背景矩形然后画控件具体内容。”dirty_list_ 变量就用于存储刷新窗口时要用的“脏”控件集。

```
std::vector<std::vector<twidget*> > dirty_list_;
```

dirty_list_是个二级 vector。这里就有个疑问，控件是一个一个的，要表示个体的集合只要一级 vector 就行了，为什么要二级？——一个窗口中控件被组织成树状结构，dirty_list_把“脏”控件组成链状，一条链表示了一级，多条链就形成了二级结构。

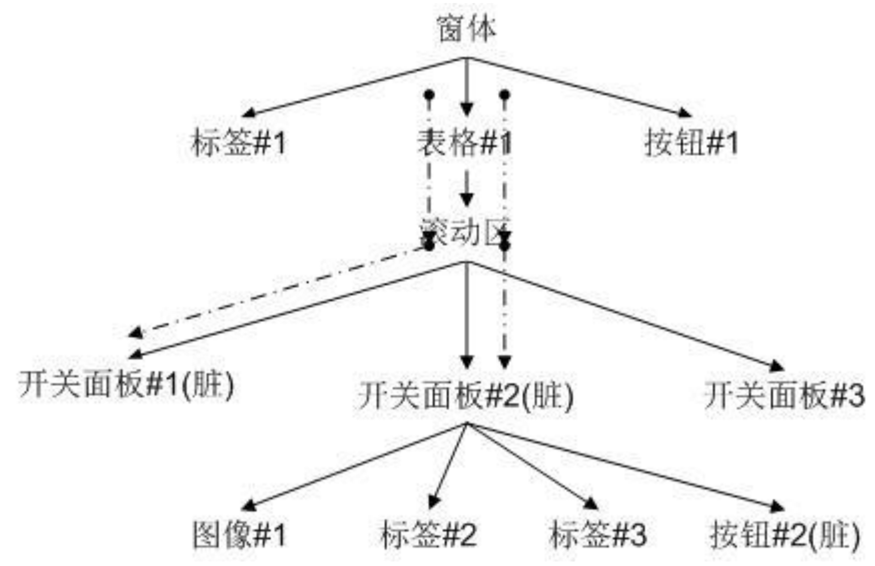


图 6-28 一个窗口中控件拓扑结构

图 6-28 表示一个窗口中控件拓扑结构。假设某一操作导致“脏”控件是开关面板#1、开关面板#2、开关面板#2 中的按钮#2，那这时形成的 dirty_list_是以下内容。

[0]	窗体、表格#1、开关面板#1
[1]	窗体、表格#1、开关面板#2

注意形成的此个 dirty_list_。按钮#2 没有形单独一条链，因为它归属的开关面板#2 已形成一条链。例子中两条链存在着重复控件，像窗体、表格#1。

如何形成 dirty_list_

形成 dirty_list_分两种情况，程序识别标志就是 need_layout_变量。一种是重布局时形成 dirty_list_；一种是按需收集“脏”控件。

```
void twindow::draw()
{
    ...
    if (need_layout_) {
        ......
        dirty_list_.push_back(std::vector<twidget*>(1, this));
    } else {
        layout_children();
        std::vector<twidget*> call_stack;
        populate_dirty_list(*this, call_stack);
    }
    ......
}
```

情况一：重布局时形成 dirty_list_

此时窗口中所有控件都是“脏”控件，形成 dirty_list_时只需一条链就行、而且这条链中只需存在一个窗体控件。

```
dirty_list_.push_back(std::vector<twidget*>(1, this));
```

this 就是窗口这个 twidget。

情况二：按需收集“脏”控件形成 dirty_list_

此时收集“脏”控件分两个步骤，每个步骤由一个函数来实现（layout_children 和 populate_dirty_list），而且两个都是 twidget 成员函数（layout_children 是虚函数）。

```
void  twidget::populate_dirty_list(twindow&  caller,  std::vector<twidget*>&
call_stack)
{
	if (visible_ != VISIBLE) {
		return;
	}
	if (get_drawing_action() == NOT_DRAWN) {
		return;
	}
	call_stack.push_back(this);
	if (dirty_) {
		caller.add_to_dirty_list(call_stack);
	} else {
		// virtual function which only does something for container items.
		child_populate_dirty_list(caller, call_stack);
	}
}
```

populate_dirty_list 是只有 twidget 实现的函数，它形成脏控件链，并放入窗口的 dirty_list_。参数 caller 就是控件归属于的窗口，call_stack 则是轮到该控件时已形成的父控件链，像图 9-30，如果正处理的控件是开关面板#1，那么 call_stack 就是{窗体、表格#1、滚动区}（当中内容除了控件，还会包括 tgrid）。

populate_dirty_list 的处理逻辑

1. 把自己放入控件链 call_stack。
2. 如果自己是脏的，不用继续搜索，此个 call_stack 形成一条链，这条链作为一个 vector 单元放入窗口 dirty_list_。
3. 如果自己是不脏的，调用 child_populate_dirty_list 继续形成可能经过它的“脏”控件链。

通过分析可得出个结论，只要正确设置了各控件 dirty_值，populate_dirty_list 可说是执行了形成 dirty_list_的所有动作，那为什么要在它之前调用 layout_children？——相比于布局整个窗口，layout_children 提供了一种只是布局某个控件的操作。要触发布局控件可能有两种情况：1）该控件被设置需要重布局，像 tlistbox 中的“void invalidate_layout() { need_layout_ = true; }”；2）内部操作设了需要重布局标志，像 tlistbox 中的 set_list_builder 会把 need_layout_置为 true。不论哪种方式，在把 need_layout_置为 true 时只是把该变量置为 true 而已，不会执行真正的布局操作，而可以想象的，经过重布局后该控件极可能会产生很多“脏”控件。layout_children 执行控件的重布局操作，从而及时把应该变脏的置为脏控件，以便后续的 populate_dirty_list 能正确收集。

layout_children 执行的是布局，只有较复杂控件才需要重布局，像表格（listbox），树形（tree）。need_layout_不是 twidget 成员变量，它是那些需要布局控件自已的，一个经常出现的同名但不同参数的“void layout_children(const bool force)”也不是 twidget 成员函数，但 void layout_children() 是 twidget 虚函数。

background/children

形成 dirty_list_后接下是向界面画这些“脏”控件，在之前“twindow::draw 内”时有说过，每个控件要画的内容被归在三个层：背景（background）、孩子（children）、前景（foreground），那三个层有什么特点？控件中图元是按什么规则归入到哪个层？通过看 twindow::draw 中处理 dirty_list_后续代码可得到更多答案。

```
void twindow::draw()
{
```

```
    ......
    foreach(std::vector<twidget*>& item, dirty_list_) {
        ......
        // Background.
        for (std::vector<twidget*>::iterator itor = item.begin();
            itor != item.end(); ++itor) {
            (**itor).draw_background(frame_buffer);
        }
        // Children.
        if (!item.empty()) {
            item.back()->draw_children(frame_buffer);
        }
        // Foreground.
        for  (std::vector<twidget*>::reverse_iterator  ritor  =  item.rbegin();
ritor != item.rend(); ++ritor) {
            (**ritor).set_dirty(false);
        }
        ......
    }
    ......
}
```

背景（background）

几乎所有图元都集中在这里。

tcontrol::impl_draw_background 是渲染背景主要函数。

```
void tcontrol::impl_draw_background(surface& frame_buffer)
{
    canvas(get_state()).blit(frame_buffer, get_rect());
}
```

控件往往被分成数个状态。举个例子，代码 6-x 中的开关面板（toggle_panel）分成六个状态：未选中时使能（ENABLED），未选中时禁用（DISABLED），未选中时获得焦点（FOCUSSED），选中时使能（ENABLED_SELECTED），选中时禁用（DISABLED_SELECTED），选中时获得焦点（FOCUSSED_SELECTED）。任一时刻开关面板只能处于六种中的一种，get_state()就用于返回当前状态。而对每种状态，控件都对应一张画布，也就是说控件内有六张候选画布，canvas(get_state())就用于返回当前状态对应的那张画布。

由于在位置上父控件肯定全包含子控件，在“脏”控件链中渲染背景次序是由根向叶。

孩子（children）

孩子层是虚拟层，所有控件在该层不放任何图元。存在孩子层的原因为了能够一次性画出“复合”控件，像包含网格的控件。

```
void tgrid::impl_draw_children(surface& frame_buffer)
{
    ......
    foreach(tchild& child, children_) {
        twidget* widget = child.widget();
        ......
        widget->draw_background(frame_buffer);
        widget->draw_children(frame_buffer);
        widget->set_dirty(false);
    }
    ......
}
```

借用 children 层，渲染逻辑在该控件上可展开渲染三层，从而实现画出该控件的所有图元。以放置图元功能来说，由于 children 没有图元，原子控件的 impl_draw_children 也就没啥事可干了，是个空函数。

渲染时只画“脏”控件链的末尾控件。画一个控件的孩子层时不画该控件的背景、前景。

小结

如没特殊需要，设计控件时把图元全放在背景层。

设计背景层时按状态划分数张画布。控件任一时刻只能属于某种状态。

孩子层是虚拟层，存在孩子层的原因是为了能够一次性画出“复合”控件。渲染时只画“脏”控件链的最尾控件的孩子层。

5.6 事件

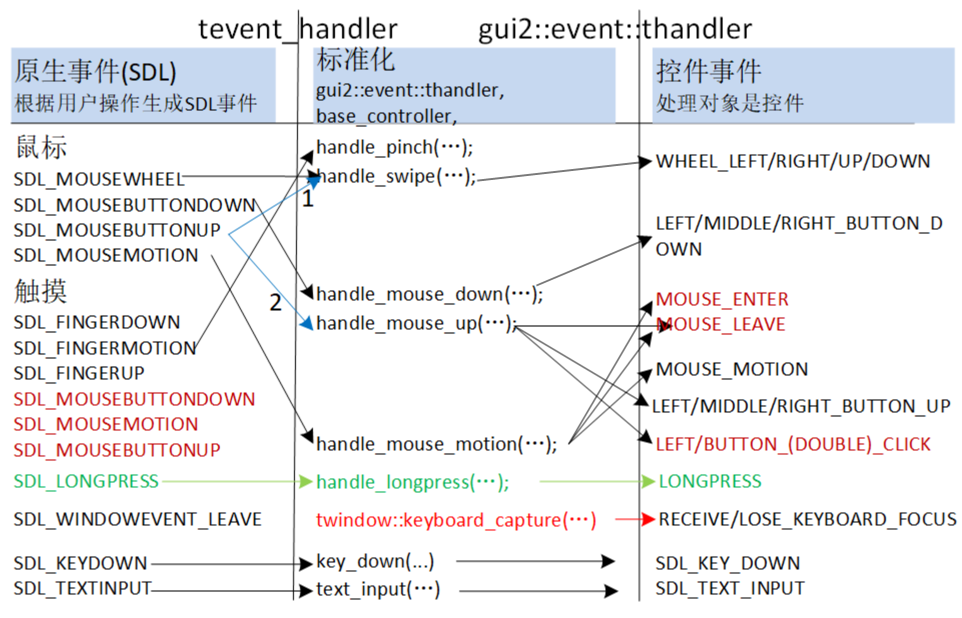


图 5-29 事件框架

5.6.1 原生模型

事件系统的基础是 SDL 提供的原生模型。对模型，首先要知道它能提供哪些事件。

图 5-29 左侧是 SDL 提供的原生事件。和鼠标相关的有滚动、单击和移动。发生触摸操作时，SDL 除了产生三个 FINGER，还会产生图中以红色标注的三个 MOUSE。除了以上两类，在 app 可以非全屏的 PC 平台，鼠标一旦离开 app 私有窗口，会发出 SDL_WINDOWEVENT_LEAVE。绿色标注的 SDL_LONGPRESS 不是 SDL 产生，关于它参考底下的“**5.6.5 长按和拖拽**”。

FINGER 时，只有第一个触点才会触发红色标注的 MOUSE 事件，而且它们会在相应的 FINGER 之前被发送，示例见下面的图 5-10。此时 MOUSE 事件附带的各字段情况：x、y 字段和 FINGER 同值，which 字段值是 SDL_TOUCH_MOUSEID。SDL 同时发 MOUSE 的一个目的是方便上层统一用它们进行跨平台编程。第一个触点指的落在屏幕的第一个手指，当所有手指离开屏幕后，接下落向屏幕的又是第一个触点。

SDL_MOUSEMOTION、SDL_FINGERMOTION 是和移动相关的两个事件，它们会附带指示移动了多少的字段(rel=移动后坐标-移动前坐标)。rel 有正、负值，计算时按照自然序，正数表示向小的方向移动，即 x 向左，y 向上。

SDL_MOUSEWHEEL 附带 x、y 和 direction。x/y 只有三种值，-1、0 和 1，它们指示了滚动方向，不能反映幅度。direction 值或是 NORMAL 或是 FLIPPED，和如何确定 x、y 正负有关。FLIPPED 表示此时 x/y 是自然序，NORMAL 时相反，即向上滚动时 y 是 1，向下时是-1。实际使用时，内容滚动的方向往往和 WHEEL 相反，因而 NORMAL 才是需要值。注意，触摸不会产生 SDL_MOUSEWHEEL。

偏移抖动(Jitter)。设备不可能移动一个像素就向上发 motion，粒度是多少，各操作系统会有不同。由于这个抖动，而 app 处理 down、up 时会希望对正处理的坐标、在之前会有个到它的 motion，于是存在个疑问：原生模型在产生 down、up 之前，是否肯定已产生过到该坐标的 motion？——基本会产生，但为确保安全，上层在 down、up 前还是须要检查，必要时加发个额外移动(extra motion)。

mousedown、mouseup 配对。设备发了 down 后，接下会有 up，那是不是 down 后有总有 up？——基本会产生，但为确保安全，上层须检查两个 down 之间有没有 up，没有时加发个额外 up(extra mouseup)。有了这个额外 up，app 可认为不会收到连续两个 down，但不保证会收到多个 up 然后一个 down。另外，一旦两个 down 之间收到了窗口事件（SDL_WINDOWEVENT），窗口事件能复位鼠标状态，这时中间不加发额外 up。

5.6.2 标准化

对 app 来说，原生事件太低级，直接处理会需要不少额外操作。下表列出了 app 会感兴趣事件。

事件	描述	tevent_handler 中函数
Pinch(捏合)	双点捏合，一般用于缩放	handle_pinch
Swipe(滑动)	快速移动，可监测滑动的方向、幅度	handle_swipe
Click/Tap (点击)	DOWN 到 UP 没离开过一个控件	
Pan(拖移)	慢速移动，可监测精确到像素偏移量	handle_mouse_down+handle_mouse_motion
LongPress(长按)	按在某处超过一定时间	handle_longpress

标准化阶段的任务是把原生事件翻译为上面这些事件。对要同时支持游戏和非游戏开发的 SDK，这些事件会同时发向 GUI 中窗口和场景中大地图（见“图 3-1 发分事件”）。在实现上，tevent_handler 把原生事件转为数个函数，窗口、大地图则从 tevent_handler 派生，以着重载函数的方式进一步处理这些事件。

```
void handle_pinch(const int x, const int y, const bool out);
void handle_swipe(const int x, const int y, const int xlevel, const int ylevel);
void handle_mouse_down(const SDL_MouseButtonEvent& button);
void handle_mouse_up(const SDL_MouseButtonEvent& button);
void handle_mouse_motion(const SDL_MouseMotionEvent& motion);
void handle_longpress(const int x, const int y);
```

handle_pinch。两个触点间的距离记为 distance，前、后两次 distance 差的绝对值大于 pinch_threshold(80)时会被调用。x、y 是调用那一刻发生移动的触点正处在的坐标。out 是 true 表示放大，否则缩小。

```
SDL_MOUSEBUTTONDOWN, (647, 439)
87001, SDL_FINGERDOWN, (647, 439)
SDL_MOUSEMOTION, xy(668, 360), ref(21, -79)
87367, SDL_FINGERMOTION, (668, 360), delta(21, -79)

SDL_MOUSEMOTION, xy(690, 261), ref(22, -99)
87437, SDL_FINGERMOTION, (689, 261), delta(21, -99)
SDL_MOUSEMOTION, xy(735, 132), ref(45, -129)
87508, SDL_FINGERMOTION, (735, 132), delta(45, -129)
SDL_MOUSEBUTTONUP, (735, 132)
87555, SDL_FINGERUP, (735, 132)
```

图 5-10 一次向上滑动（触摸）的事件序列

handle_swipe。两种情况会被调用。1）SDL_MOUSEWHEEL。2）SDL_MOUSEBUTTONUP，累计 up 前的 threshold_span(120)毫秒内 motion 偏移，如果 x、y 有一个方向超过 10。图 5-10 是执行一次向上滑动时 SDL 发出的事件序列，87555 是松开时刻，要累计距离它 120 毫秒内的 motion，包括 87437、87508，算出累计偏移是(66, -228)。对 app 来说，相比具体偏移了多少像素，可能更关心以它量化出的幅度，参数 xlevel、ylevel 就指示了这两幅度值。

偏移 delta	level	
<10	0	不会调用 handle_swipe
<70	1 + (abs_delta - 10) / 6	[1, 10]

>=70	swipe_max_normal_level + 1	11

delta 是除过 hdpi_scale 的配置尺寸。swipe_max_normal_level 固定是 10。delta_2_level 执行把 delta 换算成 level，它把 level 分为两类，偏移<70 时归为普通级别，level 值从 1 到 10，大于 70 的归为大幅移动，值是 11。

handle_mouse_down。收到 SDL_MOUSEBUTTONDOWN 时被调用。

handle_mouse_up。收到 SDL_MOUSEBUTTONUP 时被调用，如果累计偏移足够，在它之前会先调用 handle_swipe。

handle_mouse_motion。收到 SDL_MOUSEMOTION 时被调用

SDL_MOUSEBUTTONUP 的嵌套问题。在具体 app，松开手指/鼠标时有可能会阻塞程序，像按下按钮后弹出个模态对话框，关闭对话框后才回到继续执行松开代码。为避免出现嵌套，处理松开使用了两条规则。1）处理松开的代码统一放在不论触摸还是非触摸设备都支持的 SDL_MOUSEBUTTONUP，完全忽略 SDL_FINGERUP。2）松开依次执行三个操作：handle_swipe，fingers_.clear()和 handle_mouse_up。handle_swipe 时不要执行会导致阻塞的操作，要是有阻塞请放在后面的 handle_mouse_up。代码角度上看，阻塞操作特点是会调用 events::pump。

5.6.3 控件事件

这里和后面的事件链都是针对 GUI(gui2::event::thandler)，不涉及场景（base_controller）。

虽然 tevent_handler 已把原生事件翻译成数个函数，但如果 app 直接去重载那几个函数，还是太累了。为简化编程，依据 gui 编程特点，进一步细化出具体到控件的事件。

事件	描述	回调函数的额外参数
WHEEL_LEFT/RIGHT/UP/DOWN	区分了方向的 swipe	5)handle_swipe 中的(x,y)， 6)handle_swipe 中的(xlevel, ylevel)绝对值
LEFT/MIDDLE/RIGHT_BUTTON_DOWN	按下左键(手指)/中键/右键	5)handle_mouse_down 中的(x,y)
LEFT/MIDDLE/RIGHT_BUTTON_UP	松开左键(手指)/中键/右键	5)handle_mouse_up 中的(x,y)
MOUSE_MOTION	移动鼠标/手指	5)handle_mouse_motion 中的(x,y), 6)handle_mouse_motion 中的(xrel,yrel)
MOUSE_ENTER/LEAVE	进入控件，离开控件	5)鼠标/手指正处在坐标。注：LEAVE 时(-1,-1)表示离开窗口
LEFT _BUTTON_CLICK	单击左键(手指)	5)handle_mouse_up 中的(x,y), 6)单击类型
LEFT_BUTTON_DOUBLE_CLICK	双击左键(手指)	5)handle_mouse_up 中的(x,y)
OUT_OF_CHAIN		5)发生的事件

以上就是图 5-29 右侧的控件事件，它们共同特点是必定发向某个控件。

app 如何编程 gui 事件？一个主要部分是向自个感兴趣事件挂接处理例程，当事件发生时这些例程会被自动调起，调起时前四个是固定参数（见后面“事件链”），“回调函数的额外参数”指出了该事件第五个及后面的参数。

可把控件事件分为三类，一是和“原生”对应的 WHEEL、DOWN、UP、MOTION，二是进入和离开控件，三是单击和双击。后面两种从原生衍生出来，这意味着一个操作可能会产生多个事件，这时发送事件的顺序是先原生，后 enter/leave，最后是单击/双击。举个例子，一次鼠标松开会导致需要发原生的 up、leave、以及 click，这时是先发 up，后 leave，最后 click。同理，一次 motion 可能导致原生 motion，离开一个控件的 leave，进入另一个控件的 enter，这时发送次序是 motion、leave、enter。

代码保证控件会收到配对的 enter、leave，也就是说，控件 A 收到了 enter，那后面一定会有 leave。这个规则对实现拖移鼠标会较有用，拖移鼠标往往从 down 开始，leave 结束，过程中通过 motion 知道鼠标在哪里。为什么是 leave，而不是 up？1）可能是按着鼠标离开控件。2）鼠标可能离开 app 窗口，这时不会产生 up，但会产生一个坐标是(-999999,-1)的 leave。

鼠标离开 app 窗口时，只会发一个坐标是(-999999,-1)的 leave，不会发其它事件，像 motion、up。为方便 app 判断是否是(-999999,-1)，提供了叫 is_mouse_leave_window_event 的宏。

鼠标从 down 到 up 都没离开过一个控件(focus_)，那就会在该控件产生一个单击事件(click)。什么时候发送 double_click？之前已经发过一次 click，此次 up 又可以导致发送一次 click，而这两次 click 间隔小于 double_click_time，那第二次 click 会变成发送 double_click。这意味着，在发送 double_click 前一定已经发了一次 click。哪里设置 double_click_time？gui.dat 中 settings 块的 double_click_time 字段，当前值是 500(毫秒)。

处理控件事件时，Rose 允许在 LEFT_BUTTON_CLICK、LEFT_BUTTON_DOUBLE_CLICK、RIGHT_BUTTON_UP 弹出新窗口，其它事件要弹出新窗口的，须使用“**3.8.4**”小节介绍的 Post+app_OnMessage 机制。

mouse_focus_。app 会遇到这么种需要，**当鼠标是拖着离开 A 控件，还是希望把后面的鼠标事件（motion、up）导向 A**。举个例子，正在编写一个通过拨针转圈方法，然后修改时间的钟盘控件，按下手指开始拨针了，某一刻手指离开钟盘，但只要还是做出转圈手势，那还是要能操作钟盘。程序如何实现？按下鼠标时调用 twindow::mouse_capture，把当前控件（钟盘）设为 mouse_focus_，只要鼠标不松开/离开窗口，后续鼠标事件都会自动导向这个控件。要注意，它不影响 click，click 要求是鼠标从 down 到 up 都没离开过该控件。什么时候 mouse_foucs_回到 nullptr？——松开鼠标或鼠标离开窗口就会置 nullptr。

motion、down、motion、up，鼠标总是这么周而复始地操作，但可以这么认为，up 是个复位点。一旦收到 up，代码要恢复到初始状态，像会发 mouse_leave，mouse_focus_、focus_要置 nullptr。复位点除了 up，还有一个是弹出新窗口 B，之前窗口 A 会进入初始状态。此种情况要注意，移动鼠标关闭 B（弹式出菜单）时，如果立即按下鼠标，由于坐标没变过，tevent_handle 不会发额外 motion，A 的 mouse_focus_将是初始状态时 nullptr！目前处理是忽略这个 down。

希望代码能够精确统计 down、up，并进行严格检查。举个例子，用一个叫 is_down_布尔变量指示是否正按下鼠标。收到 down 时，诊断该变量必须是 false，然后置 true。收到 up 时，诊断该变量必须是 true，然后置 false。愿望美好，但很难实现。在 PC，鼠标会离开窗口，分按着鼠标离开还是不按着离开；回到窗口，也要分按着和不按着。除离开，窗口失去焦点、获得焦点、最小化也会产生影响。而且目前测试下来，SDL 事件行为在不同平台不完全一致，像 Andoird，横、竖屏切换时会收到离开窗口事件。为此，如果真想做到 down、up 配对，最好等 SDL 严格统一了各平台上事件模型。

OUT_OF_CHAIN。以上事件只能发向鼠标击中的、以及父控件组成的控件链，有时要发向这之外的控件。举个例子，长按列表行后弹出菜单，用户可能是在列表之外的地方按下鼠标，这时也该能收到这鼠标按下事件，以便关闭菜单。OUT_OF_CHAIN 第五个参数指示发生了什么事件，包括 LEFT/MIDDLE/RIGHT_BUTTON_DOWN、WHEEL_DOWN/UP/LEFT/UP。

RECEIVE_KEYBOARD_FOCUS/ LOSE_KEYBOARD_FOCUS。对键盘焦点有兴趣的控件（目前只有 text_box）需处理得到焦点、失去焦点事件。对失去焦点，第 6 个参数 extra_widget 指示了下个要获得焦点的控件，text_box 可用它判断此次失去焦点时要不要关软键盘。这两个事件不发源于 SDL，由 app 主动调用 twindow::keyboard_capture 产生。

事件系统会保证遵守影响 app 代码的规范，1）enter、leave 配对，2）down、up 之前应该有 motion 到过该坐标。由于无法做到 down、up 严格配对，down、up 处理例程要能容许连续收到多个 down 或多个 up。

5.6.4 事件(tsignal)、事件队列(tsignal_queue)、事件链(event_chain)

首先说下它们要解决什么问题。列表（listbox）控件由多行（toggle_panel）组成，“解散”按钮是 toggle_panel 中控件。当在“解散”按下鼠标时，不仅仅是按钮，它的父控件 toggle_panel、listbox 等都要处理这事件。

控件	位置	可能执行的操作

button	当前控件	执行解散
toggle_panel	第一代父控件	选中该行
listbox	第二代父控件	获得输入焦点

以上是关于事件要如何在当前控件、父控件间的先后传递问题。接下让考虑一旦按下“解散”，表示要解散该部队，那么显示该部队的 toggle_panel 就会被删除，这时“LEFT_BUTTON_CLICK”就不该继续传向 toggle_panel 这个父控件，即要能被中途中止。

把事件链相关的代码分为三部分：一是控件如何存储事件，二是系统如何依次调用事件处理例程，三是 app 如何把自个的事件例程加入到例程集合。

一、控件如何存储事件

任何控件都派生于 twidget，twidget 派生于 tdispatcher，tdispatcher 负责存储事件。

事件很多种，一控件不可能什么事件都处理，tdispatcher 只存储它感兴趣事件。根据事件处理例程的不同类型，事件被归到 6 个队列。这 6 个队列是控件存储事件的变量。

变量	例程类型	事件实例
signal_queue_	tsignal_queue<tsignal_function>, int val	tset_event MOUSE_ENTER, MOUSE_LEAVE, LEFT_BUTTON_DOWN, LEFT_BUTTON_UP, LEFT_BUTTON_CLICK, LEFT_BUTTON_DOUBLE_CLICK
signal_mouse_queue_	tsignal_queue<tsignal_mouse_function>, tpoint& coordinate, tpoint& coordiante2	tset_event_mouse SDL_MOUSE_MOTION, MOUSE_MOTION, SDL_LEFT_BUTTON, SDL_LEFT_BUTTON_UP, SHOW_TOOLTIP
signal_keyboard_queue_	tsignal_queue<tsignal_keyboard_function>, SDLKey key, SDLMod modifier, Uint16 unicode	tset_event_keyboard SDL_KEY_DOWN
signal_textinput_queue_	tsignal_queue<tsignal_textinput_function>, const char* text	tset_event_textinput SDL_TEXT_INPUT
signal_notification_queue_	tsignal_queue<tsignal_notification_function>, void*	tset_event_notification NOTIFY_REMOVE_TOOLTIP
signal_message_queue_	tsignal_queue<tsignal_message_function>, tmessage& message	tset_event_message MESSAGE_SHOW_TOOLTIP

事件处理例程的前四个参数是固定的，第一个 dispatcher(tdispatcher&)指示发送事件的窗口，第二个 event(tevent)指示事件码，第三个 handled(bool&)指示如果例程处理过该事件，不希望后面的集合继续处理（什么是集合，见后续说明），置 true，第四个 halt(bool&)指示如果例程想系统在它处理过后立即中止处理该事件，置 true（同时必须把 handle 置 true）。“,”后面是不同例程自增加的参数。

6 个列队是 tsignal_queue 这个模板类实例，接下让深入事件队列，tsignal_queue。

```
template<class T>
struct tsignal_queue
{
    tsignal_queue()
        : queue()
    {}
    std::map<tevent, tsignal<T> > queue;
    ......
}
```

模板参数 T 就是处理例程。队列有且只有一个成员变量：queue。queue 是个映射，映射的 key 是事件码，像 LEFT_BUTTON_CLICK、MOUSE_MOTION，映射的 value 是 tsignal<T>。那 tsignal<T>是什么？

```
template<class T>
```

```
struct tsignal
{
    tsignal()
        : pre_child()
        , child()
        , post_child()
    {}
    std::vector<T> pre_child;
    std::vector<T> child;
    std::vector<T> post_child;
};
```

以 signal_queue_的事件 T 代入，它的 tsignal<T>就是三个类型是 tsignal_function 函数的集合。回到 tsignal_queue 中的 queue 变量，到此可得出个结构：**key 是事件码，value 是处理该事件的函数集合**。

紧接会有个疑问，tsignal 为什么把要函数分到三个集合，这就涉及到第二个问题。

二、系统如何依次调用事件处理例程

根据调用时机，事件函数被分为三个集合，就是 tsignal 中的 pre_child、child 和 post_child。

位置	调用时机
pre_child	第一被调用。所在控件是当前控件的父控件，而且是以着逆序被调用。回到“解散”按钮例子，当前控件有两代父控件，那它的调用次序先是第二代 listbox 的 pre_child、然后是第一代 toggle_panel。
child	第二被调用
post_child	第三被调用。所在控件是当前控件的父控件，和 pre_child 相反，以着顺序被调用。在书写时，pre_child 存放它作为父控件时，先于当前控件要被执行的操作。child 存放它做为当前控件时执行的操作。post_child 存储它作为父控件，后于当前控件要被执行的操作。

让考虑一个问题，为什么 pre_child、child、post_child 要使用集合？因为要分为当前控件、当前控件前处理、当前控件后处理，较容易理解为什么要把处理例程分为三类，但一类中使用集合，意味着同一控件在每一类可能存在多个处理过程，这是为什么？——原因是控件存在多样性，像要从多个父类派生、经过多个阶段，等等，它们导致出同一控件在每一类中会存在多个处理过程。

举个例子，针对 SDL_LEFT_BUTTON_DOWN 事件。窗口这个控件有个 tdistributor 类型的成员，构造这成员时会向 child 挂接 signal_handler_sdl_button_down，负责把 sdl_button_down 翻译成控件层事件。然后在自个构造函数会再向 child 挂接 signal_handler_click_dismiss，负责可能的话把单击等同关闭窗口。这两个处理例程同属窗口这个控件，却执行着不同功能。

用于存储当前控件的父控件的变量是 event_chain（事件链），它是一个 vector，单位类型 std::pair<twidget*, tevent>>。事件链是和 tevent 事件挂接在一起，它指出对该事件“垂直”上的处理架构，即它的第一个单元是第一代父控件，第二个单元是第二代，依此类推。

事件处理例程有四个固定参数，第三个是 handled，当例程把它置 ture 时，表示处理事件只限制在它所在集合，本集合的处理结束，事件不再传播向后面集合。第四个是 halt，例程把它置 true 时，表示它处理后立即不再向后传播。为更好处理拖拽，app 处理 MOUSE_MOTION、MOUSE_LEAVE 时不要让 handled、halt 置 true。

三、app 如何把自个的事件例程加入到例程集合

1)connect_signal：加入到集合。

2)disconnect_signal：从集合中移除。

connect_signal、disconnect_signal 都是 tdispatcher 中的模板型成员函数。以下是 connect_signal 的一种实现。

```
template<tevent E>
```

```
typename                          boost::enable_if<boost::mpl::has_key<tset_event,
boost::mpl::int_<E> > >::type
connect_signal(connect  tsignal_function&  signal,  const  tposition  postion  =
back_child)
{
    signal_queue_.connect_signal(E, position, signal);
}
```

它有两个参数，第一个参数事件例程，第二个是要插入的集合及位置。第一个参数同时决定了要加入到的是哪个队列。像上面的 tsignal_function，加入的是 signal_queue_，tsignal_mouse_function 则会加入到 signal_mouse_queue_。第二个参数是加入到的集合及位置，它指出两点，加入的是哪个集合，以及是该集合的头还是尾，它可以有 6 个值。

postion	集合	集合中位置
front_pre_child	pre_child	头
back_pre_child	pre_child	尾
front_child	child	头
back_child	child	尾（默认）
front_post_child	post_child	头
back_post_child	post_child	尾

以下是 connect_signal 的一个实例。感兴趣的是 LEFT_BUTTON_CLICK 事件，加入到 signal_queue_队列，位置是添加在 child 集合的前头。

```
label_->connect_signal<event::LEFT_BUTTON_CLICK>(boost::bind(&ttree_node::signal
_handler_label_left_button_click, this, _2, _3, _4),
event::tdispatcher::front_child);
```

接下让说移除。以下是它的一种实现。

```
template<tevent E>
typename                          boost::enable_if<boost::mpl::has_key<tset_event,
boost::mpl::int_<E> > >::type
disconnect_signal(connect  tsignal_function&  signal,  const  tposition  postion  =
back_child)
{
    signal_queue_.disconnect_signal(E, position, signal);
}
```

两个参数的功能基本类似 connect_signal，除了 position 功能只是指出要从哪个集合移除，是头还是尾则被忽略了，程序逻辑是从头到尾枚举该集合中的处理例程，是参数 signal 例程的就移除。

对很多控件都会关心的事件，为简单增加了几个封装函数。

函数	事件	postion
connect_signal_mouse_left_click	LEFT_BUTTON_CLICK	post_child
disconnect_signal_mouse_left_click	LEFT_BUTTON_CLICK	post_child
connect_signal_pre_key_press	SDL_KEY_DOWN	front_child

5.6.5 长按和拖拽

一旦按在某处的时间超过 longpress_time，就有可能触发长按事件。gui.dat 中 settings 块的 longpress_time 字段设置了这个时间，当前值是 300(毫秒)。

长按允许鼠标有微小偏移，这个微小偏移有可能导致鼠标从控件 A 落到控件 B。长按只是判断偏移是否在门限内(clear_click_threshold)，而不管有没有控件变化，这时它还是会向控件 B 触发长按事件。

```
label.connect_signal<event::LONGPRESS>(
    boost::bind(
        &tocr_scene::signal_handler_longpress_name, this
        , _4, _5), event::tdispatcher::back_child);

void   tocr_scene::signal_handler_longpress_name(bool&   halt,   const   tpoint&
coordinate)
```

```
{
    halt = true;
    ......
    准备图面(surf)
    window_->set_drag_surface(surf, true);
    window_->start_drag(coordinate,
boost::bind(&tocr_scene::did_drag_mouse_motion,      this,      _1,      _2),
boost::bind(&tocr_scene::did_drag_mouse_leave, this, _1, _2, _3));
}
```

类似上面代码，app 调用 connect_signal 向自个感兴趣的控件挂接长按处理例程。在例程中，halt 被置为 true，表示我已处理过，不要在事件链继续传播该事件了。coordinate 存储着触发长按那一刻鼠标落在的坐标。

这里说的拖拽有两个特点。1）支持全窗口范围。“**6.9 事件**”中拖拽限制在大地图，“**7.3.2 拖拽**”限制在 track 控件。2）拖拽时鼠标底下会有个图像。所以即使限制在大地图或 track 控件中的拖拽，如果要让支持底下有图像，也需要使用这里的逻辑。

拖拽分为三个阶段，长按、移动和松开。拖拽时在鼠标底下会有个图像，它是个悬浮控件，类型是 magnifier 风格的 image。在长按处理例程依次做三件事。1）准备好图面。2）调用 set_drag_surface 把图面关联到悬浮控件。3）调用 start_drag，它挂接 motion、leave 例程，并显示图像。为什么要分开 2 和 3？set_drag_surface 只能让 image 显示一个 blit，app 如果想叠加出更自由的 image 得自写 set_drag_surface。

```
bool tocr_scene::did_mouse_motion(const int x, const int y)
{
    return true;
}

void tocr_scene::did_mouse_leave(const int x, const int y, bool up_result)
{
}
```

motion 例程是移动时被调用，x、y 是当时落在坐标。返回值是 true 表示继续拖拽，false 则意味着不必再继续了，中止拖拽。

leave 例程是松开时调用，存在两种时机，一是正常松开鼠标，二是意外情况。正常松开时，up_result 是 true，x、y 是松开时坐标。意外情况时 up_result 是 false，x、y 指示意外情况，像鼠标离开窗口。要注意，当 motion 例程返回 false 提前中止时，将不会回调后面的 did_mouse_leave。

5.6.6 键盘事件

键盘事件的 SDL 部分参考第二章中“**2.5 键盘相关**”。

SDL 发来的事件，Rose 只关心 SDL_KEYDOWN，SDL_TEXT_INPUT，经过标准化后分别转为 SDL_KEY_DOWN，SDL_TEXT_INPUT。

twindow 有个叫 keyboard_focus_成员，存储着正拥有键盘焦点的控件，只有 keyboard_focus_以及它的父控件能处理 SDL_KEY_DOWN、SDL_TEXT_INPUT。

正有多个窗口时，只有最顶层窗口中的 keyboard_focus_能处理键盘事件。一旦最顶层窗口 keyboard_focus_==nullptr，事件依旧被取出，发现没人可处理后，丢掉。

当存在场景时，键盘事件首先被 gui2 处理，如果 gui2 没有处理该事件，base_controller 接着处理，base_controller 则是 app 自实现函数去处理（待讨论，目前不是如此，将来可能这样）。

可以把 window_设为 keyboard_focus_，这意味着挂接 SDL_KEY_DOWN 的控件必须是 windows_，而且位于的集合必须是 child。

不管 keyboard_focus_是不是 nullptr，窗口都会处理 ESC 键，行为是关闭窗口。对没有场景的窗口，按下 ESC 默认是关闭，否则忽略。app 可通过调用 set_escape_disabled 修改这个行为。

第六章 GUI：场景（Scene）

本章目标

- 如何渲染状态报告。
- 如何渲染标签控件。

场景，指包含有大地图的窗口，例子见彩图 1、彩图 2、彩图 3。场景既然是窗口，窗口中的所有控件都可用在场景中。但场景的主要部分，大地图不是控件，因为它太复杂了，用控件去实现的话，功能多样性、执行效率等方面很难达到要求。除大地图，场景中其它都是窗口控件，类似对话框，控件可以浮动在大地图上，像彩图 3 左上角的“玩家头像”按钮。

6.1 app 的场景编程

6.1.1 基本步骤

第一步：向 app 新建场景

运行 studio，在左侧树形控件的顶结点按鼠标左键，弹出菜单中进入要增加场景的 app，子菜单选择“新建场景”，假设新建的场景标识是“home”。新建成功会增加 7 或 11 个文件，一个是放入资源包的 home_scene.cfg，它是窗口脚本。另 6/10 个放入源码包，固定的分别是从 base_controller、display、tdialog 派生，可变则从 unit、unit_map 派生，因为每个组件都有 cpp/hpp，文件就有 6 或 10 个。“**6.1.3 组件**”有对这些组件的概述。

第二步：修改窗口脚本

运行 studio，单击“窗口”，进入窗口编辑器。打开 home_scene.cfg，通过不断增加、删除、修改控件，改出你希望的样子。

第三步：修改组件对应的 cpp/hpp

在 Visual Stuido，修改 cpp/hpp，让改出希望达到的效果。场景逻辑是用户操作大地图和大地图外上的控件操作共同驱动。

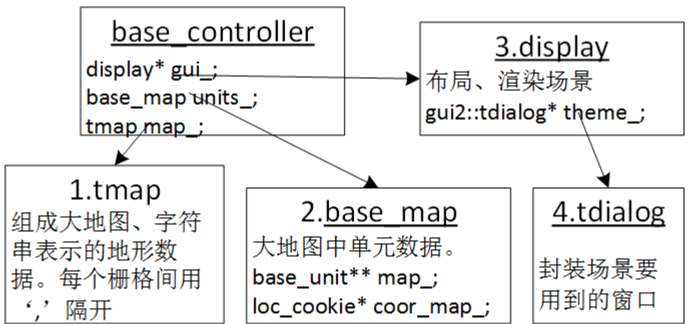
第四步：其它模块使用该场景

参考“**6.1.4 顶层逻辑**”。

6.1.2 大地图如何融入窗口

编写窗口脚本时，放置一个 id 是“_main_map”的 spacer 控件。进入场景后，控制器会在“_main_map”占据的矩形区放置、处理大地图，从而实现把大地图融入窗口。这个 spacer 控件称为地图控件。地图控件中正显示的大地图部分称为视区，如果控件尺寸小于地图尺寸，视区矩形将等于控件矩形，否则是后者从两侧向内偏移出的一个矩形。

6.1.3 组件



对图中带下划线的组件，app 须要从它们自定义派生类。数字序号表示了它们被构造次序。

控制器（base_controller）。负责从 display、base_map 读取输入事件（鼠标、触摸、键盘等），然后调用 app 提供的代码处理它们。同时向 display、base_map 读写数据。控件器必须实现两个方法，一是用于获取 display 的 get_display()，二是获取 base_map 的 get_units()。

窗口（display）。负责除大地图中单元外的窗口管理工作，像布局、渲染控件，包括渲染地形。

大地图（base_map）。负责管理大地图中单元，像如何生成、渲染单元。

地形（tmap）。根据一字符串生成大地图的地形，提供对某一栅格是何样地形特征的判断逻辑。base_controller 管理 tmap，按自个要求使用指针或对象，场景允许运行时更换地形。

tdialog。display 管理场景是基于一个 tdialog，它和对话框中的 tdialog 没啥两样。如果非要说区别，它存在两个特殊控件。一个命名为“_main_map”的 spacer，用于表示大地图。一个叫“_mini_map”的 image，用于表示小地图。

6.1.4 顶层逻辑

```
mkwin_controller mkwin(app_cfg_, video_, ...);
mkwin.initialize(display::ZOOM_72);
int ret = mkwin.main_loop();
```

逻辑分两个阶段，一是构造各组件，二是消息循环。代码中 mkwin_controller 是 app 自定义的从 base_controller 派生的控制器，它和后面的 initialize 都属于构造阶段。为什么要特别调用 initialize？因为 C++不允许在构造函数中调用虚函数。构造阶段可细分为四个步骤。

1）mkwin_controller 的构造函数。结束前需要准备好 map_、units_。

2）(initialize)调用 app_create_display，构造 mkwin_display 对象。

3）(initialize)初始化 mkwin_display，具体分为 1）调用 mkwin_display::app_create_scene_dlg 构造 tdialog。2）mkwin_display::app_post_initialize()执行初始化 mkwin_display 的后续操作。

4）(initialize)调用 app_post_initialize()执行初始化后续操作，执行完它后就进入消息循环。

在(initialize)阶段，app 须要重载 4 个函数，其中两个是创建函数，无论什么 app，这两函数就一条“new”语句。

```
void mkwin_controller::app_create_display(int initial_zoom)
{
    gui_ = new mkwin_display(*this, units_, video_, map_, initial_zoom);
}
void mkwin_controller::app_post_initialize()
{
    在 mkwin_controller::initialize执行，放置构造了 display后的想执行操作。执行完它后就进入消息循环。
}

gui2::tdialog* mkwin_display::app_create_scene_dlg()
{
    return new gui2::tmkwin_scene(*this, controller_);
}
void mkwin_display::app_post_initialize()
```

```
{
    在 display::initialize 执行，放置构造了 tdialog 后的想执行操作。
}
```

main_loop 是消息循环，程序进入它后将被阻塞，直到它退出。更多 main_loop 细节参考“**3.2.6 消息循环：场景**”。

6.2 大地图

6.2.1 Z 轴

在 Z 轴，大地图被分为四层，从底到上依次是窗口层 I、地图层、单元层、窗口层 II。

窗口层 I、窗口层 II

大地图不是窗口控件，因而这两层其实不是大地图元素，但大地图落在窗口中，在最后看到的图像上，窗口行为难免要影响到它。

窗口层 I 就是地图控件，影响大地图在窗口中位置。一旦大地图小于控件尺寸，两侧就会露出窗口部分，比如彩图一的水平方向。当然，地图机制是有提供了设置，当出现此种情况时如何去填塞，假如设置是全透明填塞，那么两侧就会看到窗口背景。

窗口层 II 是指现在悬浮在地图上的窗口控件。彩图三中左上角头像的底下部分就落在了大地图上，还有三个战法条按钮，以及右侧“强制移动”按钮。

6.2.2 地图层

地图层负责画大地图中的地形。像《王国战争》之类模拟游戏，地图层就是要尽量画出自然地貌。对于彩图一的编辑器来说，地图层则可看作是画板，程序基于这画板画单元。

许多等尺寸、不会重叠、中间不留缝隙的栅格拼凑出地图。等尺寸有两个含义，一是它们尺寸一致，二是形状一致。在形状上栅格可以是正方形（彩图一、彩图二），正六边形（彩图三），除去这两种当前已支持的形状，也可自写代码让支持像菱形，只要让满足栅格占据的矩形是正方形。

针对每一栅格，地图层逻辑会在它之上画地形。对模拟现实的大地图，地形就是指草原（Gg）、沙地、丘陵、山地、深水等等，对画板功能的大地图来说，它可能就是一可区别颜色的正方形块，像“Gg”是白色，“Gll”是红色。这里“Gg”、“Gll”是地形字符串标识，每一种地形都会赋上唯一标识，至于“Gg”具体要挂接什么地形，虽然是有约定成熟定义，像“Gg”表示绿色草地，但 app 可按需定义。为让功能更丰富，每一栅格上可画两个地形：基本层和覆盖层，在字符串标识上，它们之间用“^”相连，像“Gg^Fet”表示基本层是绿色草原（Gg） 覆盖层是一棵树（Fet）。对模拟现实的地貌地图，不少栅格须要两层，但对画板这类简单应用，用一基本层去画一栅格就够了。

地形支持动画。像水地形，为逼直要一直显示粼粼波光；沿河岸的水，要让呈现涌岸效果。

整地图被分为两部分，有效区域和边界。有效区域是大地图主体，单元只能站在有效区域上。边界位在有效区域周围，存在边界是为美观，并不是所有地图都须要边界。像采用正方形栅格的彩图一、彩图二，它们就没使用边界，使用正六边的彩图三则存在边界，拼凑正六边形无法做到对齐奇、偶行/列，为美观须要在边沿填补上地形，这些填补部分就被归入边界。因为只是填补用，边界最大只能是一半格子，为什么最大一半就够了，让看下彩图三第二列，相比于第一列它的有效区域下凹了，这下凹部分不会超过一半高度，用于填塞它的边界也就不可能超过一半。

为定位栅格，地图有一个坐标系，坐标系原点是有效区域左上角栅格，从左到右 x 递增，从上到下 y 递增。这意味着左侧边界的 x 是负值，上侧边界的 y 是负值，由于一侧的边界不可能多于一个格子，不论 x 还是 y 它们负值只可能是-1。

在 Z 轴，地图层是位在单元层下，但最后画出来时有些地形图像可能会覆盖在单元上。假设一个地形是“云覆盖的树”的栅格，一部队站在该格子，地表和树是画在部队下，但云的部

分覆盖部队，即单元上。在代码实现时，display 负责画大地图，它在渲染栅格时把栅格中要画的元素分到好多层（这个层和此中的层有相似，但不是一个概念），层号低的先画，一般来说地形元素分配去的层号要比单元低，于是先画，但地形也可分配到比单元大的层号。

构造规则（Building Rule）执行着如何把若干地形拼凑出一幅逼真大地图。要如何编写构造规则，像彩图一那种只是把地图当作画板、栅格间不存在关联的应用是较简单，但换做要模拟现实地貌的地图，要考虑的问题就多多了。它们要处理如何由海拔高的丘陵过渡到平地，如何画出逼真河岸，如何让地形播放出各样动画，等等，这当中不是复杂一点，可说是非常复杂，第十章会以较复杂的正六边形为例子详细介绍构造规则。

6.2.3 单元层

单元层用于放置单元。在编程上，app 定义单元时从 base_unit 派生，管理对象则从 base_map 派生。彩图 5 显示了实现单元层采用的数据结构。

每个单元会分配到唯一坐标（base_unit::loc_），单元坐标来自栅格坐标系，但禁示单元站边界上，因而单元坐标没有负值。要注意的是，一个单元不总是对应一个栅格。彩图 2 是不对应例子，一个控件是一个单元，很显然，一个控件和一个栅格没对应关系。

由于单元和栅格可能不对应，app 在编程时就须要定义它们对应关系。一种最常见也是最简单的对应关系就是两值相等。彩图 1、彩图 3 使用的就是让两坐标系等值。

每一栅格上可存在两个单元：基本层、覆盖层，程序中对应的是 loc_cookie 结构。

```
struct loc_cookie {
	base_unit* base;
	base_unit* overlay;
};
```

app 可能只要一个就够了，是一个时要使用覆盖层（overlay）。对一些复杂应用，像彩图 3，一部队站在城墙上，这时城墙单元是基本层、部队单元是覆盖层。在使用两层时，基本层单元的图像往往来自地形图像，渲染它们时使用地图构造规则。

base_map 负责管理众多单元。为方便管理，使用两个数据结构：map_、coor_map_。

base_unit** map_。它是一维指针数组，每个指针指向一个单元，map_vsize_指示数组中的单元个数，单元间不存在空洞，即没有指向 nullptr 的指针。当一栅格上同时存在基本层、覆盖层单元时，它要在 map_占据两个位置。利用此结构，可很快计算出当前有多少个单元，以及对单元进行排序。

loc_cookie* coor_map_。它是“二维”指针数组，每个指针指向一栅格，栅格间会存在空洞，这里空洞指是的它的两个字段 base、overlay 都是 nullptr。利用此结构，可快速按坐标得到单元，计算出一区域内的单元。

当单元发生变化时，像增加、删除，移动，须要确保让这两结构中数据始终保持一致。

6.2.4 三种坐标系

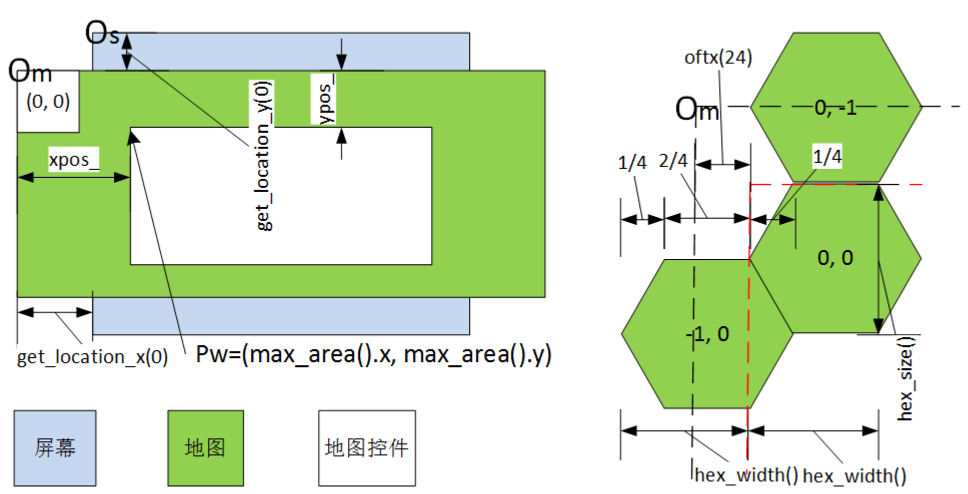


图 6-2 屏幕、大地图控件、大地图位置关系

编程大地图涉及到三种坐标系：屏幕坐标系、地图坐标系和栅格坐标系。注：场景肯定会覆盖整个屏幕，因而在值上屏幕坐标等同窗口坐标。

	屏幕坐标系	地图坐标系	栅格坐标系
描述	图 6-2 中，屏幕原点 Os 是该坐标系原点	图 6-2 中，地图左上角像素 Om 是该坐标系原点	栅格(0, 0)是该坐标系原点
单位	像素	像素	栅格
负值	会出现负值。图 6-2 中，大地图向左移后，Os 左侧像素就出现负值。同样，向上移低于 Os，y 也将负值	不会出现负值。x>=0 && y>=0	左、上边界的栅格坐标是负值，由于一侧边界最多一个栅格，负值只可能是-1。
使用	优点：直接和屏幕融合，像可直接使用鼠标单击点位置	优点：和视区在大地图哪位置无关，即不受滚动大地图影响	

地图坐标系原点是地图左上角像素，不是大地图(0, 0)栅格的左上角像素！地图没有边界时，两个像素是相同位置。存在边界时，像 6-2 右侧正六边形栅格，Om 会在(0,0)栅格的左上角的左上角，它们偏移是多少？这涉及到一个和边界相关的参数：border_size。没有边界时，border_size 是 0，一旦存在边界，值默认是 0.5（app 可修改）。偏移按以下公式计算。

```
oftx = bordr_size * hex_width();
ofty = bordr_size * zoom();

int hex_display::hex_width() { (return zoom_ * 3) / 4; }
int hex_display::zoom() { return zoom_; }
```

zoom_是当前正使用的栅格尺寸，假设是 64。hex_width()返回的是 x 方向上、下一个栅格要开始画的偏移，计算出的值是 48，当乘上 0.5 后，是图 6-2 标示的 24。Y 方向偏移是 32。

接下讨论如何在三个坐标间转换，首先让看 display 提供的三个计算矩形的函数。

函数	描述	示例：屏幕 800x600
main_map_rect	矩形左上角坐标总是(0, 0)。w、h是地图宽度、高度	(0, 0, 2448, 2484)
main_map_widget_rect	地图控件矩形	(0, 26, 800, 471)
main_map_view_rect	视区矩形。如果地图比main_map_widget_rect大，就是后者，否则是后者从两侧向内偏移出的一个矩形	(0, 26, 800, 471)

main_map_widget_rect、main_map_view_rect 返回的是基于屏幕坐标系的矩形。在图 6-2，display 成员变量 xpos_表示大地图向左滚动了的像素数，ypos_表示向上滚动了的像素数，以下

是计算公式。

```
xpos_ = Pw.x - Om.x;
ypos_ = Pw.y - Om.y;
```

由于大地图只可能向左、向上移，因此 xpos_、ypos_不可能是负值。当大地图尺寸小于地图控件时，xpos_、ypos_固定是 0。

```
// loc coordinate to screen coordinate
int loc_2_screen_x(const map_location& loc) const;
int loc_2_screen_y(const map_location& loc) const;

// map coordinate to loc coordinate
const map_location map_2_loc(int mapx, int mapy) const;

// screen coordinate to loc coordinate
const map_location screen_2_loc(int screenx, int screeny) const;

// screen coordinate to map coordinate
void screen_2_map(int& x, int& y) const;
int screen_2_map_x(int screenx) const;
int screen_2_map_y(int screen) const;

// map coordinate to screen coordinate
void map_2_screen(int& x, int& y) const;
int map_2_screen_x(int mapx) const;
int map_2_screen_y(int mapy) const;
```

以上是 display 提供的坐标转换 API。loc_2_screen_x 用于把栅格坐标中的 x 分量转换为屏幕坐标，loc_2_screen_y 则是转换 y 分量。回看图 6-2，当转换的栅格坐标是(0, 0)时，如果把转换出的 x 加上左侧边界宽度、y 加上上侧边界高度，就可得到 Om。基于这个，loc_2_screen(x,y)加上边界尺寸的方法就可实现屏幕坐标和地图坐标间转换。不过 screen_2_map、map_2_screen 没用它，而是使用视区左上角距离 Os 的偏移值和 xpos_、ypos_实现转换。

6.2.5 一些 API

一、插入

```
void base_map::insert(const map_location loc, base_unit* u)
```

loc，单元坐标系中坐标。指示单元要放在位置。u 是要放置的单元，base_unit 不会克隆一个 u，app 在调用这函数后不要去释放它，若没特殊须要，放手让 base_map 在析构时自动释放 u 占用的内存资源。

如果插入时希望 base_unit 克隆出一个，然后把克隆的放入 base_unit，可调用 base_unit::add。

二、删除

```
bool base_map::erase(const map_location& loc, bool overlay = true)
{
    if (!loc.valid()) {
        return false;
    }
    base_unit*    u    =    overlay?    coor_map_[index(loc.x,    loc.y)].overlay: 
coor_map_[index(loc.x, loc.y)].base;
    if (!u) {
        return false;
    }
    return erase2(u);
}

bool erase2(base_unit* u, bool delete_unit = true)
```

它们都可用于删除单元，app 按自个方便选择调用哪一个。delete_unit 指示在删除时是否要释放掉单元所占资源，默认是释放，当然，如果希望手动释放 u 所占资源，那只能调用 erase2。

三、移动（只能操作 Overlay 层单元）

```
void base_map::move(const map_location src, const map_location& dst)
{
    base_unit* u = extract(src);
    place(dst, u);
}
```

move 实现把位在 src 上的单元移动到 dst。若不须要排序，移动就是操作 coor_map_，不会去弄脏 map_。extract、place 也是 app 可直接调用两个 API，extract 执行从 src 提取出单元，即把单元占据的栅格上指针置 NULL，place 则把单元放到 dst 位置。

四、枚举

依次格举出所有单元。base_unit 实现了个类似 STL 的单元迭代器，让应用可按 STL 风格进行枚举。

```
for (unit_map::iterator it = units_.begin(); it != units_.end(); ++ it) {
    unit* u = dynamic_cast<unit*>(&*it);
    ......
}
```

it 向外释放的是 base_unit 类型，unit 是 app 自定义的从 base_unit 派生的单元对象。如果枚举是发生在从 base_map 派生的对象，那有着更简单、快速办法。

```
for (size_t i = 0; i < map_vsize_; i ++) {
    unit* u = dynamic_cast<unit*>(map_[i]);
    ......
}
```

五、查找

根据不同输入参数，base_map 提供了四个版本的查找函数，函数名都是 find_base_unit。

```
base_unit* find_base_unit(const map_location& loc) const;
base_unit* find_base_unit(const map_location& loc, bool overlay) const;
base_unit* find_base_unit(const int at) const;
base_unit* find_base_unit(const int xscreen, cons int yscreen) const;
```

第一个、第二个都是根据栅格查找，第一个同时查找覆盖层、基本层，第二个只查找指定层。第三个根据索引在 map_集合中找，速度最快。第四个根据屏幕坐标，当该坐标存在多个单元时，像彩图 5 处理冲突中的(2,2)，返回的是排序在最尾的那个单元。

基于 base_map 提供的 find_base_unit，app 可封装出自个的 find_unit 函数。

6.2.6 排序单元

从上面的枚举可看出，单元在平面中顺序其实就是由 map_决定的，map_是个一维数组，单元在它之中位置越靠前，就意味着这单元越优先。对于这个顺序，要是不作任何处理，那么谁先插入（insert）谁就放前头。基于功能需要，一些 app 希望能够排序单元，像彩图 1 中窗口编辑器，在生成窗口配置时它要按“Z”序依次读出控件，那就希望 map_要按此个“Z”、即单元坐标进行排序。彩图 3 也需要排序，排序依据是各部队时间槽，时间槽最满的排在最前。base_map 提供了个排序框架，依着这框架，app 要让从 base_umit 派生的单元对象重载两函数。

函数一：require_sort

```
bool base_unit::require_sort() const
{
    return false;
}
```

base_unit 默认不使能排序，采用谁先插入（insert）谁就放前头。如果须要排序单元，app 要让该函数返回 true。要不要排序的判断被放在单元级，这是因为一些应用并不是须要排序所有单元，像彩图三，建筑物单元就不须要排序。一旦把单元分为排序和不须要排序两大类，那 app 应该处理好这两类之间位置关系，像把不要排序全放在后头。

函数二：sort_compare

```
bool base_unit::sort_compare(const base_unit& that) const
{
```

```
    return loc_.y < that.loc_.y || (loc_.y == that.loc_.y && loc_.x < that.loc_.x);
}
```

一旦使能排序，base_map 默认实现的是按坐标排，即坐标小的排前。app 不想按这规则时，那就须要重载这函数。函数中 that 是 map_中已存在单元，this 是正被排序单元（新单元），如果新单元要放在 that 的前头，返回 true。

要注意排序的发生时机，排序会在两种场合被触发，一是 insert 后，二是 place 后。base_map 没提供一次对整个 map_进行排序的 API。对要排序的单元，app 要确保一开始就设上 require_sort 标志。

6.2.7 栅格坐标不等于单元坐标

彩图 2 是栅格坐标不等于单元坐标例子。要支持此种场合，单元要使用一个指示它在大地图中位置的矩形，这个矩形属于地图坐标系。

```
SDL_Rect rect_;
```

在栅格坐标等于单元坐标时，rect_总是{0, 0, 0, 0}。

应用首先设定这个 rect_，然后程序根据它计算出单元坐标。

计算单元坐标（conflict_calculate_loc）

conflict_calculate_loc 被用于从 rect_计算出单元坐标。逻辑可概括为：根据 rect_的左上角计算出栅格坐标，如果该坐标没有单元，那它就是此单元的单元坐标，否则要进行冲突处理。

彩图 5 图形化表示了如何处理冲突。依次要插入 A、B、C 三个单元，根据它们矩形的左上角计算出的地图格子都在(2, 2)。当插入 A 时，栅格上没单元，(2, 2)就直接作为 A 的单元坐标。插入 B 时，检测到栅格上有单元，这时要查找同一行上的一个空闲位置，空闲位置特点是 1）同一 y 坐标，2）x 坐标一定超过初始地图宽度。由于 y=2 上没有空闲位置，这时要扩大地图尺寸（expand_coor_map）。扩大后(4, 2)出现空闲，(4, 2)就做为 B 单元坐标。依此规则，(5, 2)则做为 C 的单元坐标。

编程

```
void base_unit::set_rect(const SDL_Rect& rect);

void base_map::insert2(const display& disp, base_unit* u)
{
    VALIDATE(!consistent_ && !u->consistent(), null_str);
    map_location loc = conflict_calculate_loc(*u);
    insert(loc, u);
}
```

不同于栅格坐标等于单元坐标，app 要多调用这两个函数。set_rect 用于设置单元在大地图中矩形，insert2 则用于代替 insert，它把单元插入 base_map。

base_map::consistent_指示当前栅格坐标是否等于单元坐标，u->consistent()则指示 u 是否已设置过矩形，VALIDATE 要确保 u 已设置过矩形。conflict_calculate_loc 根据矩形计算出单元坐标，计算出单元坐标后就调用 insert 把单元插入 base_map。

6.2.8 tmap

tmap 用于封装地图。构建地图可以是两种数据，一是地形码字符串，二是图面。

```
tmap(const std::string& data);
```

以上是用地形码字符串构造 tmap。字符串是以着“\r\n”分隔行、逗号分隔列的格式。格式的实例参考“**10.1 通过一个例子直观理解地形系统**”。

```
tmap(const surface& surf, int tile_size);
```

以上是用图面构造 tmap。tile_size 是此次地图的配置尺寸，surf 的宽度、高度必须是 tile_size 整数倍。

6.3 光环（halo）

彩图 3 中咒术师发射的火球、道士发出的光束都是光环，这些光环是动画一部分。注：图中出现的“减速”和数字没用光环，而是用 floating_label。彩图 9 中两条横线、中间竖线、竖线旁边的“03:33:58”都是光环。

光环是大地图一部分，用于增强大地图中的图形内容。它增强的图形包括来自文件的图像、字符串、自画的图形。

光环中单元称为特效(effect)，光环子系统的任务就是要在大地图画这些特效。

6.3.1 app 如何使用光环

使用光环分两种，一种是在动画中使用光环，二是在大地图覆盖上图形。对于第一种在动画中定义。以下讨论第二种。

操作：增加特效

在 display 的派生类重载 add_halos。在该函数调用 halo::add 增加特效。

```
int music_cursor_halo_;
void chart_display::add_haloes()
{
    ......
    if (music_cursor_halo_ != halo::NO_HALO) {
        halo::remove(music_cursor_halo_);
    }
    mapped_col = controller_.cursor_draging()? 0xffff0000: 0xffffffff;
    blit = image::tblit(mapped_col, create_point(0, 0), create_point(0, area.h -
1));
    music_cursor_halo_ = halo::add(cursor_pos_, ypos_, false, blit);
}
```

以上代码可画出类似图 2 中的竖线。cursor_pos_是地图坐标，于是在调用 halo::add 时把第三个参数置为 false，告知输入的 x、y 是地图坐标。由于要实现的竖线是可拖动的指针，该指针要随着拖动位置变化而变化，而要画出新位置就须要移除先前位置。

操作：删除特效

在 add_halos 调用 halo::remove 删除特效。

6.3.2 特效中内容是动画

effect 存放内容的变量是 images_，该变量类型是 animated<image::tblit>，它被定义成动画。

但实际使用时，该动画往往只有一帧，即使是大量使用动画的《王国战争》都没让 images_是多帧动画。在这里，有人直观会认为使用带有逐进变量光环的单元动画会产生多帧的 images_，其实不是这样的，即使是带有逐进变量光环的单元动画，unit_frame::redraw 已经拆分了逐进变量，是根据当前拆分到的图像生成特效。

把 images_设为支持多帧，这么设定是为支持能画出具有动画效果的光环，像让出现带闪烁照明光环的法师。

6.3.3 擦除和渲染

擦除有两个任务，一是增加、删除特效(display::add_haloes)，二是根据当前特效置脏相关栅格(halo::unrender())。add_haloes 是 display 的虚函数，派生类有特效时重载它。halo::unrender() 则由 halo 实现，app 不用管。擦除时机在 invalidate_animations 之后，invalidate_units 之前。

渲染（halo::render）指在大地图画特效。渲染时机在 draw_units 后，drawing_buffer_commit 之前，具体见“**6.4.1 display:;draw**”。

6.3.4 effect 中的 x_、y_、loc_

为了和滚动无关，effect成员变量x_、y_是地图坐标，它来自于halo::add或effect::set_location。依自个方便，app 在这两个函数可传入屏幕坐标或地图坐标。

特效的尺寸和栅格无关。但它涉及到范围需转换成涉及到栅格。

特效可以和特定一个栅格关联。一旦关联后的特点：1）关联的栅格处于黑幕时，特效不会被显示；2）栅格只能在构造时挂接一次，中间不能再改。举个例子，白袍法师有照明光环，就可让照明光环和法师所在栅格关联，当该栅格被黑幕时，同时不显示光环。光环只能挂接一次，但法师会移动，那么移时怎么确定光环位置是对的？每一次 unit::set_location 时会调用 unit::draw_unit，后者会销毁旧的光环构建新光环。

6.4 渲染

彩图 8 图形化了 display::draw。

6.4.1 display::draw

display::draw 执行渲染一次场景，被时间片函数 play_slice 调用（参考“3.2.6 消息循环：场景”）。它不是虚函数，能适用各种 app 是通过调用一些是虚函数的操作，彩图 8 中蓝色表示的操作是虚函数。

大地图可能非常大，当前在屏幕显示的部分称为视区。为提高效率，自然要做到尽量只画视区。于是 display::draw 首先计算出视区涉及到栅格，draw_area_rect_表示了这个集合。draw_init 处理和大地图背景相关事项，包括 1）需要的话重画背景，2）判断是否要把视区中所有栅格置脏，正是这可能导致整视区置脏的操作，使得它要较早被执行。pre_draw 是根据 draw_area_rect_ 计算视区涉及到单元，结果放在 draw_area_unit_。到此已得到视区中栅格和单元，接下就要开始渲染具体内容。

渲染内容分为窗口和大地图。大地图又细分为三种：地形、单元和光环。对它们要解决四个问题，一是哪些是脏的，二是要画什么，三是画的次序，四是如何画。

哪些是脏的。1）地形。从 invalidate_animations 一直到 invalidate_units 都在收集脏栅格。存在多种原因栅格要置脏。invalidate_animations：地形自身正在播放动画，或上面正经过单元正在播放的动画。halo::unrender：上面有光环要经过。invalidate_units：上面有脏单元。invalidate_float_wdigets：上面有悬浮控件。2）单元。当前采用的是位在视区的所有单元都会置脏，即 draw_area_unit_就是彩图 8 中的脏单元集合 unit_invals。这么设计是默认所有单元都存在站立（待在原地）动画。3）光环。所有光环都会置脏。

画什么。地形、单元、光环，每一种都存在好多图像，display 提供了“drawing_buffer_add”，代码调用它把要渲染内容细化成一个个 tblit（tblit 参考“2.2.7　渲染 API”），然后汇总到“drawing_buffer_”这个 tblit2 缓存。根据不同内容，存在多个版本的“drawing_buffer_add”，但不论哪种，除了要提供构造 tblit 必须的字段，还要有 layer、loc，后两个字段决定了该 tblit 的绘画次序。

```
void drawing_buffer_add(const tdrawing_layer layer, const map_location& loc, int
x, int y, const surface& surf, const int width, const int height, const SDL_Rect
&clip = SDL_Rect());

typedef std::list<tblit2> tdrawing_buffer;
tdrawing_buffer drawing_buffer_;
```

画的次序。到最终屏幕，并不是一定先画地形、后画单元、最后光环，而是相互交叉，即一个地形图像在单元图像下，另一个地形可能又在单元上。在说“drawing_buffer_add”时，提到调用它必须同时提供 layer 和 loc，layer 这个整型变量指示了此 tblit 在 Z 方向上位置。

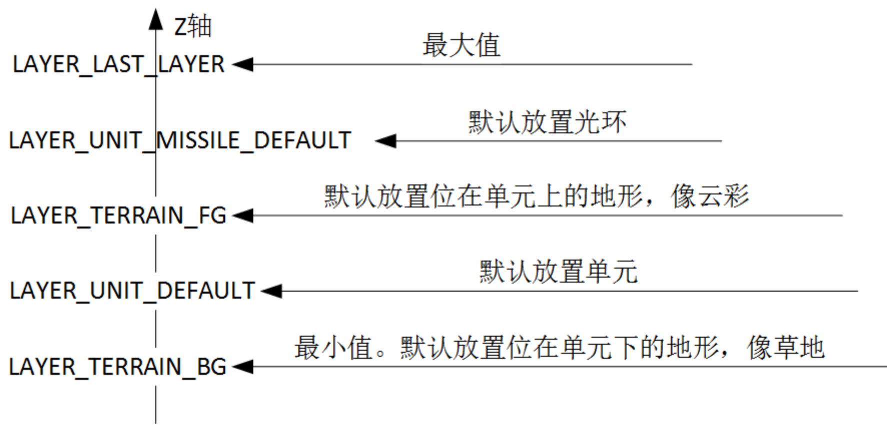


图 6-3 tdrawing_layer 和 Z 轴

图 6-3 显示了几种“enum tdrawing_layer”定义的值，app 也可使用不是 tdrawing_layer 中的值，但要确保属于[LAYER_TERRAIN_BG, LAYER_LAST_LAYER]区间。如果两个 tblit 有相同 layer，就根据第二个字段“loc”（一般就是单元坐标）进一步判断放置次序。

如何画。回看彩图 8，drawing_buffer_commit 负责把 tblit2 缓存中内容渲染到屏幕纹理。至此完成了在场景渲染大地图。

scene_->get_window()->draw()执行画窗口，draw_sidebar 更新窗口上的状态报告。draw_wrap 主要执行两个任务，一是画小地图，二是切换双缓冲机制的前、后台，场景希望的内容就被显示到屏幕。

6.4.2 效率

为支持所有单元同时播放动画，每次会强制刷新视区内所有单元。刷新频率会较高，这时需要考虑如何提高刷新效率，具体表现在 app_redraw_unit 中的代码。

尽量不要使用大图面。为减少 drawing_buffer_add 次数，app 可能会这么干。首先调用 create_neutral_surface 创建个覆盖整个单元的图面，然后把要画的元素画向这 surface，最后用一次 drawing_buffer_add 把这 surface 加到渲染缓存。这种方法会产生个副作用，当单元尺寸很大时，像(2000x1800)，光 create_neutral_surface 就可能消耗大量 cpu。这时建议的方法是针对每个元素调用一次 drawing_buffer_add。

6.5 缩放大地图

通过修改栅格的显示尺寸实现缩放大地图。不管栅格是正方形、正六边形，还是菱形，栅格占据的肯定是正方形，成员变量 zoom_表示了这个边长。zoom_称为显示尺寸，和它对应，存在个叫配置尺寸的 image::tile_size。配置尺寸用在制作素材时，所有地形按这尺寸去绘画。每个 app 依着自个需要决定这个尺寸，像彩图 3 用的是 72，彩图 1 用的是 128。一旦 zoom_大于 tile_size，就表示要放大大地图，小于 tile_size 则是缩小。

6.5.1 初始化阶段

初始化阶段具体是指 display 构造函数，这个阶段要设置好和 zoom_相关的三个变量。

变量	位置	
initial_zoom	传给 display 构造函数的参数	它将赋给 zoom_，作为初始的显示尺寸
min_zoom_	display 成员变量	可设置到的最小显示尺寸，运行过程中设到的尺寸不会小于这个值
max_zoom_	display 成员变量	可设置到的最大显示尺寸，运行过程中设到的尺寸不会大于这个值

app 在初始化 controller 时传入 initial_zoom。min_zoom_、max_zoom_则是在构造 display 的

派生类时被赋值。

6.5.2 运行时阶段

一旦要缩放，将会调用 display::set_zoom，参数 amount 指示了希望的尺寸。

```
void display::set_zoom(int amount)
{
    int new_zoom = zoom_ + amount;
    ……
    if (new_zoom != zoom_) {
            ...
            const int old_zoom = zoom_;
            zoom_ = new_zoom;

            image::set_zoom(zoom_);
            post_set_zoom(old_zoom);
    }
}
```

set_zoom 要对 new_zoom 进行[min_zoom_, max_zoom_]的越界调整，调整后才和 zoom_进行比较。app 要做的是重载 display::post_set_zoom。

6.6 小地图（minimap）

彩图 1、彩图 2、彩图 3 都有小地图。场景中地图大于一屏尺寸，为直观就可用小地图去鸟瞰整地图，小地图中矩形框指示当前视区。

```
[image]
    id=_mini_map
    definition = "blits"
[/image]
```

为让出现小地图，在窗口脚本放置个 blits 风格 image 控件，id 必须_mini_map。一旦场景中出现“_mini_map”标识的控件，Rose 认为此场景存在小地图。

display::draw_minimap()执行绘画小地图。小地图中内容由四部分组成，由底向上分别是背景矩形、地图层、单元层和视区矩形。display::draw 调用 draw_minimap，但如果每次都调用太浪费 cpu，一个叫 redrawMinimap_的布尔变量控制是否要重画小地图。app 调用 redraw_minimap()把这变量置 true。每次 draw_minimap 一定会重画背景矩形、单元层和视区矩形，但不一定重画地图层，原因见“**6.6.2 地图层**”。

6.6.1 背景矩形

背景矩形是个实心矩形，填充的颜色固定是 argb={255, 31, 31, 23}，它会填满整个小地图。

6.6.2 地图层

minimap_surface 负责画地图层，默认行为是生成大地图的一个缩小版 surface，app 可重载它画出自制地图。考虑到地图地形可能很复杂，画它较耗 cpu，而且地图数据往往不大会变，于是缓存 minimap_surface 结果到一个叫 minimap_的 surface。这就造成 draw_minimap 渲染地图层用的可能不是那一时刻的地图数据，要想强制使用当时数据需调用 recalculate_minimap()。

```
void recalculate_minimap() {
    minimap_ = NULL;
    redrawMinimap_ = true;
}
```

它就是个加了“minimap_ = NULL”的增强版 redraw_minimap。

6.6.3 单元层

```
void display::app_draw_minimap_units(surface& screen);
```

app_draw_minimap_units 绘画单元层。它是虚函数，display 派生对象需重载它以实现绘画

自个内容。app 编写它时极可能要用到表示小地图在场景中位置的 minimap_location_，这是个一定在小地图控件矩形内的 SDL_Rect。为确保宽高比不变，地图可能无法覆盖整个小地图控件，于是占据中间部分，这时或上、下，或左、右，会露出背景。minimap_location_指的是当中有效部分，它属于小地图控件坐标系，即坐标原点是小地图控件的左上角坐标。

```
double xscaling = 1.0 * minimap_location_.w / map_.w();
double yscaling = 1.0 * minimap_location_.h / map_.h();

double u_x = u->get_location().x * xscaling;
double u_y = u->get_location().y * yscaling;
double u_w = xscaling;
double u_h = yscaling;
```

map_.w()指示大地图水平栅格数，根据“小地图宽度（像素数）”除以“水平格子数”计算出的“xscaling”就是一格子在小地图中宽度（像素数）。以此类推，yscaling 表示小地图中每一栅格高度（像素数）。

u_x、u_y、u_w、u_h 计算出某一单元要对应小地图的矩形区域。

6.6.4 视区矩形

视区矩形指示视区在大地图中位置，它是一个框。draw_rectangle 绘画视区矩形。窗口脚本中的 view_rectangle_color 定义这个颜色。

6.7 状态报告

状态报告不是单一控件，它们用于实时报告程序当前状态，像图 6-1 中顶行所有项、右侧的部队名、HP、XP，等等。状态报告除了要能实时反映程序状态外，它一大特点是不支持用户输入事件。针对它，Rose 提供了统一处理框架，应用遵循以下三个步骤就可快速给自个场景增加状态报告。

6.7.1 重载 pre_show

在 gui2::tdialog 派生类重载 pre_show，在 reports_增加场景可能出现的状态报告

```
void tgame_scene::pre_show()
{
    ......
    reports_.insert(std::make_pair(UNIT_NAME, "unit_name"));
    reports_.insert(std::make_pair(UNIT_TYPE, "unit_type"));
    reports_.insert(std::make_pair(TURN, "turn"));
    ......
}
```

变量 reports_的类型是 std::map<int, const std::string>，first 中的值推荐用 enum 定义，但要注意，它们基于 0 始、且须要连续，second 是该报告对应的控件标识。

6.7.2 调用 refresh_report 刷新报告

```
void display::refresh_report(int num, const reports::report& r)
```

参数 num 就是 reports_变量中使用的 first，它指出要刷新哪个报告，第二个参数 report 存储报告内容。Rose 把报告分为两类，字符型和图面型。当显示的是字符或一次 tintegrate 就可生成的图像时，建议用字符型，其它则采用图面型，即通过混叠 surface 出来想看到的结果。

```
report(ttype type, const std::string& text, const std::string& tooltip)
```

它是报表构造函数，参数 type 指示要构造的报告类型，report::LABEL 指示构造的是字符型，report::SURFACE 指示图面型。

为什么构造图面型报告时要使用字符串，这是考虑到应用很难做到执行生成报告函数的时刻是必须时刻。以要显示回合这状态报告为例子，假设界面已显示“第 4 回合”，可应用这时还会构造 report 再次生成“第 4 回合”报告，很显然，这个构造生成的效果和界面正显示一样，白白浪费系统资源。Rose 针对这情况进行了处理，它没法阻止应用去生成 report，但要避免掉

花时间去第二次混叠图面，于是把生成图面型报告分为两个阶段，第一阶段是构造 report，传给它一个能构造图面的字符串；第二阶段则根据此字符串生成图面。应用控制着第一阶段发生时机，Rose 则控制着第二阶段发生时机，是它认为确实需要时刻，即此次提供的 text 和界面使用的 text 不一致。一旦 Rose 认为确实到了该生成图面型报告时刻，它会调用 refresh_surface_report，这些图面自然是应用才知道怎么生成，应用接下要做的就是重载此函数生成图面。

6.7.3 重载 refresh_suface_report

针对此场景从 display 派生的类（game_display）重载 refresh_suface_report 函数。

```
surface game_display::refresh_surface_report(int num, const reports::report& r,
gui2::twidget& widget)
{
    surface surf;
    const SDL_Rect rect = widget.get_rect();
    if (num == gui2::tgame_theme::TURN) {
        surf = generate_turn_surface(r.text, rect.w, rect.h);
    } else if (...) {
        ...
    }
    return surf;
}
```

num 指示要刷新的报告，widget 指示该报告所在控件，返回的图面尺寸需是该控件尺寸。

6.8 按钮命令

6.8.1 三类按钮

场景中按钮被分为三类：普通按钮、上下文菜单中按钮、除这两类之外的按钮。

普通按钮。它们往往“单独”存在，除去位置，不和其它按钮存在关联。像彩图一中左下角“聊天”，彩图二中“控件面板”顶端的三按钮。

上下文菜单中按钮。按着一定的逻辑关系，若干按钮被归入一个上下文菜单。系统提供了专门方法去实现上下文菜单，当中包括了如何处理内中按钮命令，更多细节参考“**6.5.2 上下文菜单**”。

一般按钮已经可归入以上两类，但不排除专门目的的特殊按钮。对于那些按钮，应用须要自个去专门处理。

6.8.2 处理阶段

处理按钮命令的过程包括两个阶段，一是初始化，二是运行时。

阶段一：初阶化

此阶段任务是定义命令，并把按钮和命令挂钩。

阶段二： 运行时

运行时指的是用户用或鼠标、或触屏、或快捷键按下按钮，导致希望执行该按钮挂接的命令。这里主要是两个任务，一是判断当前状态是否可以执行此命令，二是执行具体的命令操作。

判断当前状态是否可以执行此命令。如果界面做到了把当前不能执行命令让不能执行，这步其实是不须要的。界面让不能执行一般是两种方法，一是灰掉，二是隐藏，像不能聊天时，把聊天按钮灰掉，不能执行移动时，把移动按钮隐藏掉。理论上如此，实际却往往存在漏掉情况，这时程序就该进行有效性判断。当前不能执行的命令却被执行，是潜在不安全因素，那么此中判断可说给系统稳定性加了一层保险。对于这判断，普通按钮、上下文菜单中按钮是两种不同实现。

执行具体的命令操作。通过多层判断，认定了当前已可执行此命令。对于这个过程，普通按钮、上下文菜单中按钮以着同一方法被处理。

6.8.3 实现步骤

对如何处理按钮命令，Rose 提供了一个框架，以下叙述应用如何借用这框架处理按钮命令。此框架可能不适用于第三类按钮，但在编写过程中可作为参考。

步骤一：重载 pre_show

在 gui2::tdialog 派生类重载 pre_show，它实现初始化阶段任务，即定义 命令、把命令和按钮挂钩。具体代码实现上， hotkey::insert_hotkey 实现了定义此场景涉及到命令，命令和按钮挂钩则是通过调用 ttheme::click_generic_handler。

```
void insert_hotkey(int id, const std::string& command, const t_string& tooltip)
```

id 是数字标识，为方便识别，应用须给每个命令分配一个数字标识。定义出的数值不能小于 ttheme::HOTKEY_MIN，小于 HOTKEY_MIN 的标识被认为是一般场景都须要的命令，它们总是被定义的，像缩放大地图、聊天、复制、粘贴，等等。由于不能小于 HOTKEY_MIN，而且须是数字，应用自定义标识一般有着以下格式。

```
enum {HOTKEY_SELECT = HOTKEY_MIN, HOTKEY_STATUS, ...}
```

command 是命令字符串，它对应存在于场景中的一个按钮 id。

tooltip 是提示。当鼠标悬浮在该按钮上时就会出这提示。

```
void ttheme::click_generic_handler(twidget& widget, const std::string& sparam)
```

click_generic_handler，实现了把命令和按钮控件挂钩。widget 是按钮控件。sparam 是个字符串类型参数，它会被原封不动传去具体执行过程，某个按钮的执行操作须要一个字符串做参数时，那可在此设上一个值。

应用有用到内置命令时，像要用到 HOTKEY_COPY，它不必为此命令调用 insert_hotkey，但须要调用 click_generic_handler，而且按钮 id 必须定义为 HOTKEY_COPY 对应的“copy”。

```
void tmkwin_theme::app_pre_show()
{
    ...
    hotkey::insert_hotkey(HOTKEY_SELECT, "select", _("Select"));
    hotkey::insert_hotkey(HOTKEY_STATUS, "status", _("Status"));
    ...
    tbutton* widget = find_widget<tbutton>(window_, "select", true, true);
    click_generic_handler(*widget, null_str);

    widget = find_widget<tbutton>(window_, "status", true, true);
    click_generic_handler(*widget, null_str);
    ...
}
```

以上定义了两个按钮命令，命令 HOTKEY_SELECT 对应 id 是“select”的按钮，HOTKEY_STATUS 对应的按钮则是“status”。

步骤二：重载 can_execute_command

can_execute_command 是 controller_base 成员函数。要注意的是，只有普通按钮才会进入此函数，上下文菜单中按钮是另一套实现，此函数不必处理那些按钮。

```
bool can_execute_command(int command, const std::string& sparam) const
{
    using namespace gui2;
    switch (command) {
    case tmkwin_theme::HOTKEY_SELECT:
        ...
    }
    ...
}
```

command 是命令的数字标识。sparam 是命令专门的字符串参数。如果当前状态允许执行此命令，返回 true，否则 false。

步骤三：重载 app_execute_command

app_execute_command 是 controller_base 成员函数。和 can_execute_command 不同，它同时

要处理上下文菜单中按钮。

```
void app_execute_command(int command, const std::string& sparam)
{
    using namespace gui2;
    switch (command) {
    case tmkwin_theme::HOTKEY_SELECT:
        ...
    }
    ...
}
```

有人可能会问，这函数名为什么加了个“app_”，controller_base 是有存在个没有“app_”的 execute_command。

```
void controller_base::execute_command(int command, const std::string& sparam)
{
    if (!can_execute_command(command, sparam)) {
        return;
    }
    return app_execute_command(command, sparam);
}
```

execute_command 就是加了根据当前状态判断是否可执行的 app_execute_command。一般应用不须要修改 execute_command 逻辑，它没被定义为虚函数。ttheme::click_generic_handler 在把命令挂接到按钮时，挂接到的正是这个 execute_command。

6.9 事件

6.9.1 base_controller 默认处理

对如何处理事件，“**5.4.2 标准化**”有说过，tevent_handler 把原生事件转为数个函数，窗口、大地图则从 tevent_handler 派生，以着重载函数的方式进一步处理这些事件。具体到大地图，是 base_controller 从 tevent_handler 派生。

handle_pinch。base_controller 通过变化栅格显示尺寸实现缩放。放大时，结果的栅格尺寸=当前栅格式尺寸*3/2，缩小时，结果的栅格尺寸=当前栅格尺寸/2。

handle_swipe。根据(xlevel, ylevel)这两个幅度算出滚动方向，方向可能有 8 个。一旦认为要滚动，把 finger_motion_scroll_置为 true，后续的 handle_scroll（何时被调用参考“play_slice”）发现这变量是 true，就会执行滚动大地图。(xlevel, ylevel)只是决定滚动方向，滚动幅度是 handle_scroll 内定，固定 50*hdpi_scale 像素。

handle_mouse_down。1）单击在的是悬浮控件，处理结束。2）单击在的是小地图，滚动视区以显示单击在的栅格，并把 minimap 置 true。3）单击在的是大地图，调用虚函数 app_left_mouse_down 或 app_right_mouse_down。只有左键有小地图定位功能，右键没有步骤 2。

handle_mouse_up。调用虚函数 app_left_mouse_up 或 app_right_mouse_up。对左键有个额外 bool 参数 click，true 表示从 down 到 up 的过程中没产生过 motion 或多指触摸，于是认为能产生一次 click。

handle_mouse_motion。1）单击在小地图，滚动视区以显示移动到的栅格，并把 minimap 置 true。2）调用虚函数 app_ mouse_motion，该函数返回 true 表示允许执行第 3 步，false 则阻止。3）正按着鼠标左键并且 minimap=false，执行滚动地图。为方便 app 编程，在调用 app_mouse_motion 前会设置和鼠标相关的 mouseover_hex_、mouseover_unit_。

变量	归属	
mouseover_hex_	display	鼠标正落在的栅格
mouseover_unit_	controller	鼠标正落在的单元
mouseover_hex_overlay_	display	鼠标正拖曳着的图像

除了 app_mouse_motion，另两函数 app_mouse_down、app_mouse_down 也可使用这两变量。

app_mouse_leave_main_map 是和鼠标有关又一个函数。它的作用是为支持只限制在大地图

的拖拽操作。app_left_mouse_up 只在鼠标松开时才会被调用，会漏掉一些离开大地图事件，像拖着鼠标离开大地图。

6.9.2 小地图

虽然小地图是标准 GUI 控件，但 base_controller 接管了它的鼠标、触摸事件。

6.10 修改分辨率

iOS、Android 的 app 一直在全屏运行，不需要修改分辨率。需要修改的是 PC 平台，包括 Windows、Mac OS X、Linux。

Rose 沿袭 SDL，把 app 运行状态分为全屏和窗口，在两种状态下都可以修改分辨率。有人可能存在个疑问，全屏时的分辨率难道不就是屏幕当前正在使用的分辨率吗？——不是，可以认为 app 能用的分辨率和屏幕正在使用分辨率是独立的。在全屏时，app 照样可以设到其它分辨率，但不管什么分辨率都会让 app 占据整个屏幕。当然，为界面美观，全屏时建议用最大分辨率，像 1280x800 的笔记本就设到 1280x800，一旦小于它，有可能出现像黑边、像素块。

既然要修改，app 需要提供方法让用户进行修改，如何触发修改分为拉伸触发和命令触发。

6.10.1 拉伸触发、命令触发

拉伸触发指的是用鼠标拉伸窗口边框，拉着边框让改变窗口尺寸，从而改变分辨率。它包括双击标题栏使窗口最大化、恢复通常尺寸。

命发触发指的是在用户界面放置命令入口，像放置“退出/进入全屏”的选择框，弹出“修改分辨率”的按钮。

要拉伸必须要有边框，因而拉伸触发只会发生在窗口状态，它的行为等于在窗口状态时改变分辨率。拉伸触发不会导致全屏和窗口之间的切换，要实现这种切换以及全屏下修改分辨率必须使用命令触发。

在任一时刻，一个应用内可能同时存在三类单元：主窗口、对话框、场景。主窗口是容纳应用的容器，发生要修改分辨率时，对于它基本等同修改下矩形尺寸，但对话框、场景，由于内中复杂，一旦分辨率发生变化，它们要改的东西就多了。

6.10.2 对话框分辨率

拉伸触发。发生拉伸后，应用会收到 SDL 发来的 SDL_WINDOWEVENT_RESIZED，该消息 window.data1、window.data2 字段分别指示拉伸后的窗口宽度、高度，gui2 系统的 handle_event 处理这消息。为处理它，gui2::event::thandler 会向当前所有的窗口广播 SDL_VIDEO_RESIZE 通知，每个窗口于是调用和这通知对应的 twindow::signal_handler_sdl_video_resize，后者一来调用 preferences::set_resolution 执行修改主窗口分辨率，二来设置标志以重布局自己。

拉伸触发会要求重布局整个对话框，但不会改变对话框已选择的控件配置。举个例子，之前选定的控件是 480x320 配置，把窗口拉伸到超过 800x600 时，它不会就变成选择 800x600 配置，还是使用之前的 480x320！很显然，不改变配置会导致对话框不美观，缩小时还有可能使布局失败。为此一定要改为随着新窗口尺寸去决定控件配置，而要实现这个，就必须拆掉重新生成对话框。

命令触发。它没有 SDL_WINDOWEVENT_RESIZED 这一套机制，不像用户可随时拉边框，程序员为实现自个目的可灵活设置命令，目的就包括方便拆掉、重建对话框。为实现这目的，很重要一条是一旦发生改变分辨率，存放命令的对话框需要被拆掉！至于要不要重新生成则视用户需要。

```
namespace preferences;

int show_preferences_dialog(display& disp, bool first)
{
```

```
    while (true) {
        int res = instance->show_preferences_dialog(disp, first);
        if (first) {
            first = false;
        }
        if (res == preferences::CHANGE_RESOLUTION) {
            if (preferences::show_video_mode_dialog(disp)) {
                return res;
            }
        } else if (res == preferences::MAKE_FULLSCREEN) {
            preferences::set_fullscreen(disp, true);
            return res;
        } else if (res == preferences::MAKE_WINDOWED) {
            preferences::set_fullscreen(disp, false);
            return res;
        } else {
            return res;
        }
    }
}
```

instance 是全局 base_instance 对象，它的 show_preferences_dialog 方法实现了如何显示存放修改分辨率的那个对话框。一旦要求"修改分辨率"，该对话框返回"CHANGE_RESOLUTION"，如果进入全屏是"MAKE_FULLSCREEN"，退出全屏则是"MAKE_WINDOWED"。它也可以有其它返回值，像按下"关闭"按钮返回"twindow::OK"。不论什么返回值，代码在执行"修改分辨率"时已退出 instance->show_preferences_dialog，即存放命令的对话框已被拆掉了。

```
int base_instance::show_preferences_dialog(display& disp, bool first)
{
#if (defined(__APPLE__) && TARGET_OS_IPHONE) || defined(ANDROID)
    return gui2::twindow::OK;
#else
    std::vector<gui2::tval_str> items;
    int  fullwindowed  =  preferences::fullscreen()?  preferences::MAKE_WINDOWED:
preferences::MAKE_FULLSCREEN;
    std::string str = preferences::fullscreen()? _("Exit fullscreen") : _("Enter
fullscreen");
    items.push_back(gui2::tval_str(fullwindowed, str));
    items.push_back(gui2::tval_str(preferences::CHANGE_RESOLUTION,      _("Change
Resolution")));
    items.push_back(gui2::tval_str(gui2::twindow::OK, _("Close")));
    gui2::tcombo_box dlg(items, preferences::CHANGE_RESOLUTION);
    dlg.show();
    return dlg.selected_val();
#endif
}
```

以上是 base_instance 默认实现的显示修改分辨率对话框。具体逻辑是弹出个组合框，让用户去选择哪个命令，如果不修改返回 gui2::twindow::OK。要更深入理解 show_preferences_dialog 及相关逻辑，可参考"**3.10.2 添加对话框**"给出的那个例子。

6.10.3 场景分辨率

从以上分析看出，两种触发以不一样方法处理对话框，接下要叙述的处理场景则是统一的，都要先拆掉、然后重构场景。至于为什么要拆掉的原因和对话框一样，一是只有满足当前分辨率的配置才是最美观配置，二是要避免缩小时可能导致的布局失败。

```
void pump()
{
    ......
    if (event_contexts.empty() == false) {
        const          std::vector<handler*>&          event_handlers          =
event_contexts.back().handlers;

        std::deque<context>& event_contexts_b = event_contexts;
```

```
        for  (size_t  i1  =  0,  i2  =  event_handlers.size();  i1  != i2  &&  i1  <
event_handlers.size(); ++i1) {
            event_handlers[i1]->handle_event(event);
            if (display::require_change_resolution) {
                display* disp = display::get_singleton();
                disp->change_resolution();
            }
        }
    }
    ......
}
```

不管是命令触发还是拉伸触发，它们都是在 handle_event 中进行处理。一旦要修改分辨率，handle_event 就会把 require_change_resolution 置为 true，发现是 true，pump()就会调用 display::change_resolution。

```
void display::change_resolution()
{
    require_change_resolution = false;
    if (controller_) {
        hotkey::clear();

        std::map<const std::string, bool> actives;
        pre_change_resolution(actives);

        release_theme();
        // video_.
        theme::set_resolution2(theme_cfg_, screen_.getx(), screen_.gety());
        create_theme();
        post_change_resolution(actives);
        for    (std::map<const    std::string,    bool>::const_iterator    it    =
actives.begin(); it != actives.end(); ++ it) {
            set_theme_object_active(it->first, it->second);
        }
        redraw_everything();
    }
}
```

pre_change_resolution、post_change_resolution 是虚函数，display 的派生类需要重载实现自个处理分辨率变化时的逻辑。actives 存储自认为重新构造后难以判断使能/禁用状态的控件，它们的状态在 pre 时被记忆，在 post 后被恢复。举个例子，由于一些原因分辨率发生变化时刻不能聊天，相应的界面上应灰掉“聊天”按钮。拆掉重构造场景后，应用就得判断该如何设置“聊天”按钮状态，当然它可以根据其它变量判断出这按钮得灰掉，但那样一来就会让代码变复杂，这里就提供了设一个值就让知道如何恢复到变化前状态。

第七章 控件

7.1 图像（image）

内置提供了两种风格的图像控件，default 和 blits。default 只能显示一张图像，而且一定覆盖整个控件。blits 则可让自由绘制控件内容。注意，track 控件也能自由绘制内容，要在控件内捕捉鼠标事件的请使用 track。

7.1.1 标称尺寸

不管什么风格，label 用于存放图像文件，一旦文件有效，图像尺寸就是标称尺寸，否则(0, 0)。

7.1.2 api

```
// default style
void set_label(const std::string& label);

// blits style
void set_blits(const tformula_blit& blit);
void set_blits(const std::vector<tformula_blit>& blits);
```

default 风格显示 label 指定的图像，而且这图像一定覆盖整个控件。blits 风格显示 set_blits 指定的内容，虽然计算标称尺寸时会用到 label，但画出的内容完全和 label 无关。

7.1.3 blits 风格

blits 使用了“**2.2.7 渲染 API**”中介绍的 render_blit，因而它的单元可以是图像文件（LOC）、图面（SURFACE）、实心矩形（RECT）、矩形框(FRAME)和直线(LINE)。那它表示单元的类为什么是 tformula_blit，而不是 render_blit 中的 tblit？——tblit 表示位置的四个参数 x、y、width、height 必须是确定的整数，可在控件中，一些阶段不知道控件尺寸，像 pre_show，由于不知道这个尺寸，就无法得到这四个整数值了。解决办法是让这四个参数支持使用公式，公式中可用 width、height 变量，分别表示控件的渲染宽度和高度（已乘过 hdpi_scale）。tformula_blit 从 tblit 派生，功能就是增加了这四个变量。

```
struct tformula_blit: public image::tblit
{
    tformula<int> formula_x;
    tformula<int> formula_y;
    tformula<int> formula_width;
    tformula<int> formula_height;
};
```

为什么不在 tblit 直接内置这四个公式变量,这样还少了 tformulat_blit 类？——场景要用 tblit 表示要画的单元，在复杂些场景，要画单元往往上千个，这么大的基数一旦内置了四个公式变量，那增加的内存消耗，花在构造、复制上开销就造成性能问题。

下面代码在控件中画两张图像，左侧画 hp-blue.png，右侧画 hp-red.png。

```
std::vector<tformula_blit> blits;
blits.push_back(tformula_blit("misc/hp-blue.png", null_str, null_str, "(width / 2)", "(height)"));
blits.push_back(tformula_blit("misc/hp-red.png", "(width / 2)", null_str, "(width / 2)", "(height)"));
img.set_blits(blits)
```

按钮控件也有 blits 风格，使用的 API 以及技术细节和这里一样。

单元是图面时，如果传入的 x、y、w、h 都是 SURF_RATIO_CENTER，表示要缩放 surface，但不能改变宽高比，然后居中放置。

7.2 编辑框（text_box、scroll_text_box）

ttext_box、tscroll_text_box 是和编辑框相关的两种控件。ttext_box 对应单行编辑框，一旦字符串内容超过控件宽度，将有部分被隐藏。tscroll_text_box 对应可以多行的编辑框，右侧可能出现垂直滚动条。不论用户操作还是 api，两种编辑框大同小异，以下说的对它们都适用。为区分，把 ttext_box 叫编辑框，tscroll_text_box 叫滚动编辑框。

在 PC，编辑框内置了左键弹出菜单，菜单命令有复制、剪切、粘贴。在移动平台，实现了类似 iOS 的操作，长按后弹出悬浮按钮，命令有选择、复制、粘贴，进入“选择”后可通移动两侧的光标改变选择区域。

7.2.1 窗口脚本使用编辑框、标称尺寸

对编辑框中出现的字符串，只能由窗口脚本设置字号，颜色则由脚本和后面的 set_default_color 共同决定。

不论是 ttext_box 还是 tscroll_text_box，计算标称尺寸时都不会考虑框中已有字符串。

7.2.2 api

api 可分为两类：管理内容、挂接事件处理例程。

```
void set_label(const std::string& label);
const std::string& label() const;

void set_placeholder(const std::string& label);
void set_maximum_chars(const size_t maximum_chars);
void set_default_color(const uint32_t color);
void set_border(const std::string& border);
```

如果没调用过 set_default_color 或设置的颜色是 0，字符串颜色是主题对应的“normal”色。但不管有没有调用过 set_default_color，placeholder 都是主题对应的“placeholder”色。

操作二：挂接事件处理例程

	参数	描述
did_text_changed_		控件中字符串发生改变了

回调的第一个参数是编辑框自身，“参数”字段描述了其它参数。

7.2.3 密码框

编写窗口脚本时，把“password”设为 yes。

```
void set_password_char(const wchar_t unicode);
void set_cipher(const bool cipher);
```

set_password_char 设置用于表示密码的字符，参数是 unicode16 编码，默认“0x25cf”（小圆点）。set_cipher 设置是否以密文显示密码内容，默认是 true。

7.3 跟踪（track）

控件有三个特点。一是内置了拖拽鼠标的处理框架。理论上所有控件都可捕捉鼠标输入，但希望要用到捕捉的场合尽量用 track。第二个特点是直接画向 screen texture。因为它，请用 track 去显示刷新极度频繁的视频图像。第三个特点是 app 代码自由绘制前景内容。image(blits 风格)、track 都可用于自画控件，如何使用遵循两条规则。1）不需要捕捉鼠标事件的用 image，否则用 track。2）不要把 track 放在滚动控件中，它“直接”画向 screen texture，没有完整的画布缓存。

为什么 track 不能放在可滚动控件中？1）要支持滚动，则须要一个缓冲纹理，这和直接渲染冲突。2）滚动需要处理拖拽鼠标，track 本身也要处理拖拽，同时出现两种功能的拖拽会造成不太好的用户体验。

7.3.1 直接画向 screen texture

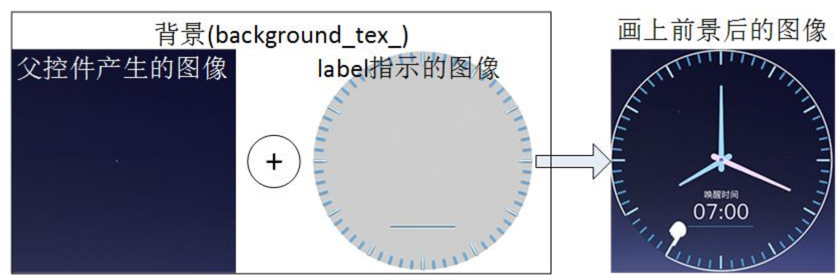


图 7-1 track 中的三部分图像

track 图像由两部分组成，一是背景（background_tex_），二是前景。背景分两部分，一是父控件产生的图像，二是自个 label 字段指示的图像。图 7-1 直观指示了这三个部分。

相比其它控件，为什么 track 渲染消耗更少 CPU？画 track 图像分两步，一是向控件覆盖 background_tex_，二是画前景。在一次 impl_draw_background 后会生成 background_tex_，只要此控件和父控件不脏（dirty_=true），background_tex_就不会变。而实际使用时，图像内容发生变化的往往只是前景。

```
boost::function<void (ttrack& widget, const SDL_Rect& draw_rect, const bool
bg_drawn)> did_draw_;
```

track 中前景完全由代码绘制，需要绘制时就会调用这函数，app 调用 set_did_draw_把自个的绘制函数挂接到 did_draw_。第一个参数是控件自个。draw_rect 是绘制矩形，非悬浮时是控件在窗口中位置，否则它的左上角是(0, 0)。bg_drawn 是 true 时表示已画过背景，否则在 did_draw_画前景前先要恢复出 background_tex_。

```
void tchat_::did_draw_vrenderer(ttrack& widget, const SDL_Rect& draw_rect, const
bool bg_drawn, bool force)
{
    SDL_Renderer* renderer = get_renderer();
    ttrack::tdraw_lock lock(renderer, widget);
    if (!bg_drawn) {
        SDL_RenderCopy(renderer,    widget.background_texture().get(),    nullptr,
&draw_rect);
    }
    ......
}
```

渲染前景之前，did_draw_需要调用示例出现的三条语句。一是得到 renderer，二是构造一个 tdraw_lock 对象，三是 bg_drawn 是 false 时恢复背景。

track 自身和 app 都会调用 did_draw_。track 有两种情况会调用 did_draw_。一是控件脏了时，此时 bg_drawn=ture。二是打开了定时器，在定时器例程中，此时 bg_drawn=false。除了 track，app 一旦发现需要重画，为效率不是把控件置脏，然后通过脏控件收集逻辑去刷新，而是直接调用 did_draw_。app 直接调用时，get_draw_rect 得到第二个参数 draw_rect，不要用 get_rect！

7.3.2 拖拽

track 提供了拖拽编程框架。一次拖拽中事件可概括为：1)一连串 is_null_coordinate(first)是 true 的 motion，2)按下左键产生 down，3)一连串 is_null_coordinate=false 的 motion，4)松开/离开控件(窗口)产生 leave。

```
boost::function<void  (ttrack&,  const  tpoint&  first,  const  tpoint&  last)>
did_mouse_motion_;
boost::function<void (ttrack&, const tpoint& last)> did_left_button_down_;
boost::function<void (ttrack&, const tpoint& first, const tpoint& last)>
did_mouse_leave_;
```

did_mouse_motion_。不管有没有按下左键，当发生 MOUSE_MOTION 时就会调用这函数。first 指示了按下左键时坐标，last 指示了此刻移动到坐标。没有按下左键之前，first 值是无效坐标，app 可用 is_null_coordinate 宏判断是否有按下鼠标左键。

```
void tchat_::did_mouse_motion(ttrack& widget, const tpoint& first, const tpoint&
```

```
last) {
    if (is_null_coordinate(first)) {
        return;
    }
    ......
}
```

did_left_button_down_。按下左键后被调用。参数 last 指示此刻坐标，这个坐标就是 motion、leave 时的 first。

did_mouse_leave_。松开或离开控件（窗口）被调用。first 指示了按下左键时坐标，last 指示了此刻移动到坐标。当是异常导致，像鼠标离开窗口，is_magic_coordinate(last)=true。“**5.4.3 事件**”有说到 mouse_focus_，“app 会遇到这么种需要，虽然鼠标离开 A 控件，但还是希望把后面的鼠标事件导向 A”，track 控件一旦被按下，默认就会把自个设为 mouse_focus_，app 可调用 set_require_capture 去修改。

7.3.3 定时器

一旦使能定时器，track 会每隔 interval_毫秒就调用 did_draw_。控件一开始关闭定时器。

```
void set_timer_interval(int interval);
```

interval>0 时，使能定时器，间隔是 interval 毫秒。“=0”则关闭定时器。

7.4 容器控件（tcontainer_）

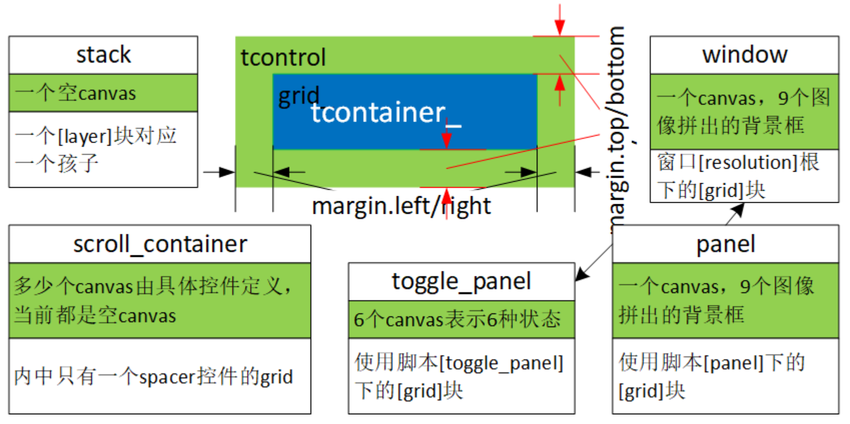


图 7-2 tcontainer_和从它派生的控件

tcontainer_不是控件，它是所有容器控件的父类，定义了容器控件的一些共有特征。控件要渲染内容分为两部分，一是表示背景的 tcontrol，二是表示孩子的 grid_。

7.4.1 tcontrol

tcontainer_从 tcontrol 派生，这里的 tcontrol 是控件自身。参考图 7-2 绿色部分，tcontrol 负责画背景，将覆盖整个控件。要画的内容写在定义控件的脚本，像 panel 来自 panel.cfg。

处理背景的逻辑和原子控件一模一样。像赋值 canvas_分两步，1）构造函数为 canvas_预分配 COUNT 个单元。2）set_definition 把控件脚本中的 tcanvas 逐一赋给 tcontrol 的 canvas_。渲染背景用 impl_draw_background。

7.4.2 grid_

grid_类型是 tgrid，存储容器控件的孩子(tcontainer_存储的是指针，对象放在派生类，这么设置的原因是有派生类会用私有 tgrid 类，像栈层控件)。除了 scroll_container，孩子内容都来自使用控件的脚本。scroll_container 使用只有一个 spacer 控件的 gird，写死在 C++代码。

见图 7-2，在放置 grid_时，grid_和控件边沿可存在外边距(margin)。margin()的返回值是 tspace4 结构，表示了四个方向上的外边距。

```
class tcontainer_ {
```

```
    tspace4 margin() const;
    void set_margin(int left, int right, int top, int bottom);
};
class tpanel: public tcontainer_ {
    tspace4 min_conf_margin() const override;
}
```

margin()会调用虚函数min_conf_margin，用它得到写在定义控件脚本中的外边距。对tpanel、ttoggle_panel、twindow 和 tscroll_panel，可调用 set_margin 在运行时修改这四个外边距，树形控件就用这方法实现了结点中的前缀空隙。

tcontainer_控件的标称尺寸包括外边距。建议用 Studio 修改脚本的方法修改 tpanel、ttoggle_panel 和 tscroll_panel 的外边距，而不是调用 set_margin。

```
void  tcontainer_::impl_draw_children(texture&  frame_buffer,  int  x_offset,  int 
y_offset) {
    grid_.draw_children(frame_buffer, x_offset, y_offset);
}
```

impl_draw_children 负责渲染孩子，它就是直接调用 grid_的 draw_children。

app 可调用 get_grid_rect 得到 grid_在窗口中矩形。

7.5 栈层（stack）

网式布局后各控件不可能出现交叉，实现简单交叉可使用悬浮控件，复杂点的就使用栈层。栈层控件让包含多个网格。定义控件时，一个[layer]块表示一个网格，[layer]块语法等同[grid]，没用“grid”名称只是为看起来更自然。代码调用“layer(int at)”得到某个网格。因为“layer”这名称，有时把栈层控件中的网格叫做层。

```
[stack]
    id = stack_panel
    mode= radio
    [layer]
        id = background
    [/layer]
    [layer]
        id = find
    [/layer]
    [layer]
        id = input_chat
    [/layer]
 [/stack]
```

定义了一个标识是 stack_panel、单选页模式、三个网格的栈层控件。“mode”属性指定模式，不同模式时控件行为会有较大区别。

模式	网格数	布局	标称尺寸	独有 API
画中画(pip)	不限	各网格在 Z 方向重叠放置。	（宽度）所有网格的最大标称宽度。（高度）所有网格的最大标称高度	
单选页(radio)	不限	各网格在 Z 方向重叠放置，但任一时刻只显示一个	正选中页的标称宽度。注 1。	set_radio_layer
垂直(vertical)	必须 2 个	上下放置	（宽度）所有网格的最大标称宽度。（高度）两网格标称高度和。各网格有 grow_factor 属性	set_splitter, set_did_can_drag, set_did_calculate_reserve, set_did_mouse_event, dragging, upward
水平(horizontal)	必须 2 个	左右放置	（宽度）两网格标称宽度和。（高度）所有网格的最大标称高度。各网格有 grow_factor 属性	和垂直一样

注 1。为什么不像画中画一样取所有 layer 中最大的标称尺寸。1）按布局规则，INVISIBLE 状态的控件不应该计算标称尺寸，layer 也是控件，一旦不符合就会造成错误。举个例子，要用 layout_init 置 text_maximum_width_为 0，而 tgrid::layout_init 只清 visible!=twidget::INVISIBLE 的栅格，此时若计算 visible==twidget::INVISIBLE 栅格标称尺寸，会使得当中存在多行文本控件时，用不是 0 的 text_maximum_width_去计算，导致算出过大标称尺寸。2）app 可能在每个 layer 放上复杂布局，每次计算所标称尺寸时算所有 layer 会影响效率。

7.5.1 画中画

画中画时只有最顶层中控件能和用户交互，因而控件中的层分为背景和最顶层的前景。背景放静态控件，像图像，可交互控件则要全放前景。渲染时会同时显示背景、前景。

为实现画中画用到的特殊收集脏控件逻辑，tstack 重载了 child_populate_dirty_list。该函数实现了这么条规则：“在书写 WML 时，栈层控件被写成最早出现的[layer]对应最底层，最后出现的[layer]对应最顶层。当前层有控件要被重画时，一定会重画位在它底下的那些层。”根据这条规则，当前层有脏控件时，位在它之下的层会自动置为脏。

auto_draw_children_是层(tgrid3)的一个变量，指示画该层的背景时是否同时画孩子。当控件的第 N 层网格脏时，为正确显示，0 到 N-1 层不能单单只画背景，还需要画孩子。但根据“5.3.9 刷新：脏控件集”写的规则，“渲染时只画脏控件链的最尾控件的孩子层”，意味着 twindow::draw 不会画 0 到 N-1 层孩子。为解决它，tgrid3 重载了 impl_draw_background，判断 auto_draw_children_是 ture 时自动画孩子，而且是默认行为，相应的，具体画孩子操作（impl_draw_children）则重载为什么都不做。那何时不须要画孩子？当该网格有控件是脏链的最后节点时，画脏链的过程已经在画孩子，要是自动画就重复了。

7.5.2 单选页

单选页指的是多页显示在同一位置，但在任一时刻只能显示一页。彩图 1 左侧面板的小地图下面就有个单选页，按下“控件”后底下显示可放入的控件集合，按下“对象”后切换为显示大地图中已存在的对象。单选页使用场合非常广，彩图 3 聊天面板就存在多个，像右侧是显示联系人还是群的树形，比它更大的整个聊天面板也是单选页，它集合聊天页、查找页（按下“查找”按钮后）、群管理页（管理某个群）。

做为单选页使用时，栈层控件中的层概念等同页。

```
void tstack::set_radio_layer(int layer)
{
    tgrid::tchild* children = grid2_->children();
    int childs = grid2_->children_vsize();
    twindow::tinvalidate_layout_blocker invalidate_layout_blocker(*get_window());
    for (int n = 0; n < childs; ++ n) {
        tgrid& grid = *(dynamic_cast<tgrid*>(children[n].widget_));
        if (n != layer) {
            if (grid.get_visible() != twidget::INVISIBLE) {
                grid.set_visible(twidget:: INVISIBLE);
            }
        } else if (grid.get_visible() != twidget::VISIBLE) {
            grid.set_visible(twidget::VISIBLE);
        }
    }
}
```

set_radio_layer 是 tstack 为辅助实现单选页提供的操作。layer 指示当前要显示的页索引，函数逻辑基本就是如所想的把要隐藏的页置为“INVISIBLE”，要显示的页置为“VISIBLE”，不应该存在“HIDDEN”页。

7.5.3 垂直

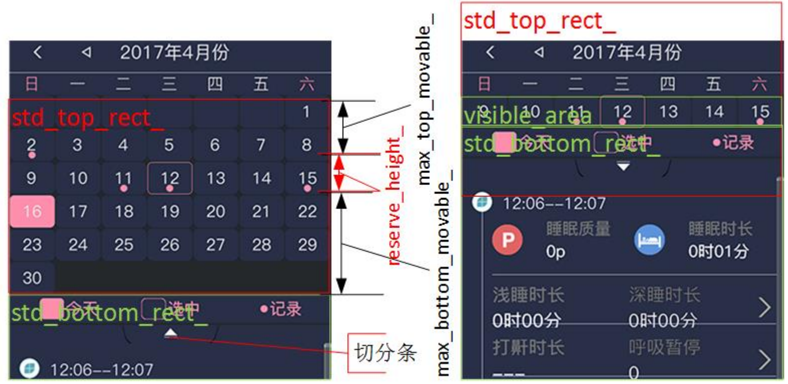


图 7-3 垂直模式的栈层控件

图 7-3 是垂直模式示例，在左侧图像按下切分条（向上白色三角形），向上拖拽，结果产生右侧图像。std_top_rect_、std_bottom_rect_分别表示了两个网格，切分后，上面网格变得只显示中间的“9-15”行，被折叠了，给下面网格腾出了更多空间。有人会问，要实现这种切分，用两个单独控件，然后配上相应代码不是就可以了，何必用专门控件？两个控件是能实现这种效果，但会需要复杂的代码逻辑。网式布局有条规则，非悬浮控件不允许交叉，而在图 7-3 右侧，栈层控件内两网格已经交叉，只有让交叉才能简化逻辑。

经常会有场合要求用垂直切分，很难抽像出一个通用处理框架，Rose 内置提供了一种实现，该实现有三个要求。1）上面网格不能有垂直滚动条。折叠上层用的是 set_origin 加 set_visible_area，不会改变渲染高度。2）会自动弹到“全折叠”状态。举个例子，要把图 7-3 移出右侧效果，不必在 y 方向向上完整移出“max_top_movable＋max_bottom_movable”像素，只要移一部分就会自动向上弹到“全折叠”，向下切分时也一样。3）须要一个切分条控件。

```
ttrack* drag_line_ = find_widget<ttrack>(&window, "drag_line", false, true);
splitter_ = find_widget<tstack>(&window, "default_stacked", false, true);

splitter_->set_splitter(*drag_line_);
splitter_->set_did_can_drag(boost::bind(&thistory::did_can_drag, this, _2, _3));
splitter_->set_did_calculate_reserve(boost::bind(&thistory::did_calculate_reserv
e, this, _1, _2, _3, _4));
splitter_->set_did_mouse_event(boost::bind(&thistory::did_draw_splitter_bar2,
this, _2, boost::ref(*drag_line_)));

drag_line_->set_did_draw(boost::bind(&thistory::did_draw_splitter_bar, this, _1,
_2, _3));
```

以上是如何使用内置实现的示例代码，分两个步骤。

1）调用 set_splitter 设置切分条控件。tstack 要向切分条挂接 down、leave、motion 处理例程。

2）向 tstack 挂接三个回调函数，并编写它们。三个函数的第一个参数都是 tstack 自身。

```
bool thistory::did_can_drag(int x, int y) const {
    return point_in_rect(x, y, drag_line_->get_rect());
}
void thistory::did_calculate_reserve(tstack& widget, int& max_top_movable, int&
reserve_height, int& threshold) const {
    tcontrol* selected = calendar_->cursel();
    max_top_movable = selected->get_y() - widget.layer(0)->get_y();
    height = selected->get_height();
    threshold = widget.layer(0)->get_height() / 4;
}
void thistory::did_draw_splitter_bar2(const int event, ttrack& widget) {
    if (event == effect::tfold::down || event == effect::tfold::leave) {
        did_draw_splitter_bar(widget, widget.get_draw_rect(), false);
    }
}
```

did_can_drag。按下鼠标时被调用，参数 x、y 指示了此刻坐标。允许切分返回 ture，否则

false。示例中只要按在切分条就认为可以开始切分。一旦不设置，坐标只要落在切分条，也就是说示例代码其实可以不挂接这函数，用默认就行。

did_calculate_reserve。开始切分时被调用，app 可设置三个参数。

参数	描述	默认值
max_top_movable	上层网格在折叠时顶部可隐藏掉的最大高度。见图 7-3 标示	0
reserve_height	要保留的高度。见图 7-3 标示	0
threshold	折叠门限，偏移一旦“>=”它，松开时将弹到“全折叠”状态	std_top_rect_.h/4

reserve_height 是 0 时，max_top_movable 依旧会发生影响，此时它表示的那一行像素将被最后隐藏。

did_mouse_event。发生 down、up、motion 时被调用，参数 event 指示了是哪种事件，第三、第四个参数是发生事件时坐标，语义参考“**7.3.2 拖拽**”。示例中切分条在下层网格，motion 时 tstack 会重画下层网格，因而没必要专门重画切分条了。

为辅助 app 编程，tstack 提供了两个函数。

```
bool dragging() const { return !is_null_coordinate(first_coordinate_); }
bool upward() const { return upward_; }
```

“dragging”用于判断当前是否处于切分状态。“upward_”指示此次可切分的方向，它由切分开始前的状态决定，中间的拖动过程不影响它。举个例子，一开始是下层折叠着，那 upward_是 true，意味着此次切分只能折叠上层。经过一次折叠成功后，upward_将变成 false，指示后面那次切分只折叠下层。

注：app 不要轻易使用 set_origin/set_visible，尤其还会让两控件出现交叉。因为“place”逻辑会确保控件不交叉，为此垂直、水平模式的栈层控件做了专门处理，见“tstack::tgrid2::place”。

7.5.4 水平

图 7-4 水平模式的栈层控件

图 7-4 是水平模式示例，在左侧图像按下切分条，向左拖拽，结果产生右侧图像。std_top_rect_、std_bottom_rect_分别表示了两个网格，切分后，左侧网格变得只显示中间一部分，被折叠了，给右面网格腾出了更多空间。和垂直一样，在图 7-3 右侧，栈层控件内两网格已经交叉。

Rose 内置提供了一种水平实现，它基本就是垂直复刻版，只不过把上面换成左侧，下面换成右侧，该实现有三个要求。1）左侧网格不能有水平滚动条。2）会自动弹到“全折叠”状态。3）须要一个切分条控件。

```
ttrack* home = find_widget<ttrack>(&window, "home", false, true);
tstack& stack = *find_widget<tstack>(&window, "default ", false, true);

stack.set_splitter(*home_);
stack.set_did_calculate_reserve(boost::bind(&tmore::did_calculate_reserve, this,
_1, _2, _3, _4));
stack.set_did_mouse_event(boost::bind(&tmore::did_mouse_event, this, _1, _2, _3,
_4));
```

不管编程步骤还是函数参数，和垂直时一样。接下让深入 did_mouse_event。

```
void tmore::did_mouse_event(tstack& widget, const int event, const tpoint& first, const tpoint& last) {
    if (event != twidget::mouse_leave) {
        diff_ = last - first;
    } else {
        if (is_magic_coordinate(last)) {
            diff_ = last - first;
        }
        if (!diff_.x || (diff_.x < 0 && abs(diff_.x) >= (int)widget.layer(0)->get_width() / 2)) {
            window_->set_retval(RETURN);
        }
    }
}
```

它实现了这么个功能，单击或向左拖拽 1/2 左侧网格宽度，关闭窗口。为和这个 1/2 一致，did_calculate_reserve 要把 threshold 设到这个 1/2 尺寸。

7.6 滚动控件（tscroll_container）

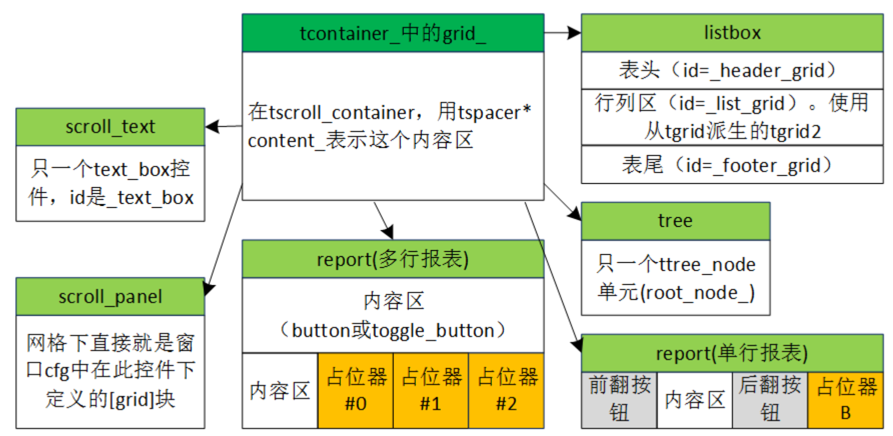


图 7-5 所有滚动控件

滚动控件指的会出现水平、垂直滚动条的控件，包括滚动编辑框(scroll_text)、滚动面板(scroll_panel)、列表(listbox)、报表(report)和树形(tree)。类血缘上，它们都从 tscroll_container 派生，tscroll_container 只是作为所有滚动控件父类，自个不对应任何控件。

7.6.1 容器和内容网格

滚动控件由两部分组成，一是容器、二是内容网格。

容器。tscroll_container 从 tcontainer_派生，本身表示了容器。grid_称为内容区，当中只一个 spacer 控件，content_指向这个占位器。界面会出现的垂直滚动条、水平滚动条是悬浮控件。

内容网格。tscroll_container 用 tgrid* content_grid_表示这个内容网格，网格 id 是“_content_grid”。被构建后就一直存在，直到控件被析构才被销毁。网格是什么内容由控件自个决定，正是这不同内容产生了多种滚动控件。

内容网格和所属窗口相互独立，它被映射向容器的内容区。用户在 content_看到的是内容网格内容，当前正被看到的内容网格部分称为视区（Visible Area）。内容网格尺寸够大、能覆盖整个 content_时，整个 content_显示视区。一旦出现内容网格小于 content_，而 content_是全透明，用户将看到背景。基于内容网格和 conetent_这种映射关系，用户移动滚动条，其实是在内容网格换个视区显示在 content_。在 content_触发事件，像单击，tscrollbar_container 会把事件导向内容网格。

内容网格和所属窗口不是完全相互独立，它的 parent_指向容器，即 tscroll_container 这个对象。当然，窗口找不到它，即 grid 的孩子中不会有内容网格，只有 content_。

7.6.2 构造、填充内容网格

构造内容网格的时机是在控件构造类的 build 函数。对 tlistbox，是 tbuilder_listbox::build，树形是 tbuilder_tree::build，依此类推。build 会调用 tscroll_container 的 finalize_setup（不是虚函数），后者实现了构造主要逻辑。

1）把 grid().widget(0,0)赋给 content_，grid()是 C++代码写出的 grid，只有一个 spacer 控件。

2）调用虚函数 mini_create_content_grid()构造内容网格，并把 this 设为该网格的父控件。

3）此时 content_grid_是个空控件，调用 mini_finalize_subclass。它是虚函数，派生类在那里把初始内容填向 content_grid_。tlistbox 是例外，填充容网格须要额外信息，mini_finalize_subclass 参数没法提供这些信息，它用的是自个的 finalize 函数。

7.6.3 标称尺寸

```
tpoint tscroll_container::calculate_best_size() const {
    return tcontrol::calculate_best_size();
}
```

tcontaienr_中 grid_中只有 spacer 控件，标称尺寸(0, 0)，因而只要计算内容网格的标称尺寸就行了。tcontrol 的 calculate_best_size 执行标称尺寸的标准计算逻辑：是否有设置 width，height，有就直接取，没有就调用 mini_get_best_text_size。

7.6.4 视区（Visible Area）

由于 content_不能显示整个内容网格，当前正被看到的内容网格部分称为视区。

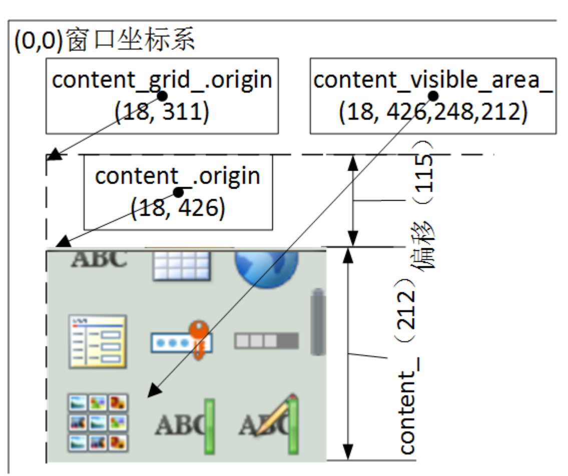


图 7-6 视区，以及相关坐标

图 7-6，图像部分是视区，同时显示了此时的内容网格（虚线）、content_、视区矩形，它们都基于窗口坐标系。由于没向左、只是向上发生移动，内容网格的原点坐标位在 content_原点正上方，其 y 坐标(311)比 content 的 y(426)小了 115，这个数值正是在垂直方向上移动了的像素数。滚动条中用 item_position_表示这个偏移，以下是垂直方向的计算公式。

```
item_postion_ = content_.y - content_grid_.y;

图 7-6，垂直滚动条中一些变量的值。
item_count_：内容网格高度，即 660。
visible_items_：content_高度，即 212。
item_poistion_：内容网格向上移动了的高度，即 115。
```

由于内容网格只可能向上、向左移，因此 item_position_不可能是负值。

content_和视区矩形（content_vsibile_area_）同属一个坐标系，那它们是不是总相等？不一定。当滚动控件是另一个滚动控件的子控件时，视区矩形将是 content_和父视区矩形的交叉部分，有可能小于 content_。

```
void tscroll_container::set_visible_area(const SDL_Rect& area)
{
```

```
    tcontainer_::set_visible_area(area);
    content_visible_area_ = intersect_rects(area, content_->get_rect());
    content_grid_->set_visible_area(content_visible_area_);
}
```

渲染视区

让看一种逻辑来实现渲染视区。

步骤一：计算内容网格中各单元的绘画方式

绘画方式分三种，NOT_DRAW：不画；DRAWN：全画；PARTLY_DRAW：部分画。图 6-39 的内容网格中有 12 个单元（右侧还有不能显示的三个单元），以下是它们的绘画方式。

PARTLY_DRAW	PARTLY_DRAW	PARTLY_DRAW	NOT_DRAW
DRAWN	DRAWN	DRAW	NOT_DRAW
DRAWN	DRAWN	DRAW	NOT_DRAW

步骤二：依次绘画各格子

有了各格子绘画方式，就可以用以下通用逻辑执行渲染

```
for (row = 0; row < rows; row ++) {
    for (col = 0; col < cols; col ++) {
        widgets[row][col]->draw_background()
    }
}
```

Rose 正是用以上两步骤实现渲染视区中内容。代码体现在两个函数：set_visible_area、draw_background。

```
void twidget::set_visible_area(const SDL_Rect& area)
{
    clip_rect_ = intersect_rects(area, get_rect());
    if (clip_rect_ == get_rect()) {
        drawing_action_ = DRAWN;
    } else if (clip_rect_ == empty_rect) {
        drawing_action_ = NOT_DRAWN;
    } else {
        drawing_action_ = PARTLY_DRAWN;
    }
}
```

twidget::set_visible_area 计算各格子绘画方式。area 是此刻正使用的视区矩形，即 6-39 中 content_visible_area_，把它和自个矩形进行交叉, 生成 clip_rect_。clip_rect_称为该控件的选区（类似 Photoshop 中选区），要画时只会限制在选区，不会涉及选区外部分。

设置好绘画方式，接下就是执行绘画。

```
void twidget::draw_background(surface& frame_buffer)
{
    if(drawing_action_ == PARTLY_DRAWN) {
        const          SDL_Rect            clipping_rectangle          =
calculate_clipping_rectangle(frame_buffer, x_offset, y_offset);
        if (!clipping_rectangle.w || !clipping_rectangle.h) {
            return;
        }
        texture_clip_rect_setter clip(&clipping_rectangle);
        impl_draw_background(frame_buffer, x_offset, y_offset);
    } else {
        impl_draw_background(frame_buffer, x_offset, y_offset);
    }
}
```

PARTLY_DRAW 方式时用 clip_rect_setter 确保此次绘画不会越出选区。因为绘画方式是 NOT_DRAWN 时不会进入此个函数（参考 twindow::draw），能满足 else 的只能是 drawing_action == DRAW。draw_children 也是类似逻辑。

渲染视区核心是要设对内容网格中各控件的选区(clip_rect_)和绘画方式(drawing_action_)。

移动视区

回看图 6-39，由于视区和内容网格同属一个坐标系，只是坐标不同，移动视区可采用两种方法。一是保持内容网格坐标不变，改变视区矩形左上角坐标，二是保持视区矩形不变，改变内容网格左上角坐标。哪种方法更好这要因地而异，针对到此处后者更好。因为视区矩形左上角坐标除了涉及内容网格还涉及归属窗口，它的左上角坐标指示它在窗口中位置!相对于窗口，内容网格矩形则是个“独立”矩形，改变它的左上角不会涉及窗口，这也正是为什么要把 content_grid_从窗口网格抽出来原因。

对移动视区，你可以这么想象：在图 6-39，窗口坐标系、content_都保持不变，只是拖着内容网格在窗口坐标系上移来移去。

那移过之后，如何保证视区中是内容网格“应该”被显示的内容？根据“渲染视区”方法，核心是要设对内容网格中各控件的选区和绘画方式，而要设对它们须要两步，一是把新左上角坐标更新到内容网格中各控件；二是虽然视区矩形没变，但由于左上角变了，要重新计算各控件的 clip_rect_和 drawing_action_。以下的 set_content_grid_origin、set_content_grid_visible_area 分别对应这两个步骤。

```
void tscroll_container::scrollbar_moved()
{
    int x_offset = horizontal_scrollbar_mode_ == always_invisible
            ? 0
            : horizontal_scrollbar_->get_item_position() *
             horizontal_scrollbar_->get_step_size();
    int y_offset = vertical_scrollbar_mode_ == always_invisible
            ? 0
            : vertical_scrollbar_->get_item_position() *
             vertical_scrollbar_->get_step_size();

    adjust_offset(x_offset, y_offset);
    const  tpoint  content_grid_origin  =  tpoint(content_->get_x()  -  x_offset,
content_->get_y() - y_offset);
    set_content_grid_origin(content_->get_origin(), content_grid_origin);
    set_content_grid_visible_area(content_visible_area_);
    content_grid_->set_dirty();
}
```

一旦发生移动，设置好 item_position_后，就会调用 scrollbar_moved()，它根据新的偏移重新设置视区。

tscrollbar_::item_positin_。视区小于内容网格时，垂直滚动条的 item_position_指示出垂直方向视区在内容网格中偏移。item_positin_单位是像素。同样的，horizontal_scrollbar_->item_position 指示出水平方向视区在内容网格中偏移。

tscrollbar_::step_size_。步长。指示移动加速度值，和 item_position_联合使用来计算“一步”引起偏移多少像素，值是 1 时不加速，>1 加速。当前固定是 1。

x_offset/y_offset 指示视区在内容网格的左上角偏移，content_grid_origin 指示要被新设置到的内容网格的左上角坐标。从代码可看出，移动视区不会改变视区矩形（content_visible_area_）。

set_content_grid_origin 用于设置 content_grid_的左上角坐标；set_content_grid_visible_area 则设置 content_grid_中所有控件视区（实际是设置显示方式）。对网格（tgrid），有专门函数用于设置左上角坐标（set_origin）、设置显示方式（set_visible_area），而 content_grid_正好是网格，这里为什么不用那两个专门函数而要用两个 tscroll_container 虚函数？——从 tscroll_container 派生的控件对 content_grid_中内容有不一样的显示要求。像列表，它存在个不论垂直滚动条移动到哪位置都要显示的表头。

```
void  tlistbox::set_content_grid_origin(const  tpoint&  origin,  const  tpoint&
content_origin)
{
    tgrid* header = find_widget<tgrid>(content_grid(), "_header_grid", true,
false);
    tpoint size = header->get_size();
```

```
    header->set_origin(tpoint(content_origin.x, origin.y));
    generator_->set_origin(tpoint(content_origin.x,     content_origin.y     +
size.y));
}
```

set_content_grid_origin 为总是显示表头（id=_header_grid），强制把表头左上角 y 值设为占位控件在窗口的左上角 y 值。既然垂直上的一部分已被表头占据，数据区（generator_）就要少去这部分区域，实现方法就是增大数据区左上角 y 值：content_origin.y + size.y。

发生移动时设置视区步骤小结

- (mini_set_content_grid_origin)设置内容网格中所有控件左上角坐标。因为控件尺寸不变，设置左上角坐标同时是改变了控件矩形。
- (mini_set_content_grid_visible_area)设置内容网格中所有控件视区。控件内部没有专门保存视区矩形的变量，设置视区实际是设置选区和绘画方式。

7.6.6 垃圾回收算法

app 会用滚动控件处理大数据，像用列表展示聊天中历史记录、影视评论，用报表展示搜索“*.png”出来的结果。对垃圾收集算法，首先是要确定什么是垃圾数据，以下用列表作为例子。

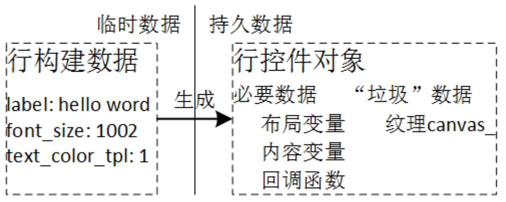


图 7-7 控件的必要和非必要数据

app 产生“构建”行需要的数据，由它们生成行控件，一旦生成控件，“构建”数据往往就被释放了。垃圾收集算法实质是当行数超过 N 时，要释放资源，这资源可以有两种，一是整个行控件，二是控件中的非必要数据。相应地，按释放的是哪种资源分两种方案。

对第一种方案，一旦删除整个行控件，会引发一连串后果。1）app 可能存有指向它的指针，那些针指将非法。2）app 要访问该控件数据，但控件被删除了，需要额外逻辑。由于删除、重构控件是自动的，会给 app 编程带来很大不方便。

对第二种方案，控件中数据可分为必要数据、非必要数据。必要数据包括布局变量、内容变量、回调函数，等等。非必要数据包括为提高效率而使用纹理缓存。实际使用时，必要数据每行大概 5K，而一个 452x52 图像，对应的纹理缓存中像素就要占用 94016 字节（452x52x4），即一个 452x52 的图像数据就抵上 18 行。

虽然必要数据相比非必要部分是要少很多，但会一直占用内存，还是希望尽可能减少，尤其是所有控件的基类 twidget。至于如何节省，就要涉及到 C/C++语法，像少用 std::string。

这里采用第二种方案，垃圾回收时回收的是行中的非必要数据。当前列表、树形实现了垃圾收集算法，要了解算法更多细节参考“列表的垃圾回收算法”。对树形，通过维护一个和当前正显示节点时刻一致的“线性”表（ttree_node** gc_lookup_），让这“线性”表特征类似列表的行表，于是就可使用列表一样的算法实现垃圾回收。

7.6.7 place 和 layout_children

内容网格中内容发生改变后，像增加、删除、隐藏了单元，那时只是设置 need_layout_=true，并不执行布局操作。要等到执行下一个 twindow::draw，那里会调用 tscroll_container 的 layout_children（这里逻辑参考“**5.3.8 中的如何形成 dirty_list_**”）。layout_children 的操作就是调用 place，相比重新布局，此时多了一个任务，要恢复到之前的视区偏移，即布局前后不改变水平、垂直滚动条位置。

place 是 twidget 的虚函数，执行着布局操作。因为还要被 layout_children 调用，place 增加了一个任务，一旦布局结束后，要恢复到之前的 item_position_。以下是步骤。

1）mini_place_content_grid。重新布局内容网格。此时内容网格放置的左上角还是 content_的左上角，为什么不用要求恢复的左上角？此次布局出的宽度、高度可能小于之前尺寸，这意味之前偏移可能会有错误，而要如何调整则要等 mini_place_content_grid 结束。为此把恢复左上放在后面的 mini_set_content_grid_origin。

2）根据新尺寸，较正水平、垂直两向方偏移，并反馈到水平、垂直滚动条。

3）mini_set_content_grid_origin 加 mini_set_content_grid_visible_area，重新设置视区。

7.6.8 show_content_rect

通过修改垂直方向的 item_position_使 rect 显示在视区，不处理水平方向。它往往是其它函数的子函数，像 tlistbox 的 scroll_to_row。

```
void show_content_rect(const SDL_Rect& rect);
```

参考图 7-8，它的实现方法是最多通过两次 set_item_position，即挪动两次内容网格。第一次是如果 rect 的底行超过视区底行，计算出差值，把这差值加到 item_position，即向上挪动内容网格，使视区底行对齐 rect 底行。第二次是如果 rect 的顶行在视区顶行上面，则直接把 rect 的顶行作为 item_position，即向下挪动内容网格，使视区顶行对齐 rect 顶行。

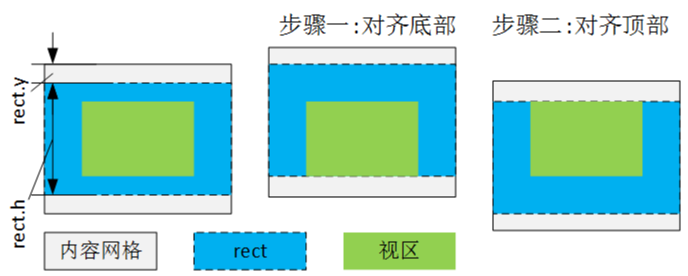


图 7-8 show_content_rect

rect 基于的是内容网格坐标系统，它的(0, 0)是内容网格的(0, 0)，不是屏幕坐标系！

7.6.9 下拉刷新（Pull refresh）

除滚动编辑框外的滚动控件都支持下拉刷新。和下拉刷新相关的可说就一个 api。

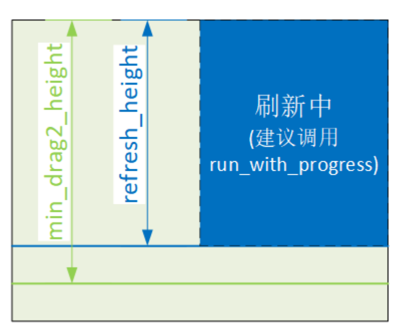


图 7-9 刷新

```
void set_did_pullrefresh(const int refresh_height, const boost::function<void (ttrack&, const SDL_Rect&)>& did_refresh, const int min_drag2_height, const boost::function<void (ttrack&, const SDL_Rect&)>& did_drag1, const boost::function<void (ttrack&, const SDL_Rect&)>& did_drag2);
```

一旦开始拖拉，进入 drag1 状态，当下拉出的高度>=min_drag2_height 时，进入 drag2 状态。随着拉出高度的变化，drag1、drag2 可以相互转换。对执行的操作，在 drag1 状态会调用 app 注册的 did_drag1，drag2 则是 did_drag2。它们的调用时机只发生在拉出的高度出现变化时。

如果停止拖拉时刻处于的是 drag1 状态，则下拉刷新结束了，否则（在 drag2 状态）会把下拉高度回弹到 refresh_height，并调用 app 注册的 did_refresh。

在 did_refresh，建议 app 弹出个限制在“rect”区域内的新窗口。1）did_refresh 之前可能会有个回弹，要正确显示回弹结果需要重画窗口，新建窗口可自动重画。2）新窗口会阻塞当前进程，并且把后面发生的事件导向这个窗口，避免了要在原窗口屏蔽掉事件的操作。

要没意外，在 did_refresh 调用 run_with_progress，当中的 rect 参数就填 did_refresh 给的“rect”。did_refresh 返回后，系统把下拉高度回弹到 0，下拉刷新结束。

7.7 报表（report）

报表用于存放若干单元， 这些单元属同一类控件，或按钮或开关。布局时，水平、垂直方向的单元间隔是同一个值，即水平方向间隔是 8 像素的话，垂直方向也只能是 8 像素。按是否可垂直扩展，分单行报表、多行报表。

单行报表/标签条。按钮全部放在一行，垂直方向上不可滚动，水平方向上，即使全单元的宽度超过屏幕尺寸，也不会用水平滚动条。以下是一些单行报表。

图 7-10 单行报表/标签条示例

A 是 Tabs。B 是分段控件（Segment Controls）。C 是一行放不下，可滚动 Tabs。D 是一行放不下，用了“前翻”、“后翻”两个按钮，翻一次时翻整段。

它们有不同叫法，但都可用单行报表去实现。有时把单行报表称标签条，单元称标签。

按标签使用的控件分为按钮式标签和开关式标签。一标签条中标签必须或全是按钮或全是开关。

一次不能显示所有标签时，也不使用滚动条，按方法不同分为分段翻（图 7-7 中 D）和滚动翻（图 7-7 中 A、B、C）。分段翻是一次翻过一段，一段指的是控件当前正显示的标签，滚动翻是鼠标拖着标签条翻。

分段翻要增加“前翻”、“后翻”两个按钮。即使当前不须要显示，依旧会给它们预留位置。

滚动翻中的标签可以不同宽度，分段翻中标签必须固定宽度。

多行报表。按报表控件可用宽度、单元宽度计算出一行可存放个数，当超过这个数目时会自动放到下一行。一旦这些行总和的行高超过控件高度，则会增加垂直滚动条。彩图 1 左侧的可用控件用了多行报表，还有像日历、电视剧剧集列表。

7.7.1 窗口脚本使用报表

脚本中 unit_width、unit_height、gap 是配置尺寸，显示时要乘上 hdpi_scale。其中 unit_width、unit_height 支持用公式，可以使用“screen_width”、“screen_height”变量。

参数	描述
multi_line	yes：多行报表。no：单行报表
toggle	yes：按钮。no：开关
unit_definition	哪风格的控件。它和 toggle 联合决定了单元是哪种控件
unit_width	单元宽度。支持用公式
unit_height	单元高度。支持用公式
gap	间隙
fixed_cols	（多行报表独有）每行单元数，0 表示自动计算
segment_switch	（单行报表独有）yes：分段翻。no：滚动翻

对 app 的 C/C++代码，当脚本没定义单元宽度/高度时，它会去取单元当时的标称宽高/高度。对多行报表，fixed_cols 非 0 时，值表示报表一行固定用几列，此时单元宽度将变成“报表控件的渲染宽度除以 fixed_cols”，基于 fixed_cols 能自适应宽度特点，可用设置了 fixed_cols 的多行报表（虽然那里只需一行）去实现 app 主页的底部导航栏。

最后渲染出的单元，单行报表中的可能不是等宽、等高，多行报表则一定等宽且等高。

```
[report]
    id = "calendar"
    toggle=yes
```

```
    unit_definition="sleep__date"
    unit_height="(if(screen_height >= 667, 45, if(screen_height >= 568, 33, 30)))"
    multi_line=yes
[/report]
```

定义了一个单元风格是“sleep__date”的开关式多行报表，控件中单元高度根据屏幕高度或是 45，或是 33，或是 30。每行 7 个单元，每个单元宽度是控件宽度除以 7，单元间隔 0 像素。

7.7.2 标称尺寸

报表的标称尺寸是内容网格加上两滚动条。下表是四种类型时内容网格的标称尺寸。

类型	宽度	高度
单行报表(分段翻)	固定 0	unit_h
单行报表(滚动翻)	内容区中所有要显示单元的宽度和	内容区中所有要显示单元的最大高度
多行报表(fixed_cols=0)	固定 0	固定 0
多行报表(fixed_cols!=0)	fixed_cols * unit_w + (fixed_cols - 1) * gap	rows * unit_h + (rows -1) * gap

7.7.3 api

api 可分为两类：管理单元、挂接事件处理例程。

操作一：管理单元

```
tcontrol& insert_item(const std::string& id, const std::string& label, int at =
npos);
void erase_item(int at = npos);
void clear();
void hide_children();

tcontrol& item(int at) const;
int items() const;

void select_item(int at);
tcontrol* cursel() const;

void set_item_visible(int at, bool visible);
bool get_item_visible(int at) const;

bool multi_select();
const std::set<tcontrol*>& selected_items();
```

参考图 7-5，位置参数（at）是内容区单元的下标，而不是报表中控件下标。对标签条，固定前置了“前翻”按钮，标签下标和报表控件下标有个固定偏移。以上提供的 api 都是只针对内容区，即 at 是内容区单元从 0 开始的下标，items()是内容区单元个数。

insert_item 在报表的 at 位置插入一个单元，at = npos 表示插在末尾。erase_item 是删除，at = npos 表示删除末尾单元。

得到单元指针后，app 是可调用 set_visible 设置隐藏/显示状态。但报表有用私有方法处理单元，直接调用它们会产生混乱，一定要调用 set_item_visible。

开关式报表时，erase_item、set_item_visible 可能会触发再选，见后面“**挂接事件处理例程**”。

multi_select、selectd_items 是和允许多选开关报表有关两个 api。

操作二：挂接事件处理例程

	类型	参数	描述
did_item_pre_change_	开关	2)要进入选中的开关。3)此刻报表内正选中开关。	某个开关将进入选中状态。函数有 bool 返回值

did_item_changed_	开关	2)选中了的开关	某个开关进入了选中状态
did_item_click_	按钮	2)触发了 click 的按钮	某个按钮触发了 click 事件

三个回调的第一个参数都是报表自身，“参数”字段描述了其它参数。

对开关式报表，希望在任一时刻有、而且只有一个开关处于选中状态。did_item_pre_change_、did_item_changed_只和选中有关，不过 did_item_pre_change_的第三个参数 previous 指示了即将失去选中的那个开关。previous 用了指针，因为即将失去选中的单元有可能是 nullptr，一个肯定会出现情况是第一次“选中”时，那时的 previous 肯定是 nullptr。。did_item_pre_change_返回 true 表示允许切换，后续将调用 did_item_changed_，返回 false 则阻止切换。

基于“对开关式报表，希望在任一时刻有、而且只有一个开关处于选中状态”这条使用原则，在调用 erase_item，要删除的开关正处于选中状态时，删除它后要自动进入 select_item()，这里会调用 did_item_pre_change_、did_item_changed_。重新 select 选择哪个单元的规则是 1）一定是 active、不隐藏的单元。2）假设正删除的索引号是 n，则搜索次序是 n，n+1，n+2，如果到尾了都没找就逆向找，n-1，n-2，一直到 0。

类似 erase_item，set_item_visible 时如果要被隐藏的是正选中单元，设置好状态后也要执行和 erase_item 类似的重新选择逻辑。

开关式报表经常和栈层/面板一块工作，组成单选页/多页面板。此时推荐做法，1）在 did_item_pre_change_判断是否允许切换，如果允许就保存当前页状态，并返回 true，否则返回 false 阻止切换。2）在 did_item_changed_放置重新加载新页代码。要注意，这里没提供关闭窗口时保存当前页的方法，这个需要 app 自个去处理。

7.7.4 上下文菜单

上下文菜单是指当中项的个数、显示图像要根据当前状态变化的菜单。图 6-1 中部队列表右侧就是个上下文菜单，当大地图没选择任何单位时，它显示“决战”、“调整”、“情报”和“系统”，当选中城市时显示“征兵”，“出征”，“重编”和“移动”，选中部队时则显示“警戒”。Rose 内置了上下文菜单实现框架，以下是该框架的几个特点。

- 整个菜单使用单行报表控件。即菜单中所有项同尺寸、同间隔，不加水平、垂直滚动条。
- 存在可同时显示的最大项数。报表控件尺寸和菜单项尺寸计算出这个最大项数。图 6-1 上下文菜单最大项数是 4。
- 菜单项默认使用“surface”风格的按钮控件。
- 支持级联菜单。级联是指按下菜单中一项可弹出下级菜单，更多见底下的“_m”后缀。
- 存在级联时，支持让菜单项成为操作的一个参数。按下菜单项会触发一个操作，N+1 级菜单项可作为展开它的 N 级菜单项的参数，更多见底下的“p”参数。

存在级联时上下文菜单将存在多个菜单，一个主菜单，其它级联出的称为子菜单。接下叙述应用如何让场景支持上下文菜单。

步骤一：编写配置

在场景配置增加[context_menu]块，一个[context_menu]块对应一个上下文菜单。

```
[context_menu]
    report = ctrl-bar
    main = build_m, recruit, expedite, armory, move, abolish, guard, extract,
advance, upload, finalbattle, employ, technologytree, list, system
    build = keep|hp, wall|hp, tower|hp
[/context_menu]
```

它定义了一个拥有两级菜单的上下文菜单。

report。菜单标识，在此场景必须存在一个该标识的单行报表。

main。定义第一级（主菜单）所有可能出现的菜单项。对于菜单项标识，“_m”是个特殊后缀，表示按下它后不是执行应用特定操作，而是展开一个下级菜单，除去“_m”后缀的字符串就是下级菜单标识。例子中存在“build_m”级联菜单项，除去“_m”后的“build”就是下级菜单标识。

build。定义级联菜单所有可能出现的菜单项。要显示级联菜单了，程序先是把原显示的菜单项全擦除，从报表控件最左侧开始逐个显示菜单项（如果须要显示）。

菜单项标识中的后缀。除去“_m”这个后缀，还有以“|”开始的后缀。“h”表示执行该菜单操作前先隐藏整个菜单。举个例子，main 菜单中“system”指示按下它后弹出“系统对话框”，在弹出前采用不隐藏菜单（没有“h”参数），这样使得关闭“系统对话框”后立即就可再按“system”。但对于建造菜单下的几个项来说，要建造“要塞（keep）”了，按下“要塞”后，是先隐藏掉菜单，避免用户重按到而让出现混乱。“p”表示此菜单项标识将做为上级菜单参数。例子中按下“keep”后，“keep”将作为“build”的一个参数，此时级联菜单等同实现了“build(keep)”。

菜单项默认使用“surface”风格的按钮控件，创建按钮时会首先去查找资源包中是否存在“buttons/菜单标识.png”。例如“system”菜单项，如果存在“buttons/system.png”这图像文件，创建时就会用它作为按钮图面。应用可在将来修改这个图面，见底下重载“prepare_show_menu”。

步骤二：重载 app_in_context_menu 等三个函数

```
void tcontext_menu::show(...) const
{
    ......
    const gui2::tgrid::tchild* children = report->content_grid()->children();
    size_t size = report->content_grid()->children_vsize();
    gui2::tpoint unit_size = report->get_unit_size();
    const std::vector<std::string>& items = menus.find(adjusted_id)->second;
    for (size_t n = 0; n < size; n ++) {
        gui2::tbutton*                          widget                          =
dynamic_cast<gui2::tbutton*>(children[n].widget_);
        if (n >= start_child && n < start_child + items.size()) {
            const std::string& item = items[n - start_child];
            if (controller.app_in_context_menu(item)) {
                widget->set_visible(gui2::twidget::VISIBLE);
                widget->set_active(controller.actived_context_menu(item));
                controller.prepare_show_menu(*widget,      item,      unit_size.x,
unit_size.y);
            } else {
                widget->set_visible(gui2::twidget::INVISIBLE);
            }
        } else {
            widget->set_visible(gui2::twidget::INVISIBLE);
        }
    }
}
```

它是处理上下文菜单时显示菜单项使用的逻辑。“adjusted_id”指示要显示的菜单，这个可能是主菜单（main）也可能是子菜单（build）。不管此个上下文菜单中存在多少个菜单，Rose 把所有菜单项规范到一条“线”上（单行报表），一个菜单对应“线”中连续一段，主菜单一定是第一段。start_child 指示该菜单中第一个菜单项在“线”的开始位置，若是主菜单，那此值是 0。items.size()是该菜单项数，start_child 开始的 items.size()个项正是属于“adjusted_id”的菜单项。针对这些项要依据 app_in_context_menu 等三函数执行进一步处理，其它项则肯定不显示。

存放在“线”上的子菜单项标识采用二级命名：子菜单标识:菜单项标识。例如“build”子菜单下的“keep”菜单项，items[n - start_child]返回的 item 值是“build:keep”。针对例子给的菜单配置，以下是它对应的“线”上菜单项排列。

```
build_m, recruit, expedite, armory, move, abolish, guard, extract, advance,
upload, finalbattle, employ, technologytree, list, system, build:keep, build:wall,
build:tower
```

要重载的三个函数都是 controller_base 成员，由它的派生类去实现，分别是 app_in_context_menu、actived_context_menu 和 prepare_show_menu。

```
bool play_controller::app_in_context_menu(const std::string& id) const
{
    std::pair<std::string,          std::string>          item          =
gui2::tcontext_menu::extract_item(id);
```

```
    ......
}
```

app_in_context_menu。判断当前是否可以显示 id 指示的菜单项，如果可以显示返回 true，否则 false。对于一级菜单，参数 id 就是菜单项标识，对于 N+1 级菜单，则是“N 级菜单项标识:N+1 级菜单项标识”，例如“build:keep”。id 参数已除去“|”后缀，但不会去除“_m”后缀。

actived_context_menu。菜单项已可以显示，判断该菜单项当前是使能还是禁用状态，如果使能返回 true，否则 false。参数 id 的意义等同 app_in_context_menu。

prepare_show_menu。菜单项已可以显示，应用可在此函数根据当前状态重画菜单项。像要显示“建造”这个级联菜单中的“要塞(keep)”项，为直观要在菜单项左上角显示建造要塞须要多少金，这个金是和当前物价指数挂钩，这时就可重载此函数、重画和这菜单项对应的控件。参数 id 的意义等同 app_in_context_menu。

步骤三：按须调用 show_context_menu、hide_context_menu

```
void display::show_context_menu(const std::string& main_id = null_str, const std::string& id = null_str);
void display::hide_context_menu(const std::string& main_id = null_str);
```

show_context_menu 的第二个参数是级联菜单标识，要没特殊情况应用调用此函数使用默认值，即显示主菜单，Rose 会处理后续的弹开级联菜单操作。“main_id”是要处理的主菜单标识，传入 null_str 时会直接取此场景定义的第一个上下文菜单，即如果场景只存在一个上下文菜单的话可让“main_id”使用默认值。

7.7.5 treport 注释

在图 7-5，占位器作用可分为两种：1）在刷新时让继续画出占位器底下的那些控件，使得画出报表要求的背景。2）当此次内容区尺寸小于上一次时，让可擦除这两次内容区差别部分。

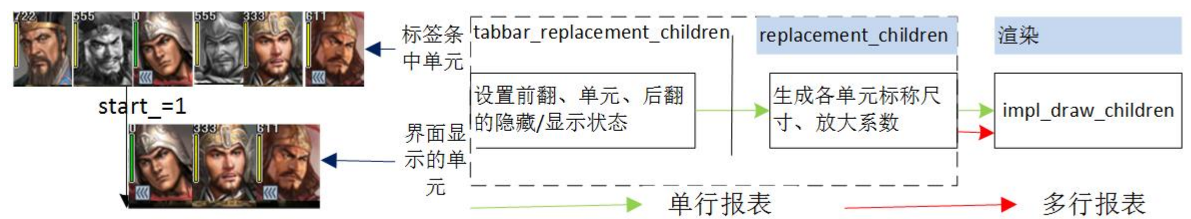


图 7-11 start_和 treport 的计算步骤

相比多行报表，标签条多了一个计算隐藏/显示各单元状态的步骤，该步骤需要 start_参数，指示此次显示要略过多少个单元，这个数目不包括那些要隐藏的。图 7-8 左侧直观显示了 start_在显示单元和实际单元之间作用，灰度图像是要隐藏的单元。

7.8 列表（listbox）

列表用于存放若干行，表示行的是 toggle_panel 控件，渲染时所有行等宽。当垂直方向一屏显示不下内容时，会在右侧增加垂直滚动条，但不会增加水平滚动条，即使列表宽度超过了一屏宽度。

由上到下，列表分为两个部分：表头（Header）和表体（Body）。表头位在表格的顶部，经常放标题，有时也叫标题行。在实现上，表头是网格控件，它不能包含多行文本。表体位在表头下面，是列表主要区域。

按是否可以让行进入选中状态，列表分单选和不选择列表。单选列表在任一时刻有、而且只有一行处于选中状态，不选择列表则是在任一时刻不会有行处于选中状态。这里没提供同时选择多行的列表，如果有 app 要使用这种列表，建议在行首放个开关控件，用开关来表示选择哪些行。列表默认是单选方式，app 可调用 enable_select 执行修改。

实现的列表框架不支持隐藏行。

7.8.1 实现建议

由于列表特点，app 经常用它去表示大数据，像聊天记录，视频评论。这里以“有 1000 条评论，评论放在网上的服务器”为例子，描述如何去实现。

增加分为主动增加、被动增加。主动增加是 app 主动调用 insert_row 向列表增加行。被动增加指的是在列表处于一些状态时框架会向 app 索要更多行，这时会调用 app 注册的 did_more_rows，app 则在这函数中调用 insert_row。什么时候会索要更多行？当前已显示了最后一行，用户依旧上拉列表。

内容间不能有“黑洞”。app 告知用户有 1000 条评论，可很快用户就想看到第 500 条，当前框架是要求必须先读出前面的 499 条。对这样问题，建议把 app 进行分段，像 100 条为一段，这样想看第 500 条，让先跳到第 5 段。

不支持隐藏行，所以一旦换了筛选器，app 应该清空列表。假设列表一开始没内容，用户只想看到 a 的评论，于是设了筛选器为“用户=a”，接下向列表增加的行都是 a 的评论。在时刻 t，用户把筛选器设为无，也就是想看到所有评论，这时建议的作法是清空列表内容，然后把评论依次加入列表。这个列表内容要由 app 来维护。

排序和后续行。listbox 支持行排序，但那是对已有行的排序，如果该列表后续还有行，对新增行，app 需要自个按设定的排序规则维护列表了。表现在代码，insert_row 时增加到的位置(at)不再是默认的末尾，而是按设定的排序规则算出要增加到的 at。

did_more_items 时增加多少行由 app 自个决定。原则上，建议能满足 3/2 屏，以便让微移时可以不必重新 did_more_items。

7.8.2 窗口脚本使用列表、标称尺寸

		控件	
表头		网格(grid)	不能包含多行文本。
表体	行	开关面板（toggle_panel）	计算等尺寸组时和表头一块计算
	弹出菜单	网格(grid)	不能存在多行文本

列表的标称尺寸是表头加上表体。高度是表头高度加上表体高度，宽度则取表头和表体的最大值。表头是个 grid，计算方法和普通网格一样。表体则是从第 0 行开始，依次计算各行尺寸，一直到累加的高度>=当前屏幕高度。这些行中最大的行宽将作为表体的标称宽度，由这些行估算出的表体高度作为表体的标称高度。

行的标称高度就是布局时行最终在用的渲染高度。窗口脚本中，会忽略 toggle_panel 的 border、对齐模式（居左、居中、居右、边沿）。

列表标称高度就是列表中内容网格最终在用的渲染高度，这意味着，内容网格的渲染高度有可能小于列表高度。

每个列表有单独的一套等尺寸组，适用于表头和行，不能用在该列表外的控件。同样，该列表外的等尺寸组不能用在列表内的表头和行。弹出菜单属于列表，但用窗口级的等尺寸组。

7.8.3 api

api 可分为两类：管理行、挂接事件处理例程。

```
void insert_row(const std::map<std::string, std::string>& data, int at = npos);
void erase_row(int at = npos);
void clear();

bool select_row(const int at);
void scroll_to_row(int at = npos);
int rows() const;
ttoggle_panel& row_panel(cons int at) const;

void    sort(const    boost::function(bool    (const    ttoggle_panel>&,    const
ttoggle_panel&>& did_compare);
```

```
int drag_at() const;
void cancel_drag();
```

位置参数（at）是表体中各行的下标，得到表示行的 ttoggle_panel 后，可调用成员 at()返回该行的在表体中下标，反过来，基于 at 可用 row_panel 得到 ttoggle_panel。

insert_row 在列表的 at 位置插入一行，at＝npos 表示插在末尾，data 中的 first 指示要 set_label 的行内控件的 id，second 是 label 值。erase_row 是删除，at＝npos 表示删除末尾行。

scroll_to_row 会确保 at 指定的行显示在视区。为了让显示，可能执行两次滚动，具体见“**7.5.7 show_content_rect**”。

sort 用于排序列表，内部实现是通过 std::stable_sort，参数是 stable_sort 要回调的比较函数，返回值需要符合 stable_sort 规范。如何得到排序前后的行位置信息？在排序前，各行的 at_是逐行加 1，stable_sort 后、sort 退出前，列表会修改各行的 at_，让依旧保持逐行加 1，因而根据 at_不能得到这信息。想过让 sort()返回 std::vector，存储排序后各行之前的 at_，但 app 往往会多“关键字”累积排序。举个例子，有一个显示武将信息的表格，它有三个可排序关键字，姓名、统率和武力，按姓名排序后得到一个 std::vector，然后又按统率排序，这是在姓名排序了的基础上，这种情况下返回 std::vector 不能从根上解决问题。这里推荐的方法是对行使用 set_cookie/cookie，不管排序过几次，怎么排，行跑到哪里，它的 cookie 不会变。

操作二：挂接事件处理例程

	参数	描述
did_row_focus_changed_	2）发生事件的行。3）true：鼠标进入该行，false：鼠标离开该行	指示鼠标移动时进入、离开某行
did_row_pre_change_	2)要进入选中的行。3)此刻报表内正选中开关。	某行将进入选中状态。函数有 bool 返回值。非选择列表不会触发该列程
did_row_changed_	2)选中了的行	某行进入了选中状态
did_row_double_click_	2)正双击的行	某行触发了双击事件

四个回调的第一个参数都是列表自身，“参数”字段描述了其它参数。

对单选列表，希望在任一时刻有、而且只有一行处于选中状态。did_row_pre_change_、did_row_changed_和选择有关，和报表不同，did_row_pre_change_没提供指示即将失去选中行的 previous 参数，app 想得到可调用 cursel()。did_row_pre_change_返回 true 表示允许切换，后续将调用 did_row_changed_，返回 false 则阻止切换。

类似单选列表，在界面单击非选择列表或在代码用 select_row，一样会触发 did_row_changed，不同的是在它之前不会有 did_row_pre_change_。

基于“对单选列表，希望在任一时刻有、而且只有一行处于选中状态”这条使用原则，在调用 erase_row 时如果要删除的行正处于选中状态时，删除它后要自动进入 select_row，这里会调用 did_row_pre_change_、did_row_changed_。重新选择哪行的规则是 1）一定是 active 行（行不可能隐藏）。2）假设正删除的索引号是 n，则搜索次序是 n，n+1，n+2，如果到尾了都没找就逆向找，n-1，n-2，一直到 0。

7.8.4 左滑弹出菜单

视区已显示内容网格的最右部分，此时依然向左滑动，可弹出个右侧对齐的菜单。

窗口脚本

如果要支持弹出菜单，让表体出现第二行，在那里放置个网格。也就是说，当表格有弹出菜单时，表体的(0,0)必须是 toggle_panel，指示如何构造行，(0,1)必须是 grid，指示如何构左滑弹

出的菜单。和 toggle_panel 一样，将忽略 grid 的 border、水平对齐、垂直对齐设置。

菜单的渲染宽度等于标称宽度，渲染高度是行高度。因为渲染宽度等于标称宽度，将忽略菜单内控件的水平对齐方式。因为渲染高度是行高度，加之菜单要放在行内，菜单标称高度必须<=任何一行的高度。一旦标称高度小于行高，垂直对齐将发生影响。举个例子，行高是 80 像素，菜单标称高度是 72，设置为上对齐时，下面的 8 像素将被留空。

注：支持左滑动弹出菜单的列表不能出现在滚动面板，原因是向左滑动列表、挪动滚动面板都涉及到挪动鼠标，这可能会引起混乱。

搜索菜单中控件

列表的 left_drag_grid()返回菜单所在网格，通过该网格去搜控件。

```
list.left_drag_grid()->find("erase", false);
```

搜索 id 是“erase”的控件，然后挂接处理例程。处理例程调用 drag_at 得到菜单是在哪行。

关闭菜单

设计原则是希望用户操作触发自动关闭菜单，这操作包括按下鼠标、弹出新窗口、删除行。当然，提供了用代码关闭菜单，具体 api 是 cancel_drag。

7.8.5 tlistbox 注释

mini_handle_gc。函数用在两种场合，一是计算标称尺寸，二是除(1)之外的场合。布尔变量 gc_calculate_best_size_指示了是哪种场合。

第二种场合时函数有两个功能，1）重新计算表示操作区的 first_at_、last_at_。2）需要的话修正 content_、conteng_grid_的渲染尺寸及 item_position。计算操作区存在两种参考，一是指定行，二是 y_offset，gc_locked_at 指示了是哪一种。不论哪种，用一样方法修正 content_、conteng_grid_的渲染尺寸及 item_position，然后向上返回修正过的 item_position。

为什么 N、N+1 次时内容网格尺寸可能变化很大？举个例子，N、N+1 时，gc_next_precise_at_都是 1，由于等尺寸组影响，#0 行高由 N 次时的 1080 变成了 N+1 次的 1306，导致同样 1000 行，内容网格高度由 1080000 变到 1306000。由于 这个变化，N 次的 item_postion 用在 N+1 次时，虽然 item_postion 值没变，可真正指向的行已是变化换大了。——解决这问题方法是向 gc_locked_at_设定指定行，后面的 mini_handle_gc 保证该行必须出现在新的操作区。

7.9 树形（tree）

树形可以让一目了然地表示某种层次和衍生关系。控件中的结点分为枝桠和叶子。底下还有枝桠或叶子的结点称为枝桠，否则属于叶子。

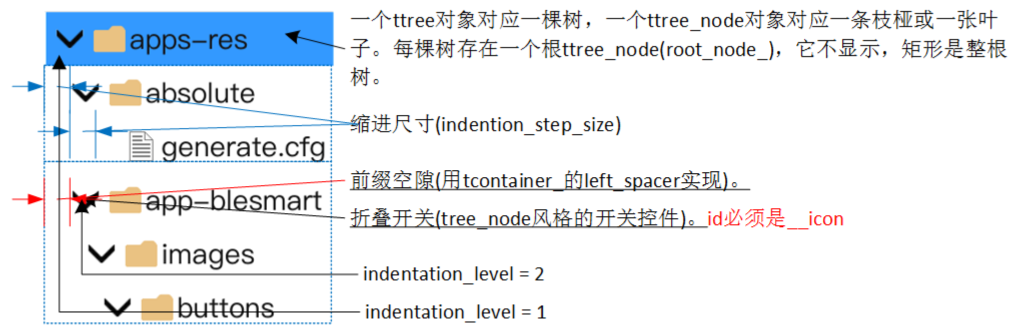


图 7-12 树形控件

一个 ttree 对象对应一棵树，一个 ttree_node 对象对应一条枝桠或一张叶子。每棵树存在一个根 ttree_node(root_node_)，它的矩形是整棵树，但不显示，不存储数据，只发挥管理功能。root_node_和 tree 的 content_grid_是什么关系？1）root_node_是 content_grid_中的孩子，而且是

唯一孩子。2）ttree 的构造函数构造出 root_node_。3）ttree::mini_finalize_subclass()设置content_grid_孩子数是一个，而且就是这个 root_node_。

ttree_node 从 ttoggle_panel 派生，不管底下是否有结点，渲染矩形只是顶上部分。除了root_node_的 parent()是 content_grid_，其它结点的 parent()都是 root_node_。要得到树中的上层节点指针，使用 parent_node()。

前缀空隙。ttree_node 的 left_spacer 实现了这个空隙。不管 indentation_level 值是多少，每个tree_node 的 x 都从一样的 content_grid_.x_开始，前缀空隙占据了该 tree_node 前面的(indentation_level-1)*indentation_step_size 部分。

折叠开关。tree_node 风格的开关控件，id 必须是__icon。系统控制它是显示还是隐藏，如果存在孩子（枝）则显示，没有（叶）则隐藏。处于折叠状态时表示选中该按钮，展开状态表示未选中。当是叶时，开关处于 HIDDEN 状态，即虽然不显示但还是占用空间。

indention_step_size_（缩进尺寸）。在 x 方向，N+1 级的开始 x 比 N 级增大了的像素数。

indentation_level（缩进级别）。它不是谁的成员，ttree_node 的 get_indention_level 计算这个值。root_node_的值是 0，往后逐级增 1。由它和 indentation_step_size 可算出前缀空隙的宽度。

7.9.1 实现建议

当树中有非常多结点时，建议按需增加节点。按须增加的核心是一开始只创建必须部分，更多的则等到用户按下折叠开关后再增加。在 pre_show，1）增加必要结点，对于确定是枝桠、但一开始不在底下增加结点的，insert_node 时把 branch 置 ture；2）向树形挂接 did_fold_changed_此函数类似以下代码。

```
void trose::did_explorer_icon_folded(ttree_node& node, const bool folded)
{
    if (!folded && node.empty()) {
        // call insert_node to insert node to this branch.
    }
}
```

7.9.2 窗口脚本使用树形、标称尺寸

不论枝桠还是叶子，WML 中都定义为[node]块。每个[node]都有一个 id，在 add_child 时，第一个参数就是这个 id。窗口配置以定义[node_definition]方法指定 node 中控件，每个[node_definition]块中有个“id”字段以唯一标识。

一棵树中可出现多个[node_definition]，每一个[node_definition]表示一种结点风格。app 在添加一个结点时可使用它们任一一种，方法是在 insert_node 第一个参数指定要使用的风格。

结点是个 toggle_panel 控件。toggle_panel 控件中有一个 id 是“__icon”的 toggle_button 控件，用于折叠、展开底下结点。

树形的标称尺寸总是(0, 0)。

7.9.3 api

api 可分为三类：管理树、管理结点、挂接事件处理例程。

```
// ttree session
void insert_node(const std::string& id, const std::map<std::string, std::string>&
data, int at = npos, const bool branch = false);
void erase_node(ttree_node& node);
void clear();

void select_node(ttree_node* node);
void scroll_to_node(const ttree_node& node);
ttree_node& get_root_node();

// ttree_node session
ttree_node& child(int at) const;
void set_widget_visible(const std::string& id, const bool visible);
```

```
void set_widget_label(const std::string& id, const std::string& label);
void fold();
void unfold();
void  sort_children(const  boost::function(bool  (const  ttree_node>&,  const
ttree_node&>& did_compare);
```

位置参数（at）是枝椏下各节点的下标，得到表示节点的 ttree_node 后，可调用成员 at()返回该结点在枝椏中下标，反过来，基于 at 可用 child 得到 ttree_node。

ttree、ttree_node 都有提供 insert_node、erase_node，ttree 版本是 ttree_node 的一种特殊情况，即要操作节点固定是 root_node_。insert_node 在枝椏的 at 位置插入一结点，at = npos 表示插在末尾，data 中的 first 指示要 set_label 的节点内控件的 id，second 是 label 值。erase_node 是删除，at = npos 表示删除末尾结点。

不管是增加枝椏还是叶子，添加都是执行 insert_node，如果查出一结点下还有结点，会把它认为是枝，否则是叶。insert_node 为什么有 branch_参数？考虑到效率，有时创建枝椏时并不同时创建孩子，而是要按下折叠开关后才创建。可由于此时没有孩子，框架会认为它是叶子，于是不在前面显示折叠开关，这就使得按下折叠开关再创建失去可能性。为解决这问题引入 branch，作用是告知框架，这个节点是枝椏，不管此刻有没有孩子，都显示折叠开关。

得到行指针后，app 是可以调用 set_visible 设置行中控件的隐藏/显示状态，调用 set_label 设置当中文字。但是，直接调用它们可能会产生混乱，app 需要用列表提供的私有 api 处理行内控件，具体是 set_widget_visible、set_widget_label。

scroll_to_node 会确保 node 指定的结点显示在视区。为了让显示，可能执行两次滚动，具体见“**7.5.7 show_content_rect**”。

sort_children 用于排序枝椏下节点，内部实现是通过 std::stable_sort，参数、比较函数的返回值，以及如何得到排序前后的节点位置信息参考“**7.7.3 api**”中 sort。

操作二：挂接事件处理例程

	参数	描述
did_fold_changed_	2）正发生折叠/展开的枝椏。3）true：该枝椏已被折叠，false：该枝椏已被展开	已经折叠、展开后才被调用
did_node_changed_	2)选中了结点	结点进入了选中后调用。
did_double_click _	2)双击了叶子	只有叶子结点才产生这回调

三个回调的第一个参数都是树自身，“参数”字段描述了其它参数。

did_double_click_。双击树椏被强制解释为折叠/展开这树椏，不会回调这函数。

7.10 模板控件

7.10.1 为什么要使用模板控件

减少控件种类。为提高加载效率，每种控件都有部分数据要常驻内存，像 label 控件的 tlabel_definition。由于是 Rose 一部分，一个设备有 N 个 app 在用 Rose 开发时，控件占用常驻内存就 N 倍增长。为节省内存，要尽量减少控件种类。举个例子，可用 track 控件实现进度条，然后提供个叫 tprogressbar_bar 的类，app 使用 tprogressbar_bar 就好像单独有个“进度条”控件。

绑定多个控件。试想下输手机号时，一个编辑框加右侧一个控钮，控钮功能或是“清空”，按下可清除框中内容，或是“明文”，按下时以明文显示密码框。在布局窗口时需要按一定规则放置这两个控件。如果每个窗口都重复这么些布局操作，一来不易维护二可能出错，为此最好能把这些控件和规则封装成一个“控件”。

7.10.2 模板控件需要的内存

常驻内存部分

从 tbuild_widget 派生的控件构造实例，像顶层控件是 tstack 时对应 tbuilder_stack。如果有竖向，会有两个 tbuild_widget 实例。全局只一个，存储在 gui.tpl_widgets。

tbuilder_tpl_widget。gui2 初始化时加载窗口，存在窗口模板(twindow_builder)中。对某个模板控件，它不是全局一个，而是被使用了多少次就有多少个。但存储在 twindow_builder 时，只是一个“控件”而已，没有扩展出内中部分，因而占用很少内存。

正在弹出窗口时

app 确定要弹出窗口 A 了，在创建窗口 A 时会调用该模板控件的 build 操作，即 tbuilder_tpl_widget::build，此时会扩展出内中部分。当窗口被销毁后，作为窗口数据一部分，tbuilder_tpl_widget::build 中的内容也会被销毁。

总的来说，使用模板控件后常驻内存大小是一个 tbuild_widget + tbuilder_tpl_widget x N，N 指的是该控件在所有窗口中的累计使用次数。

7.10.3 window.cfg 中的存储格式

示例：模板控件 id 是“progress_bar”，app 是“rose”。

```
[column]
    grow_factor=1
    horizontal_alignment="edge"
    vertical_alignment="edge"
    [rose__progress_bar]
        height="(48)"
        id="_progress"
        width="(screen_width * 80 / 100)"
    [/rose__progress_bar]
[/column]
```

解析到块名是“rose__progress_bar”，gui2 或 studio 会识别出它是一个模板控件。控件中只支持 id、width、height 字段，不支持 linked_group。

对一种模板控件，一个窗口可包含多个。

7.10.4 将模板控件挂接到实现它的类

这个挂接过程需要 app 去手动做，建议在 t<dialog>::pre_show 时执行。挂接时要知道的信息，1）该模板控件应的实现类，2）控件 id。

```
[column]
    horizontal_alignment="edge"
        [rose__text_box2]
            id="username"
        [/rose__text_box2]
[/column]
```

模板控件 rose__text_box2 对应的实现类是 gui2::ttext_box2，id 是“userame”，挂接操作就以“username”为参数构造 ttext_box2。

```
mobile_.reset(new      ttext_box2(*window_,      *find_widget<tcontrol>(window_,
"username", false, true), null_str, "misc/user96.png"));
```

模板控件实现类不从 twidget 派生。

模板控件 id	C++实现类	
rose__progress_bar	gui2::tprogress_bar	基于 track 实现的滚动条
rose__text_box2	gui2::ttext_box2	左侧图标，中间编辑框，右侧按钮
rose__chat	gui2::tchat_	聊天控件。

第八章 动画

本章目标

- MOD 时钟、程序时钟差别及转换。
- 单位动画、字符动画、地形动画的各自特点。
- 单位动画的配置语法。
- 画线的开始、结束帧块问题的原因。

动画指如何组织、播放一组相关图片，以达到特定的视觉效果。

术语

- 动画制作者：它负责播放一次“全”动画。一次要播放的动画可以有数条画轨。
- 画轨：一次动画中运动的可能不只一个角色，把一个角色的行为称为一条画轨。例如攻击动画，它一次可能有四条画轨：攻击单位攻击，防御单位防御，有领导能力单位领导，有坚定能力单位坚定。画轨是个运行时概念，只有运行时才有意义，没动画就不存在。
- 动画素材：要播放动画，不可能每次运行时都执行从图像文件形成动画的整个过程，而是从动画仓库中取出一种想要的动画素材。动画素材是和什么类型单位、执行什么动作关联。所有动画素材被以单位为仓库分类存放。动画素材是静态概念，没动画时也存在。当要播放动画时，画轨根据特定参数，主要是单位和行为从素材库中取出素材，然后进行播放。
- 画线：一个动画素材除了核心人物运动外可能还有其它行为。像魔法师用魔法攻击，它除了魔法师本身动作还有魔法球的出现、扩大和消失动作。由此区分出核心人物是一条画线，魔法球又是一条画线。
- 画帧：一条画线由数幅画帧组成。画帧是动画最小单位。
- MOD 时钟：MOD 制作者观点中的时钟。这个时钟把 MOD 编写员认为该动画要表示的“重要”事件作为 0 点，位在该事件之前的时间是负值，之后正是正值。
- 程序时钟：程序员观点中的时钟。这个时钟把动画开始时候固定为 0 点，然后累加，不会出现负值。MOD 时钟换算为程序时候：1）MOD 时钟开始时刻作为程序时候 0 点，接下时候以这个固定差值作为偏移。2）在刻度上程序时钟可以不是 MOD 时钟的 1：1，这时要按刻度比例把 MOD 时钟转换为程序时钟。

8.1 通过实例直观理解动画系统

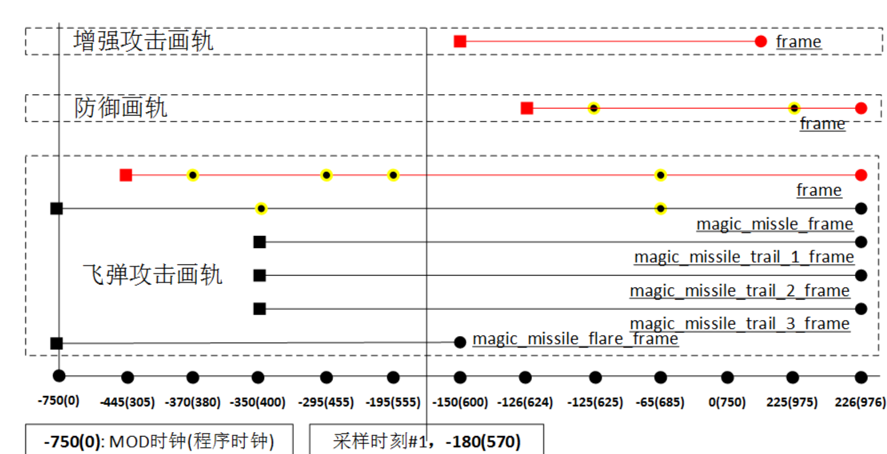


图 8-1 “飞弹”攻击动画及相关画线

图 8-1 的攻击动画涉及到三个角色，攻击部队周瑜，防御部队袁术，以及站在周瑜右下方增强攻击力的孙坚。相应地动画中有三条画轨，依次是周瑜的攻击画轨（attack_anim）、袁术的防御画轨（defend）、孙坚的增强攻击画轨（leading_anim）。图中右侧直观显示了此动画的三条画

轨、画轨中画线、以及各画线中重要时刻，而这些内容预定义在动画素材。

8.1.1 查看动画素材

存放动画配置的文件名是animation.cfg，对目录，Rose内置动画是<data>/core/units-internal，app私有则放在<data>/app-<app>/units-internal。“飞弹”功击是《王国战争》内动画，以下摘出主要部分，然后对各种块附上相关说明。

```
[anim]
    hits="hit, kill"
    offset=0
    start_time=-445
    [filter_attack]
        name="$attack_id"
        range="$ranged"
    [/filter_attack]
    [frame]
        duration=75
        image="$ranged-attack-1_png"
    [/frame]
    [frame]
        duration=75
        image="$ranged-attack-2_png"
    [/frame]
    [frame]
        duration=100
        image="$ranged-attack-3_png"
        sound="$hit_sound"
    [/frame]
    [frame]
        duration=130
        image="$ranged-attack-4_png"
    [/frame]
    [frame]
        duration=65
        image="$ranged-attack-1_png"
    [/frame]
    …
[/anim]
```

[anim]。动画素材的核心配置块，底下可能有多条画线。数个相同名称的帧块组成一条画线。

[filter_attack]（单元动画独有）。设置动画筛选条件，用于定义该动画对哪些情况是能发挥作用的。根据距离不同，兵种有三种攻击：近战（相邻）（melee）、远程（跨一格）（ranged），投射（跨两格）（cast），“ranged=ragned”指示这动画对应跨一格时的攻击。

[frame]。主画线的帧块。画轨是由数条画线组成，一系列[frame]块组成了这个动画的主画线。帧块内出现的几个字段。1）duration：该帧显示的持续时间。2）image：该帧显示的图形。3）sound：轮到该帧时会播放的音效。4）offset：用于辅助计算在哪渲染图像。参考“**x，y，offset如何决定图片被放置到的位置**”。

此个[anim]块还有四条支画线。

```
    magic_missile_start_time=-750
    magic_missile_trail_1_start_time=-350
    magic_missile_trail_2_start_time=-350
    magic_missile_trail_3_start_time=-350

    [magic_missile_frame]
        duration=400
        halo=halo/mage-halo1.png,…
        halo_x=11~0
        halo_y=-20~-54
        offset=0.001~-0.083,-0.083~-0.25,-0.25~-0.5
    [/magic_missile_frame]
    [magic_missile_frame]
        duration=350
        halo=halo/mage-halo1.png,…
        halo_y=-54~-45,-45~-27,-27~0
```

```
      offset=-0.5~-0.25,-0.25~0.25,0.25~1.0
   [/magic_missile_frame]

   [magic_missile_trail_1_frame]
      duration=350
      halo=misc/blank-hex.png:40,…
      halo_y=-54:40,-54~-45,-45~-27,-27~0
      offset=0.001:40,-0.5~-0.25,-0.25~0.25,0.25~1.0
   [/magic_missile_trail_1_frame]
   [magic_missile_trail_2_frame]
      duration=350
      halo=misc/blank-hex.png:80,…
      halo_y=-54:80,-54~-45,-45~-27,-27~0
      offset=0.001:80,-0.5~-0.25,-0.25~0.25,0.25~1.0
   [/magic_missile_trail_2_frame]
   [magic_missile_trail_3_frame]
      duration=350
      halo=misc/blank-hex.png:120,…
      halo_y=-54:120,-54~-45,-45~-27,-27~0
      offset=0.001:120,-0.5~-0.25,-0.25~0.25,0.25~1.0
   [/magic_missile_trail_3_frame]
```

注意几个用于标识块名的字符串："magic_missile_frame"、"magic_missile_trail_1_frame"、"magic_missile_trail_2_frame"和"magic_missile_trail_3_frame"，它们可统一用"xxx_frame"格式表示，当 xxx 是"magic_missile_trail_2"时是 magic_missile_trail_2_frame，依此类推。再联系到[frame]块，虽然没有"_"字符致使不满足"xxx_frame"格式，但由于[frame]块和[magic_missile_frame]实现功能都是表示一条画线，于是把[frame]称为"xxx"等于空时的"xxx_frame"。"xxx"称为帧前缀，[frame]块是前缀为空时帧块，当帧前缀非空时，它可以是 WML 制作者依主观定义的、只要满足是个不以下连线（"_"）字符开始的字符串。对位在和帧块同目录的 start_time 属性，像 magic_missile_start_time，和帧块类似它可统一为"xxx_start_time"格式，具体到 magic_missile_start_time，它和[magic_missile_frame]具有相同的前缀，则指示了 magic_missile_start_time 是给[magic_missile_frame]这条画线指定播放开始时刻。

[magic_missile_frame]、[magic_missile_trail_1_frame]、[magic_missile_trail_2_frame]、[magic_missile_trail_3_frame]这些帧块有什么作用？——一组同名帧块构造出动画的一条画线。也就是说这四组帧块用于为该动画增加四条画线。

(1)magic_missile_frame：-750 时开始播放，-750 到-350 这 400 毫秒内显示 duration=400 的[magic_missile_frame]内 halo 字段指示的图像；-350 到 0 这 350 毫秒内显示 duration=350 的[magic_missile_frame]内 halo 字段指示的图像。

(2)magic_missile_trail_1_frame：-350 时开始播放，-350 到 0 这 350 毫秒内显示 duration=350 的[magic_missile_trail_1_frame]内 halo 字段指示的图像。

(3)magic_missile_trail_2_frame：-350 时开始播放，-350 到 0 这 350 毫秒内显示 duration=350 的[magic_missile_trail_2_frame]内 halo 字段指示的图像。

(4)magic_missile_trail_3_frame：-350 时开始播放，-350 到 0 这 350 毫秒内显示 duration=350 的[magic_missile_trail_3_frame]内 halo 字段指示的图像。

四条画线加上 [frame]这条画线，五条画线独立地按自个设定的时间轴显示相应图像。那为什么要存在五条画线？——要描述一个动画，往往一条画线是不够的，就像此个飞弹攻击，为显示更为逼真，除了部队本身动作还需要有飞弹动作，而只一条画线就要同时表现出两个物体动作几乎是不可能的。作为分工，[frame]画线用于显示部队在攻击中动作，[magic_missile_frame]、[magic_missile_trail_1_frame]、[magic_missile_trail_1_frame]、[magic_missile_trail_1_frame]画线则用于显示飞弹动作，描绘飞弹用了四条画线，这是为实现飞弹更细致的动画效果。

对攻击动画来说，部队本身画线是主要的，或许可以这么说，只要部队本身画线出来了，虽然粗糙但至少已是表达出了这动画，在此把这个"核心"画线称为主画线，在配置中主画线用帧前缀是空的[frame]块表示，其它的则是支画线，用前缀非空的[xxx_frame]块表示。在使用习惯上，支画线中往往用"halo"方式指示要画图像，"halo"被称为光环，它们往往是为辅助主

画线让整个动画更逼真而存在的。

飞弹攻击动画中还有一条支画线：magic_missile_flare。也就是说这个飞弹攻击动画总共有六条画线。

```
    magic_missile_flare_start_time=-750

    [magic_missile_flare_frame]
        duration=600
        halo=halo/mage-preparation-halo1.png…
        halo_x=11
        halo_y=-20
        offset=0
    [/magic_missile_flare_frame]
```

要播放动画，自然要计算动画开始播放时刻。所有画线中最小的 xxx_start_time 值为作该动画的开始播放时刻，以上飞弹攻击动画就是[magic_missile_frame]、[magic_missile_flare_frame]设定的-750（毫秒）。WML 中定义的时刻值、持续时间值都是 MOD 时间，它会被以开始时刻对应“0”的方法换算到程序时间。MOD 时间是 WML 制作者观点中的时间，便于理解需要可以是负值，像针对攻击动画，MOD 制作者往往把攻击武器碰到对方那一刻作为参考点，即把那一刻定义为时间 0 值，而把碰到前动作的发生时刻定义为负值，碰到后定义为正值。

下面表格总结了飞弹动画中各画线的时间轴。

画线	-750(0)	-445(305)	-370(380)	-350(400)	-295(455)	-195(555)	-150(600)	-65(685)	0(750)
frame		开始	改变		改变	改变		改变	结束
magic_missile_frame	开始			改变					结束
magic_missile_trail_1_frame				开始					结束
magic_missile_trail_2_frame				开始					结束
magic_missile_trail_3_frame				开始					结束
magic_missile_flare_frame	开始						结束		

时刻格式：WML 时间（程序时间）。像-750(0)，指示 WML 时间中时刻是-750，对应的程序时间中时刻是 0。

此个飞弹攻击动画持续时间 750 毫秒。

8.1.2 MOD 时钟、程序时钟

播放动画需要运行一个公共时钟，它用于同步各画轨、画线，在给定周期内播放指定帧。

时钟要对这三条画轨进行同步，同步基本内容就是在某个时刻开始计时，作为时间零点，然后以着固定精度向前走，在走的过程中各画轨把各自重要时间点、持续时间换算到公共时间轴上。

对于同步，首要事情是计算出开始时刻，这个开始时刻就是各画轨被画的最早时刻。画轨是由画线组成，画轨的起始点就是该画轨中所有画线的最早被画时刻。针对 8-1 攻击动画，周瑜攻击画轨已在 8.1 中分析出有 6 条画线，这画轨起始时刻是-750。

对于袁术的防御画轨（defend），此个画轨起始时刻是-126。

```
[defend]
    hits="hit"
    start_time=-126
    [frame]
        duration=1
        image="units/human/commander2.png"
    [/frame]
    [frame]
        duration=350
        image="units/human/commander2-defend-hit.png"
    [/frame]
    [frame]
        duration=1
        image="units/human/commander2.png"
    [/frame]
[/defend]
```

孙坚的增强攻击画轨（leading_anim），此个画轨起始时刻是-150。

```
[leading_anim]
    start_time=-150
    [frame]
        duration=300
        image="units/human/commander2-leading.png"
    [/frame]
[/leading_anim]
```

合起这三条画轨，此个动画起始时刻是-750。

以上分析时间值时基于是 MOD 制作者观点，对 MOD 制作者来说，它会如何设计攻击动画的时间轴？——他极可能会把武器碰到敌方部队那一刻的时间作为基准时间，即时间零点，然后把碰到前时间定义为负值，碰到后时间作为正值。

很显然，时间是不可能负值的，为此程序内部不可能处理一个负值的时间。回看图 7-2，如果把所有时间固定加上 750，所有时间也就变成零或正值，也就说，MOD 制作者观点的时间只要经过一个相加运算就可被程序处理了。

```
start_tick_ = starting_frame_time_ - start_time;
```

具体到 magic_missle_frame 画线，让计算出它的三个时间点。

- -750 - (-750) = 0
- -350 - (-750) = 400
- 0 - (-750) = 750

定义两个时钟：MOD 时钟和程序时钟。MOD 制作者观点中的时间属于 MOD 时钟，它的基准时间由 MOD 制作者按事件灵活定义，时钟中的时刻值可能是负值。程序员观点中的时间属于程序时钟，时钟中的值就是把相应的 MOD 时钟值按一定运算得到，最简单、也是一般的运算就是一个加法。

MOD 时钟和程序时钟相互换算可以不只一个加法。让考虑一种情况：同样的 WML 动画配置下要把实际动画播放时间缩到配置中的一半，要达到这个目的只须在两个时钟刻度间引入系数。

```
acceleration_ = 2;
start_tick_ = (starting_frame_time_ - start_time) / acceleration_;
```

相应的如果“acceleration_ = 0.5”则变成同样的 WML 动画配置下实际动画播放时间拉伸为原来两倍。

8.1.3 实现程序时钟

在实现上，MOD 时钟基于程序时钟，因而只要实现了程序时钟，也就等同实现了动画中的两个时钟。

要实现程序时钟，它必须依赖操作系统提供的时钟，SDL_GetTicks()正好提供了这个时钟。SDL_GetTicks()返回从 SDL 库被初始化那一刻到该函数被调用那一刻之间经过的以毫秒为单位的时间，程序一运行就会初始化 SDL 库，SDL_GetTicks()返回的时间一般就等同程序启动那一刻到 SDL_GetTicks()被调用那一刻经过的时间。由于操作系统当前环境影响，像运行着不一样应用程序， SDL_GetTicks()得到的值往往是随机的，也就是说即使同一份代码、同一个SDL_GetTicks()在两次运行时得到的值极可能是不同的。但这个不同对程序时钟借它来实现自个目的并无影响，一来对系统时钟时刻，程序时钟只关心动画启动那一时刻的值，二来对于重要时刻判断用的是大于等于而不是等于。

对系统时钟时刻，程序时钟只关心动画启动那一时刻的值

让以周瑜攻击袁术来继续分析。假设周瑜攻击袁术时 SDL_GetTicks()得到的值是 5000，即动画启动那一时刻系统时钟时刻值是 5000。

以归零化过的程序时钟来描述飞弹攻击画轨 magic_missile_frame 画线中的事件。

- (0, 400)：显示 duration=400 的[magic_missile_frame]内 halo 字段指示的图像。

- (400, 750)：显示 duration=350 的[magic_missile_frame]内 halo 字段指示的图像。
- 750 之后：结束画线。

把动画启动时对应的系统时钟时刻值 5000 代入归零化过的程序时间。

- (5000, 5400)：显示 duration=400 的[magic_missile_frame]内 halo 字段指示的图像。
- (5400, 5750)：显示 duration=350 的[magic_missile_frame]内 halo 字段指示的图像。
- 5750 之后：结束画线。

当实际采样时刻是 5570 时，即图 7-2 中的采样时刻#1，对 magic_missile_frame 这画线来说它判断出应该“显示 duration=350 的[magic_missile_frame]内 halo 字段指示的图像”。内部就根据前面画线运行情况分两种情况：1）正在显示已是此个图像，不做进一步处理；2）正在显示的是“duration=400 的[magic_missile_frame]内 halo 字段指示的图像”，则把 duraion=350 图像替换上去。

对程序时钟来说，它只关心动画启动那一时刻的系统时钟时刻值，然后把它作为一个固定偏移量。基于的根据是系统时钟精度和程序时钟精度是一样的，都是毫秒。

对于重要时刻判断用的是大于等于而不是等于

既然是时钟就要依一定规则不断更新当前值，对系统时钟这个更新是由操作系统完成的，对程序时钟则要由程序自己完成，更新程序时钟操作就是一条语句。

```
current_ticks = SDL_GetTicks();
```

由于更新是由程序自己完成，更新频率也就依赖于操作系统调度，简单来说 CPU 堵不堵直接影响了更新频率。SDL_GetTicks()最小值是 1 毫秒，即程序时钟如果更新得够快，他是可以达到 1 毫秒精度，但现实系统由于 CPU 处理能力限制不可能达到这个精度，而且两次更新之间间隔极可能也不是一个固定值，还可能会发生很恶劣情况。像图 7-2 中的 magic_missile_frame 画线，假设程序时钟在采样时刻#1 时被更新，要是之前 CPU 被“堵”得厉害，致使一直不调度更新程序时钟的线程，虽然理论上 duraion=350 图像应该已被显示 170 毫秒，但正显示的图像可能还是“duration=400 的[magic_missile_frame]内 halo 字段指示的图像”！当出现这种误差时，界面上确实造成了 170 毫秒错误地显示动画，但由于判断时刻用的是大于等于而不是等于，170 毫秒之后动画会被纠正过来，界面上错误的也就是 170 毫秒而已。

当然这种误差是应该尽力被避免的，但实在没法避免时这误差不会累积，更是和系统时钟时刻值无关。

比较两时钟

	MOD 时钟	程序时钟
使用范围	编写 MOD 时	编写程序时
精度	毫秒	毫秒
零值	按动画执行的是什么动作主观定义	动画起始时刻
负值	会出现，指示是零值前播放的动画	不会出现
更新	不关心何时更新，在意的是哪时间段执行什么操作	每调用一次 current_ticks = SDL_GetTicks()更新一次
停止	动画结束	动画结束
代码中变量	以_time 后缀。start_time_	以_tick 后缀。start_tick

在 C/C++代码，MOD 时钟变量以“_time”为后缀，像表示帧开始时刻的 start_time_，画线开始时刻的 starting_frame_time_。程序时钟则以“_tick”为后缀，像表示画线开始时刻的 start_tick_，最后一次刷新的 last_update_tick_。time_to_tick 执行把 MOD 时钟转为程序时钟，tick_to_time 则把程序钟转为 MOD 时钟。

8.2 配置语法

8.2.1 素材类型

按参考位移不同，动画素材分基于单元的动画和基于矩形区域的动画。它们主要区别：前者参考位移是以主角色为起始、终点到从角色的直线；后者参考位移是以矩形左上角为起点、终点到矩形右上角的直线（主角色、从角色、参考位移参考 8.3.4 中的**“正方向、参考位移”**）。区分它们依据是块内的 area_mode 字段，值 false 是单元动画，否则区域动画。

8.2.2 帧数据

有两种地方会存在帧数据。一是画线，二是帧块。

画线（[anim]块）。语法中“xxx_”是画线前缀，它使得同一动画内画线有了不同标识，作为特殊情况，“xxx_”为空时的画线叫主画线。一画线包括两类配置：全局字段和帧块，具体到前缀是“magic_missile_”画线，以它为前缀的像 magic_missile_start_time、magic_missile_offset、magic_missile_x 都是它的全局字段，[magic_missile_frame]则是属于它的帧块。

深入一些动画配置，会发现全局字段和帧块内字段经常“重合”，像全局字段 magic_missile_x 和帧块内 x（写在帧块内时省略画线前缀）。同名字段是全局的还是帧块中，**它们区别是影响范围不一样，全局字段对整条画线有效，帧块中的则只对该帧块有效；一旦出现冲突，帧块中的值会覆盖掉全局值。**可以这么认为，全局字段是默认值，哪帧块不想使用默认值时则在块内专门设置。

帧（[frame]块）是动画的最末端配置块，是具体渲染单元。常见动画要能播放图像、声音、文本，既然帧块负责最终渲染，它就要能播放这三种内容。帧块内字段较多（一部分和画线全局字段“重复”），按要播放内容把它们进行归类。下表已把图像分成图形、光环，它们差别参考“**7.3.4 图像何时用光环、何时用图形**”。

内容	主字段	位置	修饰
图形	image	x、y、offset_x、offset_y、layer(1)	image_diagonal(2)、image_mod(3)、auto_hflip(4)、auto_vflip
光环	halo	halo_x、halo_y、offset_x、offset_y	auto_hflip、auto_vflip
文本(5)	stext		font_size、text_color
声音	sound		

(1)layer 用于设置该图形要被放置到的层，即三维坐标的 Z 轴。Layer 语义参考“6.4.1 display::draw”。同一画轨中，主画线中图像会先于支画线被放置，多条支画线时按它们在 WML 中第一个帧块的位置决定放置次序。layer 默认值是 40（LAYER_UNIT_DEFAULT - LAYER_UNIT_FIRST）。此处的 x、y 不是配置中的 x、y，如何计算(x, y)参考“**7.3.5 x，y，offset_x、offset_y 如何决定图片被放置到的位置**”。

(2)image_diagonal 指示当正方向（正方向细节参考 7.3.4 中的**“正方向、参考位移”**）是斜向（东北、东南、西南、西北）时使用的图像。正方向正处于斜向，而且它有效就优先使用它，否则使用 image。正方向是正向（南、北）时肯定使用 image。

(3)image_mod 用于变换图形，只要非空则必须以图像变换符“~”开始。像“~CROP(0, 0, 48, 48)”表示以原图像(0, 0）为左上角，裁剪出一个 48x48 的矩形为新图像。“~GS()”表示把原图像灰白化。

(4)auto_hfilp、auto_vflip 指示是否要支持自动翻转。自动翻转指的是程序会根据当前参考位移方向（参考 7.3.4 中的**“正方向、参考位移”**）执行水平或/和垂直翻转图形/光环。即使要满足各个方向，制作 MOD 时只需做两张图：正向时朝北、斜向时朝东北，当正方向不是这两个方向时程序会自动翻转图像以达到应用要求。在程序具体实现上，用于表示图形（facing_west、facing_north)、光环（orientation）翻转的变量虽然不同，但实现出的是一样结果。

(5)文本只适用基于区域的动画，而且不支持同一帧块内图像、文本共存。都设置时，图像

优先极要高于文本，播放了图像则不会播放文本。

8.2.3 图像何时用光环，何时用图形

帧内画图像有两种方法，一种是使用图形（image）（使用图形术语是为区别“图像”），一种是使用光环（halo），那么何时使用图形何时使用光环？让比较下两种方式。

	图形（image）	光环（halo）
渲染时机	在光环之前	在图形之后
擦除时机	在光环之后	在图形之前
是否涉及层（layer）	涉及。后叠加的图形可能被之前覆盖	不涉及。直接把光环图面叠加到主图面，后叠加的光环不可能被之前覆盖
是否可使用逐进变量	不能	能。恰当使用逐进变量可解决第一帧块、最后帧块 BUG
自动翻转	支持	支持
位置字段	x、y、offset、layer（Z 方向）	halo_x、halo_y、offset

注：

1. 一帧内是否可以同时存在图形、光环？理论上可以，但实际一般不会这么做，因为决定它们位置有共同变量：offset_x/offset_y。就一个值很难同时满足两种需求。

小结

- 图形和光环冲突时，光环一定能覆盖图形。对图形来说，由于要考虑到层因素，后叠加的可能还被之前覆盖。对光环来说，后叠加一定能覆盖之前的。
- 图形不能使用逐进变量，要解决第一帧块、最后帧块 BUG 需手动添加“哑”帧块。光环可使用逐进变量，恰当使用逐进变量可自动解决第一帧块、最后帧块 BUG。什么是第一帧块、最后帧块 BUG，以及图形、光环是如何解决的参考“**7.6.3 画线的初始、结束帧块 BUG**”。

8.2.4 x，y，offset_x，offset_y 如何决定图片被放置到的位置

x、y、offset_x 和 offset_y 共同用于定位图像要被放置到什么位置（以像素为单位）。

图 8-2 x、y、offset_x、offset_y 示例

记住一个设定：具体到播放一画线时往往存在主角色和从角色，主角色、从角色不是内定是谁，它根据场合在变。像攻击动画中画线，那攻击者是主角色，防御者是从角色；对于防御动画则防御者是主角色，攻击者是从角色。分清主角色、从角色的一个作用是确定一个概念：**正方向**，把主角色朝向从角色的方向定义为动画正方向。**沿着正方向，起点是主角色、终点是从角色的直线称为参考位移**。offset 表示参考位移上的相对偏移，0 表示落在起点，即主角色，1 表示落在终点，即从角色，0.5 表示落在正中央。offset 是个比例系数，只用它的话在表示偏移时会不那么自由， x、y 弥补它不足，让基于 offset 基础上再做设置，它是以像素数表示的绝对偏移，x 方向上，负数表示向左偏，正数向右；y 方向上，负数表示向上偏，正数表示向下。

知道了 offset、x、y 大致作用，就可小结下如何确定位置参数步骤：1）确定相对偏移 offset；2）如果 offset 不能满足要求，用 x、y 进行微调。

图中甘宁用投石攻击袁术，主角色是甘宁、从角色是袁术，正方向是从甘宁出发指向袁术，甘宁所在格子中心点到袁术所在中心点的直线为参考位移。上中部分是投石这条画线（为清晰给投石加了红框）在 x=0，y=0，offset_x=0.5、offset_y=0.5 时的截图，相对偏移=0.5 使得投石这图像中心位在参考位移的正中央，x、y 都等于 0，于是结果投石就位在参考位移正中央。对右下图像，相比上中 y 多出-72 像素偏移，投石就从上中向上偏移 72 像素。对左下图像，相对偏移是 1.0，即投石图像中心将落在参考位移终点，即袁术所在格子的中心。

相对偏移 offset 范围不是[0, 1]，当是负数时，它将参考位移反向延伸，大于 1 时则正向延伸。地图格子标称尺寸是 72x72，当实际在用尺寸小于标准尺寸时，像 64x64、56x56、48x48，动画逻辑处理图像尺寸一样，自动对 x、y 执行缩小。相对偏移是个比例系数，不必执行缩小。

[attack_anim]中 x、y、offset 默认值

[xxx_anim]中不存在 x、y、offset 时，程序会赋给默认值，所有 x、y 默认值都是 0。对 offset，除了[attack_anim]（攻击动画）外默认值都是 0，[attack_anim]的 offset 分两种情况。

1、 [attack_anim]块中不存在[missile_frame]。

```
offset=0~0.6,0.6~0
```

2、 [attack_anim]块中存在[missile_frame]。

```
offset=0
missile_offset=0~0.8
```

为何这么给默认值的原因

攻击可分两种，一种是攻击者扑向被攻击者执行攻击，也就是通常所说的近战，这种情况经常设置为只有攻击者这一条画线。对于这条画线，offset 默认值用 0~0.6,0.6~0，即初始攻击者是站在原地不动（offset=0 使位在参考位移起点），到中间时刻时扑到被攻者(0.6)，攻击完之后又站回原地。

另一种是攻击者使用投掷物进行攻击，也就是通常所说的远程，像红袍法师的魔法飞弹。程序把它分成两条画线，攻击者（[frame]）和投掷物（[missile_frame]）。魔法飞弹的攻击者是法师，投掷物是法球。对于攻击者这条画线，程序默认 offset=0，也就是一直站原地不动，而对于投掷物，使用默认值 0~0.8，即投掷物最远射向和被攻击者相距 0.2 的地方。——为什么 0.8 后面就没有 0.8~0 了呢？投掷物一般是不回收的，像法球到达敌方部队就消失了，弓箭射到敌方部队就消失了。但这里就存在问题，投掷物消失时刻往往并不是攻击结束时刻，攻击者还会再动几下才表示此次攻击结束，所以投掷物的 offset 一般是到时刻 0（一般表示恰好攻击到这个时刻）就停止了，而攻击者的结束时刻则大于 0。

注：

- 程序默认会特殊照顾的是[frame]和[missile_frame]这两条画线。
  当还有第三条画线，像弓箭攻击时，同时从地上冒出火来，这个火又是一条画线，假设这画线定义为[fire_frame]，这时它对应的 fire_offset 程序给的默认值是 0。
- 对于有[missile_offset]的[attack_anim]块，它会自动加的字段除了 missile_offset 外还有

missile_layer 字段和[missile_frame]块。

missile_layer 字段：missile_layer=90。这个值要比[frame]中的 layer=60（也是自动加的）高，所以如果出现坐标重叠的话，魔法球会盖住魔法师。

[missile_frame]块：会加两个[missile_frame]，分别加在已有[missile_frmae]的头和尾，结果形成的[attack_anim]至少有三个[missile_frame]，为什么要加两个 duration=1 的“哑”帧块参考“**7.6.3 画线的初始、结束帧块 BUG**”。

```
[missile_frame]
  duration=1
[/missile_frame]
[missile_frame]
  begin=-150
  end=0
  image="projectiles/missile-n.png"
  image_diagonal="projectiles/missile-ne.png"
[/missile_frame]
[missile_frame]
  duration=1
[/missile_frame]
```

8.2.5 自动缩放图像、x、y

在单元动画，image、halo 表示的图像会按栅格 zoom 进行缩放。x、y 会按 zoom 进行缩放。

在地图动画，image 表示的图像会按栅格 zoom 进行缩放。x、y 会按 zoom 进行缩放。

在图元、窗口动画，image 表示的图像 width、x 会按 zoomx 进行缩放，图像 height、y 会按 zoomy 进行缩放。那怎么计算 zoomx、zoomy？它们来自构建图元、窗口动画的 config，像 tcontrol 的 insert_animation。另外，x、y 是配置尺寸，使用时要乘上 hdpi_scale。要注意，虽然地图地画也是区域动画，但它的 x、y 不乘 hdpi_scale，一句话，和大地图有关的动画都不乘 hdpi_scale。

字段	类型	默认	语义
width	整数	0	基准宽度。zoomx = width? 1.0 * rect.w / std_w_: 1.0
height	整数	0	基准高度。zoomy = height? 1.0 * rect.h / std_h_: 1.0
constrained	bool	false	zoomx、zoomy 不一致时，是否要让相等
up	bool	true	true 支持放大，否则 zoomx、zoomy 大于 1 时，强制设置到 1。

8.3 字符动画

8.3.1 动画播放参数

字符动画的内容是一串字符，由于可以是图文混排，内容中可以夹杂图像。它的过程就是字符串产生、浮动、隐去。使用场合像攻击时给对方造成的伤害值。

字符动画按要播放字符串如何产生分两种：实时产生字符串的字符动画，动画素材中指定字符串的字符动画。不论哪种，内容都是一次指定，中间不能再修改。

实时产生字符串

攻击时产生的损血值就属于此种动画，它的特点是不须要 WML 配置。以下是这种动画的播放参数。

- 开始时刻：MOD 时钟零值。攻击时零时刻指攻方武器刚碰到敌方那一刻。
- 持续时间：“60/turbo_speed()”次刷新。turbo_speed()是个用户可指定参数，默认值是 1，刷新是个渲染系统的更新函数，两次刷新间隔至少会保证大于 10 毫秒，实际往往要远大于 10 毫秒。
- 要显示内容：指定的字符串。对于 alpha 值，初始显示时 100%，即不透明，然后每次刷新以“alpha - (-255 / 持续时间)”的新 alpha 值不断更透明。
- 显示位置：初始坐标是目标格子的(x + 36, y)，这个是字符串矩形中上坐标。然后每次刷新

以“y - (-2 * turbo_speed())”的新 y 值垂直上移。

动画素材中指定字符串

它的特点是在 WML 的帧块中出现不为空的 text 属性。

```
[frame]
    duration=75
    text=abc
    image="units/human-kingdom/mage1-magic-attack-2.png"
[/frame]
```

对比“实时产生字符串的字符动画”，此种动画的播放参数除了开始时刻不同，其它参数一样。开始时刻是该 text 归属帧块的开始显示时刻。至于帧块中的 duration 字段和它的持续时间无关。

8.3.2 程序实现

字符串动画只是支画线，而且是播放时增加（animation::start_animation 检测到 text 不为空，参考底下“**渲染字符动画的画线**”），它没有专门动画标识，它混合在特定场合的动画中，像被攻击造成的伤害值是“defend”动画，被治疗时增加的 HP 是“healed”动画。

字符动画显示字符串使用了浮动标签机制，浮动标签机制也决定字符动画的生命期，即动画持续时间。

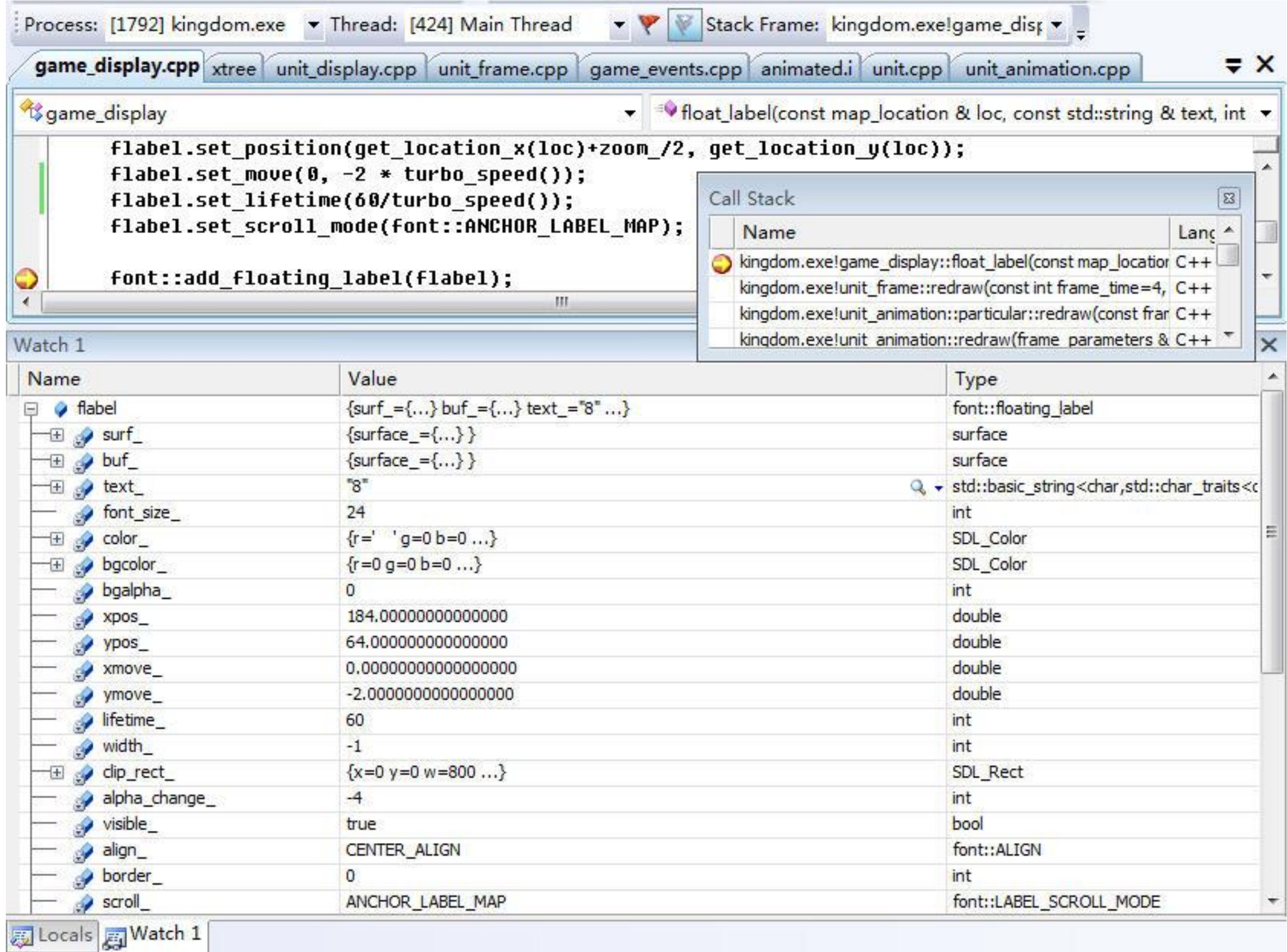


图 7-4 float_label

flabel 是个为此次字符动画构造的浮动标签对象。

- set_position：设置初始位置(xpos_, ypos_)。
- set_move：设置位置的刷新增量(xmove_, ymove_)。
- set_lifetime：设置生命期(lifetime_)及 alpha 增量(alpha_change_)。

设置中 turbo_speed()返回的是默认值 1。

通过图 7-4 的调用栈可进入 unit_frame::redraw。该函数负责构造 flabel 对象，很显然，对同一字符串的 flabel，这个对象只要构造一次就行了。这个构造一次是通过“first_time”变量。

```
void unit_frame::redraw(..., bool first_time, ...)
{
    ......
    if (first_time) {
        if (!current_data.sound_empty()) {
            sound::play_sound(current_data.sound);
        }
        if (!current_data.text.empty()) {
            game_display::get_singleton()->float_label(…,current_data.text,…);
        }
    }
    ......
}
```

first_time 指示的第一次不是该画线的第一次，而是该画线中新帧的第一次，因而画线中有多少帧，理论上就该有多少个第一次。如何计算 first_time 参考底下“图 7-9 particule::redraw”。

void floating_label::undraw(surface screen)

```
font::floating_label                    undraw(surface screen)
void floating_label::undraw(surface screen)
{
    if(screen == NULL || buf_ == NULL) {
        return;
    }

    SDL_Rect rect = create_rect(xpos(surf_->w), ypos_
    const clip_rect_setter clip_setter(screen, &clip_
    sdl_blit(buf_,NULL,screen,&rect);

    update_rect(rect);

    move(xmove_,ymove_);
    if(lifetime_ > 0) {
        --lifetime_;
        if(alpha_change_ != 0 && (xmove_ != 0.0 || ymove_ != 0.0) && surf_ != NULL) {
            // fade out moving floating labels
            // note that we don't optimize these surfaces since they will always change
            surf_.assign(adjust_surface_alpha_add(surf_,alpha_change_,false));
        }
    }
}
```

Name	Lang
kingdom.exe!font::floating_label::undraw(surface screen=	C++
kingdom.exe!font::undraw_floating_labels(surface screen	C++
kingdom.exe!display::flip() Line 1208 + 0x19 bytes	C++
kingdom.exe!display::update_display() Line 1258	C++
kingdom.exe!display::draw_wrap(bool update=true, bool	C++
kingdom.exe!display::draw(bool update=true, bool force=	C++
kingdom.exe!controller_base::play_slice(bool is_delay_ena	C++
kingdom.exe!unit_animator::wait_until(int animation_time=	C++
kingdom.exe!unit_display::unit_attack(const map_location	C++
kingdom.exe!attack::perform_hit(bool attacker_turn=true	C++

图 7-5 floating_label::undraw

- move(xmove_, ymove_)：以增加更新字符串显示位置。
- --lifetime_：生命期减一。
- surf_.assign(adjust_surface_alpha_add(surf_,alpha_change_,false))：改变要显示字符的透明度。

lifetime_指示了动画生命期，但它毕竟是用 floating_label::undraw 调用次数来表示的，这个次换算到时间上值是多少？——没法确定回答这个问题，要给个估计值可看“Call Stack”窗口，会发现是 display::draw 调用了 floating_label::undraw（也只有 display::draw 会调用 floating_label::undraw），对于字符动画生命期也就可以这么说：字符动画生命期就是开始后经过“60/turbo_speed()”次 display::draw 调用。

渲染字符动画的画线

字符动画是和单位动画一样，也通过画线进行渲染，但两种字符动画它们在选择哪画线上

存在不同。“动画素材中指定字符串的字符动画”是使用帧块归属的那条画线，画线可能是主画线也可能是支画线，“实时产生字符串的字符动画”则会新建一条名为“_add_text”的支画线。

```
void  unit_animation::start_animation(...,  const  std::string&  text,  const
Uint32 text_color, ...)
{
    ......
    if (!text.empty()) {
        particular crude_build;
        crude_build.add_frame(1,frame_builder());
        crude_build.add_frame(1,frame_builder().text(text,text_color),true);
        sub_anims_["_add_text"] = crude_build;
    }
    ......
}
```

由于添加“duration=1”空帧，“实时产生字符串的字符动画”的开始时刻严格说来是 MOD 时钟的 1 值，至于为什么要有 duration=1 的空帧，原因是画线内要是只有一画帧，动画逻辑会忽略该画帧的起始时间，致使只要动画一开始就显示该帧，参考“**7.6.3 画线的初始、结束帧块**”。

8.4 地形动画

8.4.1 概述

除去单位动画、字符动画，另有一种动画是地形动画，像流动的深水，转动的风车。

配置

地形动画配置在构造规则[terrain_graphics]的[image]块中，具体是它的文件名字段。

name 语法：name=<image name 1>:<ms 1>,<image name 2>:<ms 2>[,<image name n>:<ms n>]

```
[terrain_graphics]
    ...........
    [image]
        ..................
        name=misc/windmill-A01.png:130,misc/windmill-
A02.png:130,misc/windmill-A03.png:130,                        misc/windmill-
A04.png:130,misc/windmill-A05.png:130,misc/windmill-A06.png:130,
misc/windmill-A07.png:130,misc/windmill-A08.png:130,misc/windmill-
A09.png:130,                        misc/windmill-A10.png:130,misc/windmill-
A11.png:130,misc/windmill-A12.png:130,                        misc/windmill-
A13.png:130,misc/windmill-A14.png:130,misc/windmill-A15.png:130,
misc/windmill-A16.png:130,misc/windmill-A17.png:130,misc/windmill-
A18.png:130
        variations=";2;3;4;5;6;7;8;9;10;11"
        .................
    [/image]
    .............
[/terrain_graphics]
注:variations字段没有发挥作用,写在这里只不过是为了指示要是name中有“@V”字符,variations就会和name共同作用决定文件名。
```

此个[image]创建了一个周期是 2340(130*18)毫秒的动画，第一个 130 毫秒显示 misc/windmill-A01.png，第二一个 130 毫秒显示 misc/windmill-A02.png，第 18 个 130 毫秒显示 misc/windmill-A18.png，之后它又回到头部，即第 19 个 130 毫秒显示 misc/windmill-A01.png，依此类推。

```
[terrain_graphics]
    ...........
    [image]
        ..................
        name=grass/green.png
        variations=";2;3;4;5;6;7;8;9;10;11"
        .................
    [/image]
    .............
[/terrain_graphics]
```

以上构造中的中的 name 也会形成地形动画，只不过这个地形动画很简单，它只会显示

grass/green.png。

动画播放参数

- 开始时刻：地形动画是周期播放。第一周期的开始时刻是地形刚被显示时刻。
- 持续时间：一周期时间是 name 中各时间和。
- 要显示内容：name 中指定的图像文件。
- 显示位置：所属地形构造规则决定。

8.4.2 程序实现

假设当前 MOD 时钟是 131 毫秒，系统调用 display::draw 执行刷新大地图，格子(1, 1)在视区内，而它之上正应用着代码 7-32 的构造规则。

1. display::draw()调用 display::invalidate_animations()无效掉视区内所有格子。针对某子 loc 格子，调用 builder_->update_animation(loc)。
2. terrain_builder::update_animation 调用 tile_map_[loc] 上每个图像的 animated<T,T_void_value>::need_update()，由于时钟是 131，发现格子(1, 1)图像要进入下一画帧，调用 update_last_draw_time 让动画时进入下一画帧 misc/windmill-A02.png，并让 builder_->update_animation(loc)返回 true。
3. update_animation(loc)返回 true 倒致 display::draw 会置(1, 1)为脏格子。display::draw 会调用 display::draw_hex(loc)重绘该格。

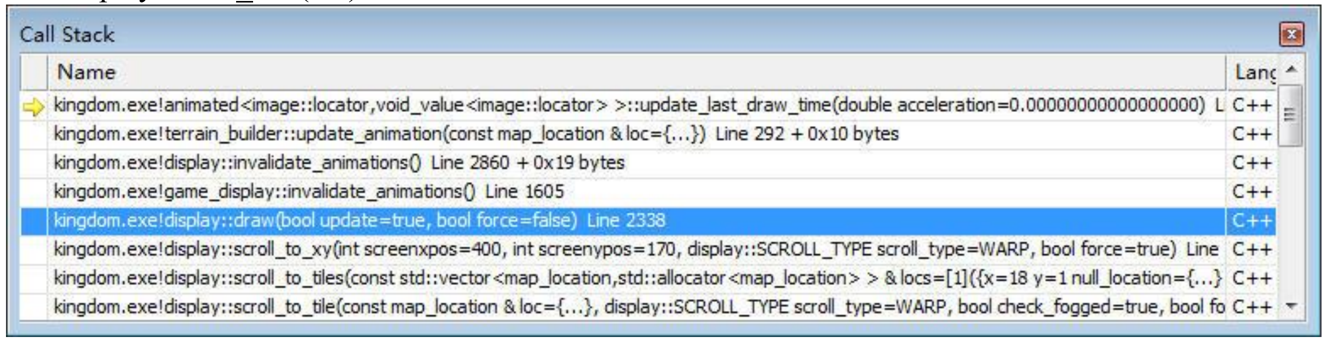


图 7-6 display::draw_hex

4. display::draw_hex 调用 display::get_terrain_images()希望得到该(1, 1)格子上所有地形图像的 surface。
5. 为得到地形图像 surface，get_terrain_images()调用 builder_->get_terrain_at()希望得到(1, 1)格子上所有地形图像的 terrain_builder::imagelist。
6. get_terrain_at 返回 tile_map_[(1, 1)]上背景/前景 terrain_builder::imagelist。
7. get_terrain_images()得到 terrain_builder::imagelist 后，针对每个 animated<image::locator>调用 get_current_frame()得到该时刻下的画帧，于是包括 misc/windmill-A02.png 的 image::locator 就被返回了，并被构造出相应的 surface。
8. 有了 loc 格子上所有地形图像 surface，display::draw_hex 把这些图像叠加到主图面。

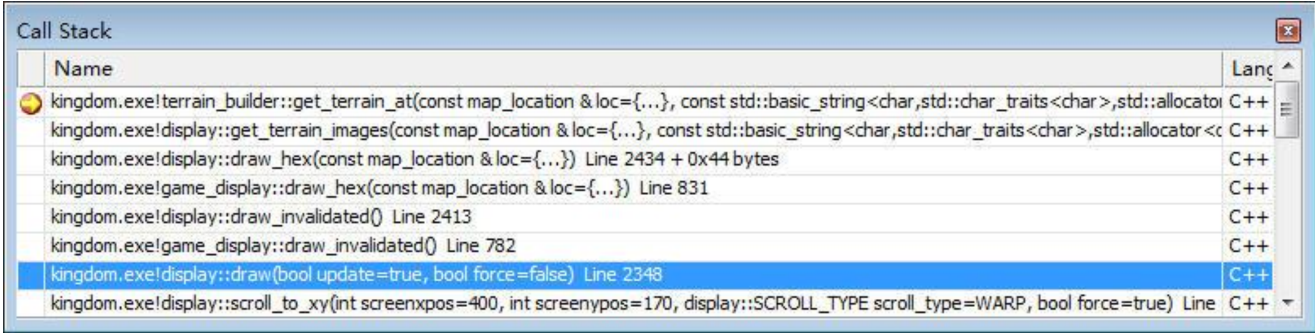


图 7-7 get_terrain_at

注

1. terrain_builder::get_terrain_at 返回的是 terrain_builder::imagelist，它的更易懂定义是 std::vector<animated<image::locator> >。
2. 得到动画中具体帧是在 display::get_terrain_images()，terrain_builder 只是负责把“当前帧号”设置为恰当值（通过 update_last_draw_time）。

8.5 模板类：animated

8.5.1 概述

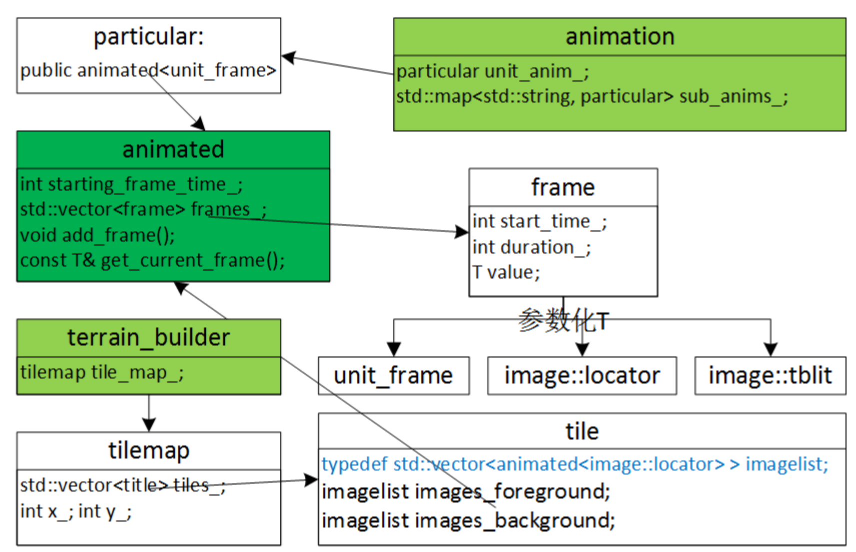


图 8-8 animated 框架

动画框架中，画线是核心概念，它向上组成画轨，向下包含画帧，代码用一个叫 animated 模板类表示画线。animated 封装了播放逻辑，模板参数 T 表示要播放的画帧。当是[frame]定义的的动画时，此时 T 对应的是 unit_frame，对 T 是 unit_frame 的画线，定义了从 animated 专门的派生类：particular。

8.5.2 animated 播放逻辑

一、帧是构成画线的单位

animated 定义一个结构，struct frame，frame 封装了帧。frame 结构只三个成员：

T value_。T 就是被实例化的类型，像 unit_frame，image::locator 和 image::tblit。value_是具体该类的一个对象，也就是 frame 要封装的对象。

int start_time_（MOD 时钟）。播放该帧的开始时刻。对于第一帧开始时刻值就是 starting_frame_time_，之后帧的开始时刻则等于上一帧开始时刻+上一帧持续时间。

int duration_。该帧的持续时间（mod 时钟）。

二、可控播放速度

double acceleration_指示播放速度。1 指示正常速度，>1，加速，<1，减速。

三、支持多画线

int starting_frame_time_（MOD 时钟），一个 animated 只封装一条画线，starting_frame_time_指示该条画线开始播放时刻。画轨中存在多画线时，有几条画线就需要几个 animated 对象。如果每条画线有不一样开始时刻，那每个 animated 对象有不一样的 starting_frame_time_值。starting_frame_time_并不是指示最早画线的播放时刻。

四、播放对象要求

unit_frame、image::locator 既然能成为播放对象，它必须得满足一些要求。

- 必须提供一个 T()构造函数。像 image::locator()。当然 T()并不是一定要求没有参数，也可以是提供默认参数的无参调用形式，像 unit_frame(const frame_parsed_parameters& builder=frame_parsed_parameters(config()))。要求存在这种构造函数，是为了能让 animated 向上返回一个表示“无效”帧的值（一般作为函数返回值）。

8.5.3 画线的开始时刻

要计算画线开始时刻（start_tick_），很自然会想到用画线“开始”（animated::start_animation）那一时刻的系统 tick，——不能用这方法，原因是要让同一动画内所有画线使用同一个 tick。同样原因，一动画内可能有多条画轨，也不能使用 animation::start_animation 那一刻的系统 tick。在用的是取自“current_ticks”，这个值取决于最近调用“new_animation_frame()”函数的时刻。

```
void animated<T,T_void_value>::start_animation(int start_time, bool cycles)
{
    started_ = true;
    last_update_tick_ = current_ticks;
    acceleration_ = 1.0;
    start_tick_ = last_update_tick_ +
        static_cast<int>(( starting_frame_time_ - start_time)/acceleration_);
    ......
}
void new_animation_frame()
{
    current_ticks = SDL_GetTicks();
}
```

由于“current_ticks”取决于最近调用“new_animation_frame()”函数的时刻，这就存在个潜在危险：要是不急时调用“new_animation_frame”，画线开始时刻会算出一个“过早”时刻，用户角度看来，这个“过早”开始时刻会导致该有的动画不显示。为什么会造成不显示让举个例子，假设存在这样一画线，以系统时钟表示的理论开始时刻是 98000（调用 animated::start_animation 时刻），持续时间是 300，但由于没急时调用“new_animation_frame”，画线开始时刻被“过早”到 97000（即最近一次调用“new_animation_frame”是在系统时钟等于 97000 时）；伴随时间流逝 100，画线内计算出的当前时刻是 97100，但外面却拿着 98100 这个当前时刻和画线来比较，由于 1000 大于 300，导致程序认为这条画线已经结束，除了第一帧，其它帧都不会被显示了。

既不能在画线也不能在画轨的 start_animation 调用“new_animation_frame”，为正确计算 current_ticks，只能由外面急时调用，像 unit_animator::start_animations 中要启动某一动画前。

8.5.4 画线开始、结束额外处理

```
void animation::particular::redraw(const frame_parameters& value,const
map_location &src, const map_location &dst)
{
    if (get_animation_time() < get_begin_time()) {
        return;
    } else if (gt_animation_time() > get_end_time()) {
        Return;
    }
    const unit_frame& current_frame = get_current_frame();
    const frame_parameters default_val =
parameters_.parameters(get_animation_time() -get_begin_time());
    if(get_current_frame_begin_time() != last_frame_begin_time_ ) {
        last_frame_begin_time_ = get_current_frame_begin_time();
    current_frame.redraw(get_current_frame_time(),true,src,dst,&halo_id_,default
_val,value);
    } else {
    current_frame.redraw(get_current_frame_time(),false,src,dst,&halo_id_,defaul
t_val,value);
    }
```

```
}
```

redraw 函数中，第一个 if 用于额外处理当画线的开始时刻晚于动画的开始时刻，第二个 if 则处理画线的线束时刻早于动画的结束时刻。一旦发生，不再调用画线的 redraw。

get_current_frame_begin_time()返回当前要播放帧的开始时刻，last_frame_begin_time_是上一帧的开始时刻。如果程序是第一次进入 particular::redraw 函数，last_frame_begin_time_值是该画线没有播放过任何帧的值，即执行 particular::start_animation 后，这个值是 get_begin_time() – 1。

再看下 get_current_frame_begin_time()，只要画线中有帧块，它肯定会返回有效帧块的开始时刻，特殊情况下，当是第一次调用 get_current_frame_begin_time()时，即使当前时刻并未到第一帧号的开始时刻，但它还是会返回 frames_[current_frame_key_].start_time_！

第一次进入 particular::redraw 时，last_frame_begin_time_值是“get_begin_time() – 1”，肯定满足 last_frame_begin_time_ != last_frame_begin_time_，致使没到开始时刻画线就被播放了！为此需要用第一个“if”额外处理。

对于画线提早结束看 animated<T,T_void_value>::update_last_draw_time。

```
template<typename T, typename T_void_value>
void animated<T,T_void_value>::update_last_draw_time(double acceleration)
{
    ......
    if(cycles_) {
        while(get_animation_time() > get_end_time()){ // cut extra time
            start_tick_                                                   +=
static_cast<int>(get_animation_duration()/acceleration_);
            current_frame_key_ = 0;
        }
    }
    if(get_current_frame_end_time() < get_animation_time() && // catch up
            get_current_frame_end_time() < get_end_time()) {// don't go after the
end
        current_frame_key_++;
    }
}
```

要改变当前帧号 current_frame_key_须满足两个条件：get_current_frame_end_time() < get_animation_time()和 get_current_frame_end_time() < get_end_time()，当动画已播放时间大于了该画线应该结束时刻时，满足条件 get_current_frame_end_time() < get_animation_time()，但不满足 get_current_frame_end_time() < get_end_time()，除非它是个周期播放动画，像地形动画。周期动画时 cycles_=true，一旦动画已播放时间大于该画线应该结束时刻它会把下一要播放的帧转回到头部，即 current_frame_key_=0。但此种情况下周期动画不会执行 current_frame_key_++，因为它在把 current_frame_key_置 0 同时更新了 start_tick_，使两个条件都不满足。

动画已播放时间大于了该画线应该结束时刻，但画线却一直停留在最后 current_frame_key_等于最后一帧号，动画中看到就是该画线一直在画最后一帧块的最后图像（画帧中可能用了逐进图像变量）。

从分析可以看出，结束帧块问题并不仅仅是只有一帧的画线才存在，只要是非周期画线都有这问题。为此需要用第二个“if”额外处理。

第九章 国际化和 gettext

本章目标

- 非 UNICODE 程序不能国际化根源
- UNICODE 程序缺陷
- UNICODE、UTF-8 之间转换规则
- pot、po、mo 之间关系
- 理解 locale
- 运行时切换语言
- 如何支持多种输出格式

术语

- NLS（Native Language Support）：本地语言支持。
- 地区码（ANSI 编码）：特定于地区的字符编码，码表中收录的字符局限于该地区。像中文（简体，中国）就是 GB2312；中文（繁体，台湾）是 Big5。它的对立概念是 UNICODE 编码，因而有时也叫非 UNICODE 码。ANSI 和 UNICODE 都是字符代码的一种表示形式。为使计算机支持更多语言，通常使用 0x80~0xFF 范围的 2 个字节来表示 1 个字符。不同 ANSI 编码之间互不兼容，当信息在国际间交流时，无法将属于两种语言的文字，存储在同一段 ANSI 编码的文本中。
- 统一码：全球统一的字符编码，码表中收录全球在用的所有字符。像 UNICODE、UTF-8。
- UTF（Unicode Transformation Format）（此中 Unicode 不是指 UNICODE 编码）：通用转换格式。UNICODE 编码用两个字节表示 ASCII 字符，对 ASCII 来说高字节的 0 毫无用处，为了解决类似存储效率低问题，就出现了一些中间格式的字符集，它们被称为通用转换格式，UTF-8 就是当中的一种。

以用户角度说，程序国际化要求指的是可以用指定语言来显示界面上信息，像伤害，让英文时显示“Damage”，简体中文时显示“伤害”，繁体中文则是“傷害”。为达到这个要求，程序可以写成一次只支持一种，即通常听到英文版、简体中文版、繁体中文版，对国际化要求更强的须要做到程序可以运行时切换语言，即多国语言版。多国语言版是同一个安装包可适用于“任何”语言，程序一般通过两种方法让选择界面上语言，1：第一次启动时以默认语言启动，像查到当前系统是简体中文 Win 7 时自动选择简体中文；2：给用户提供一个切换语言命令，像菜单项命令，一按命令就弹出有列表框窗口，列表框列出程序能支持的所有语言，切换当中列表项就可实时切换掉界面语言。

要实现的是多国语言版，程序在逻辑上要大致分两个任务，1：以一种或多种统一码表示程序中字符；2：设计一种可让一个程序包“存储”并切换多语言的机制。UNICODE、UTF-8 实现第一个任务，gettext 实现第二个任务。

UNICODE 和 UTF-8 都是统一码，都可表示全世界所有字符。理论上说，要实现国际化只靠 UNICODE 就够了，但由于效率、历史等原因，只使用 UNICODE 会遇到编程麻烦，为此要加入 UTF-8。虽然 UTF-8 是基于 UNICODE 才存在的编码，但它的“变长”特性使得编程起来比 UNICODE 还顺手，往往导致 程序中 UTF-8 用得比 UNICODE 还要广泛。

gettext 是个 GNU 开源项目，它使用 locale 设计出一种可让一个程序包“存储”并切换多语言的机制。程序中在不同语言须显示不一样信息的字符串称要翻译字符串，gettext 汇总程序所有要翻译字符串，形成一个或多个扩展名是 pot 的模板文件，例如 wesnoth.pot，然后基于 wesnoth.pot 针对不同语言形成该语言下扩展名是 po 的文件，例如英文时叫 en_GB.po，简体中文时 zh_CN.po，繁体中文是 zh_TW.po，最后把各个文本格式 po 一对一生成扩展名是 mo 的二进制格式文件，mo 就是要包含在发布程序中的语言包部分。由于程序中一种语言对应一个 mo，

当程序设置到某一语言时就在那语言 mo 中找，像英文在 en_GB.mo，简体中文在 zh_CN.mo，繁体中文在 zh_TW.mo。既然 po 和 mo 是一一对应，为何要多一步骤去生成个 mo，这除了 mo 是更适于发布的二进制格式，另外原因是 po 在生成 mo 时，mo 在存储这些上对运行时要到来的搜索操作于做了些优化。

9.1 统一码

9.1.1 Windows 下的 UNICODE、非 UNICODE 程序

UNICODE 程序指的是编译时定义了 UNICODE 宏的程序，相应的，编译时没定义 UNICODE 宏的程序叫非 UNICODE 程序。要开发两种程序，Visual Studio C++ .net 就是设置不同的“Character Set”。

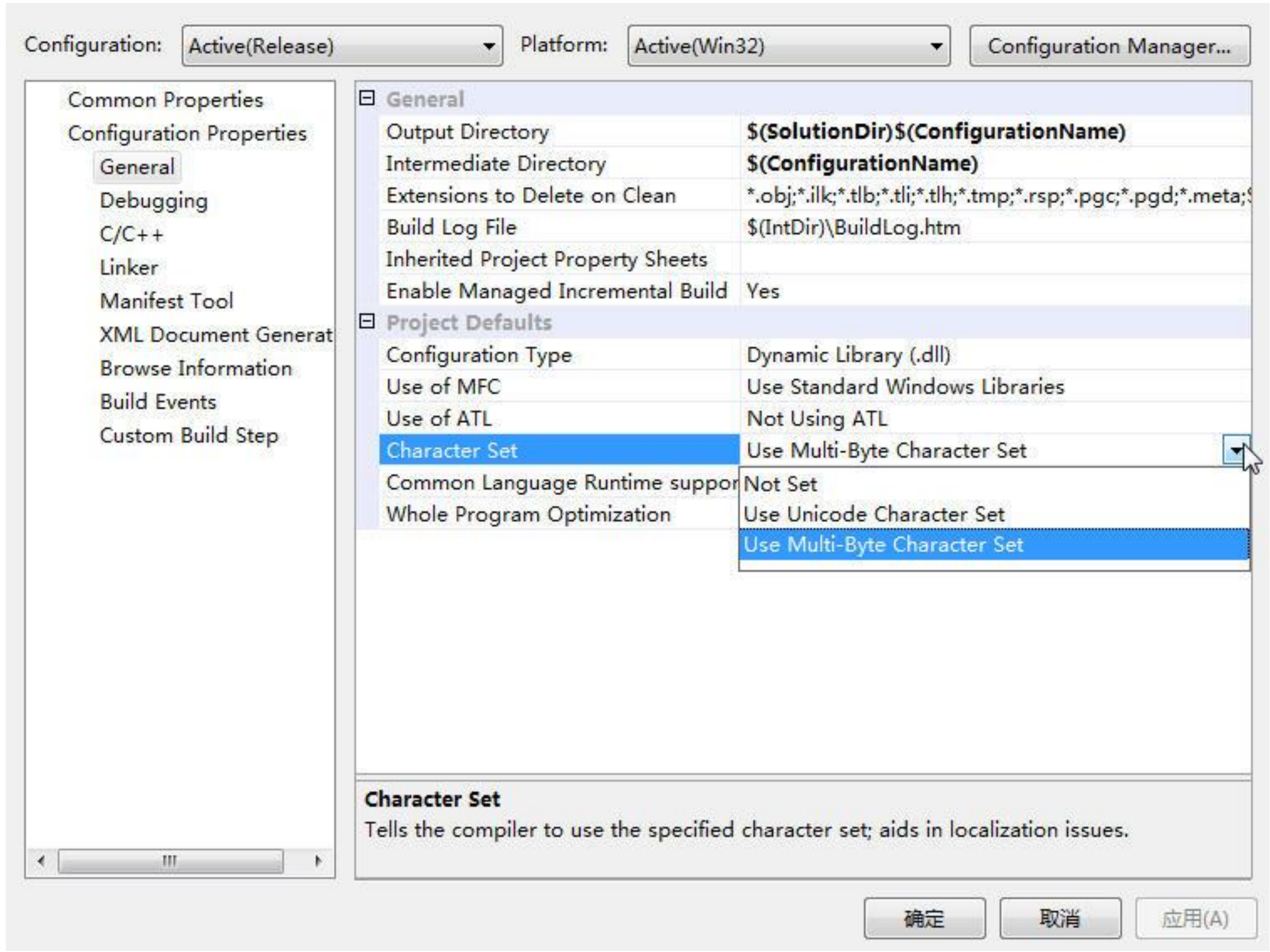


图 8-1 Visual Studio 设置 UNICODE/非 UNICODE

“Character Set”置为“Use Unicode Charcter Set”时会给项目加上预编译宏“UNICODE、_UNICODE”（置为“Use Multi-Byte Charcter Set”则预定义宏“_MBCS”）。这个“UNICODE”宏有什么作用呢？以下是摘自系统头文件 ShlObj.h 一处代码。

```
#ifdef UNICODE
#define SHGetFolderPath  SHGetFolderPathW
#else
#define SHGetFolderPath  SHGetFolderPathA
#endif // !UNICODE
```

一旦定义 UNICODE，即只要“Character Set”置为“Use Unicode Charcter Set”，程序中的 SHGetFolderPath 其实是调用 UNICODE 版本 SHGetFolderPathW，否则是 ANSI 版本 SHGetFolderPathA。SHGetFolderPathW、SHGetFolderPathA 的主要区别是它们返回的字符串一个是 UNICODE 编码，一个是 ANSI 编码（地区码）。

UNICODE 程序使用的字符编码很直观，就是 UNICODE 码。非 UNICODE 程序使用是 ANSI 编码，即地区码。

《王国战争》设置的“Character Set”是“Use Multi-Byte Charcter Set”，因而它是非 UNICODE 程序。调用的 API 都是 ANSI 版本。

哪里设置地区码

在任一时刻，操作系统只允许存在一种地区码。要察看、更改地区码，Win7 下是“控制面版”——“区域和语言”——“管理”

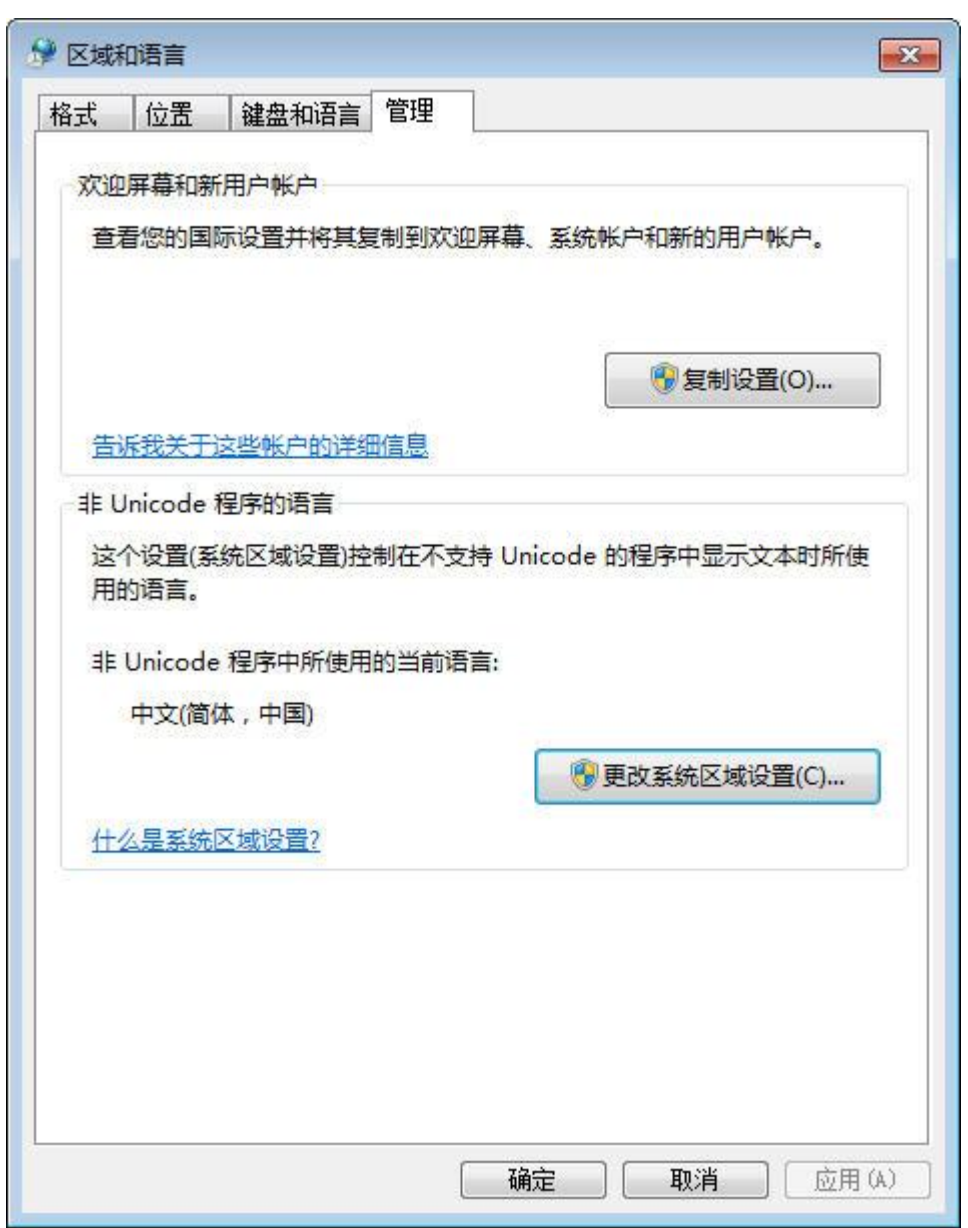


图 8-2 Win7 设置地区码

上图表示地区码是 GB2312（中文（简体，中国），“更改系统区域设置”按钮可以更改地区码。

地区码如何影响非 UNICODE 程序

为直观描述地区码如何影响非 UNICODE 程序，看一个调用 SHGetFolderPath 例子，让进入代码级调试。

1. 在 filesystem.cpp 的 set_preferences_dir 函数内设断点。

2. 运行时选“Debug”——“Starting Debug”。

地区码是 GB2312 时

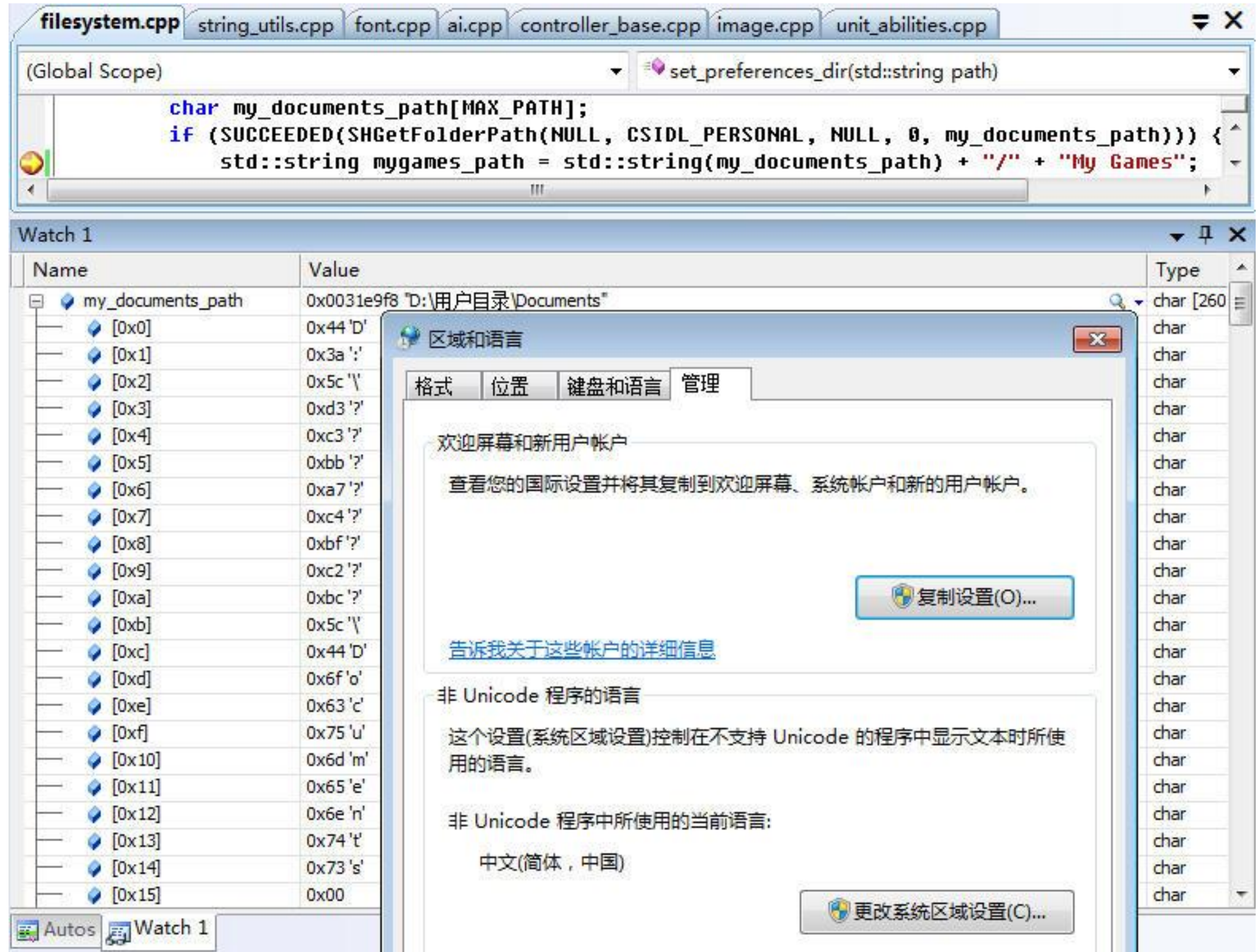


图 8-3 SHGetFolderPath-gb2312

此处执行的 SHGetFolderPath 返回本地“My Documents”目录，my_documents_path 变量存储了返回的名称，此 PC 上“My Documents”目录是“D:\用户目录\Documents”。

结合 GB2312 码表（http://ash.jp/code/cn/gb2312tbl.htm），小结下图 8-3。

- GB2312 以一个字节编码 ASCII 字符。
- GB2312 以两个字节编码汉字字符。D3C3：用；BBA7：户； C4BF：目；C2BC：录。

地区码是 BIG5 时

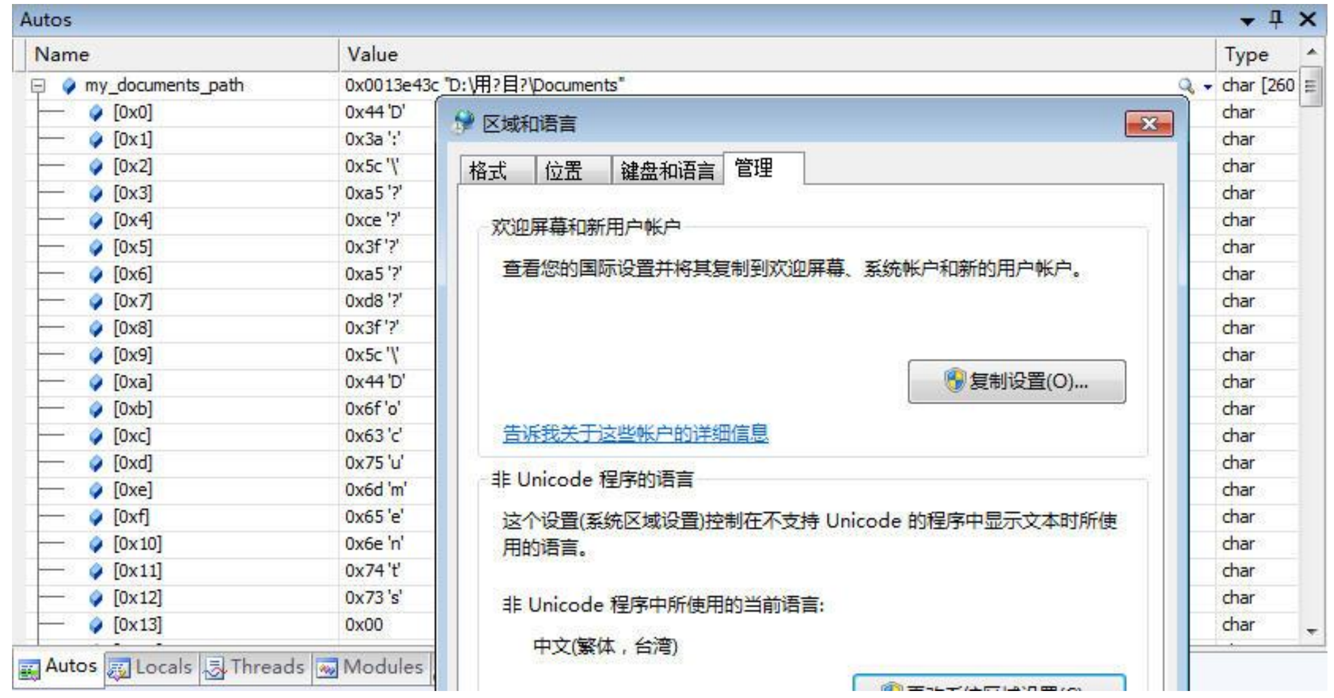


图 8-4 SHGetFolderPath-big5

结合 Big5 码表（http://ash.jp/code/cn/big5tbl.htm），小结下图 8-4。

- Big5 以一个字节编码 ASCII 字符。
- Big5 以两个字节编码汉字字符。A5CE：用；A5D8：目。
- “户”、“录”被置了“?”字符。这是由于 BIG5 中没有收录“户”、“录”字符，于是就强制被用 ASCII 字符“?”进行替代。

深入叙述下为什么会出现“?”。Win 7 内部已全是 UNICODE，也就是说文件系统管理磁盘，标注“D:\用户目录\Documents”这个字符串已采用 UNICODE 码。当应用程序调用 ANSI 版本的 SHGetFolderPath，系统发现上层要的是一个 ANSI 字符串，它反而须要做一个额外工作：把内部的 UNICODE 码字符转换成 ANSI 字符串！而 ANSI 字符串编码方式就是控制面板中设的地区码。具体转换“用户目录”这四个汉字，用，Big5 有收录，码值是 A5CE；户，Big5 没收录，于是就用一个 ASCII 字符“?”代替。让转去地区码是 GB2312 时，GB2312 有收录“用户目录”四个汉字，它由 UNICODE 转成 ANSI 时就不存在问题。

为说明更完全，让看 UNICODE 程序中 SHGetFolderPathW（不是 SHGetFolderPath）的返回值。结合汉字 UNICODE 码表（http://www.chi2ko.com/tool/CJK.htm）会发现当中全被用了 UNICODE 码。它正是 Win7 内部用于存储“D:\用户目录\Documents”这字符串的值。Win7 是个 UNICODE 操作系统，相比 SHGetFolderPathA，调用 SHGetFolderPathW 还不须要额外的 UNICODE 到 ANSI 之间转换开销。

Watch 1

Name	Value
my_documents_path	0x001ceae8 "D"
[0x0]	0x44 'D'
[0x1]	0x00
[0x2]	0x3a ':'
[0x3]	0x00
[0x4]	0x5c '\'
[0x5]	0x00
[0x6]	0x28 '('
[0x7]	0x75 'u'
[0x8]	0x37 '7'
[0x9]	0x62 'b'
[0xa]	0xee '?'
[0xb]	0x76 'v'
[0xc]	0x55 'U'
[0xd]	0x5f '_'
[0xe]	0x5c '\'
[0xf]	0x00
[0x10]	0x44 'D'
[0x11]	0x00
[0x12]	0x6f 'o'
[0x13]	0x00
[0x14]	0x63 'c'
[0x15]	0x00
[0x16]	0x75 'u'
[0x17]	0x00
[0x18]	0x6d 'm'
[0x19]	0x00

Autos Locals Threads Modules Watch 1

图 8-5 SHGetFolderPath-unicode

非 UNICODE 程序的死结

Microsoft 宣传 Windows 7 时使用“完全”支持 UNICODE，指的就是操作系统内部“完美”地用 UNICODE 码表示了“所有”字符串。所以虽然 BIG5 没有收录“户”、“录”字符，即使你还是硬把地区码设为 BIG5，操作系统自带的资源管理器却仍旧能很好地显示“D:\用户目录\Documents”这串字符，因为资源管理器是 UNICODE 程序，UNICODE 码表中是收录了“户”和“录”。

再来看非 UNICODE 程序，由于地区码少收录字符，操作系统叫它转换（从 UNICODE 到地区码）一些字符时它没有，那些没有的就被替换为“?”。很明显，接下程序如果继续用 my_documents_path 这个已是错了字符串时，那些访问都将出错。

非 UNICODE 程序的死结就是 PC 中存在此刻正使用的地区码未收录字符。既然收录字符就局限于地区，怎么能要求一台 PC 尽是“你”收录的字符呢。

要解决死结，UNICODE 程序是趋势。但很可惜，基于各式各样原因，像程序要跨平台，在条件不具备情况下又只能是非 UNIOCDE 程序。当然，既然知道非 UNICODE 程序和 UNICODE 程序区别就是有没有 UNICODE 宏，而有没有 UNICODE 宏主要影响的是同一个函数名调用的是 W 版本还是 A 版本，那在非 UNICODE 中可使用直接调用 W 版本来避免掉非 UNICODE 程序死结。这个直接调用是会让很烦，而且很难做到全面，但有时是不得已而使用的办法。

9.1.2 UNICODE 程序缺陷

以上可看出 UNICODE 程序不但支持国际化，还少去调用系统 API 时 UNICODE 字符和 ANSI 之间转换开销，似乎程序写成 UNICODE 就真的很完美。事实是怎样，让看一个例子。

1、修改 lua\luaxlib.c 的 luaL_loadfile 函数。

[list]

- 1.1：增加代码。

```
int wlen = MultiByteToWideChar(CP_UTF8, 0, filename, -1, NULL, 0);
char unicode_filename[MAX_PATH];
MultiByteToWideChar(CP_UTF8,  0,  filename,  -1,  (LPWSTR)unicode_filename,
wlen);
lf.f = fopen(unicode_filename, "r");
```

- 1.2：在 luaL_loadfile 前增加“#include <windows.h>”
- 1.3：注释掉“lf.f = fopen(filename, "r");”

2、在“if (lf.f == NULL)”处设断点。

3、运行时选“Debug”——“Starting Debug”。

4、标题屏幕选“战役”，一路“确认”，直到触发此个断点。

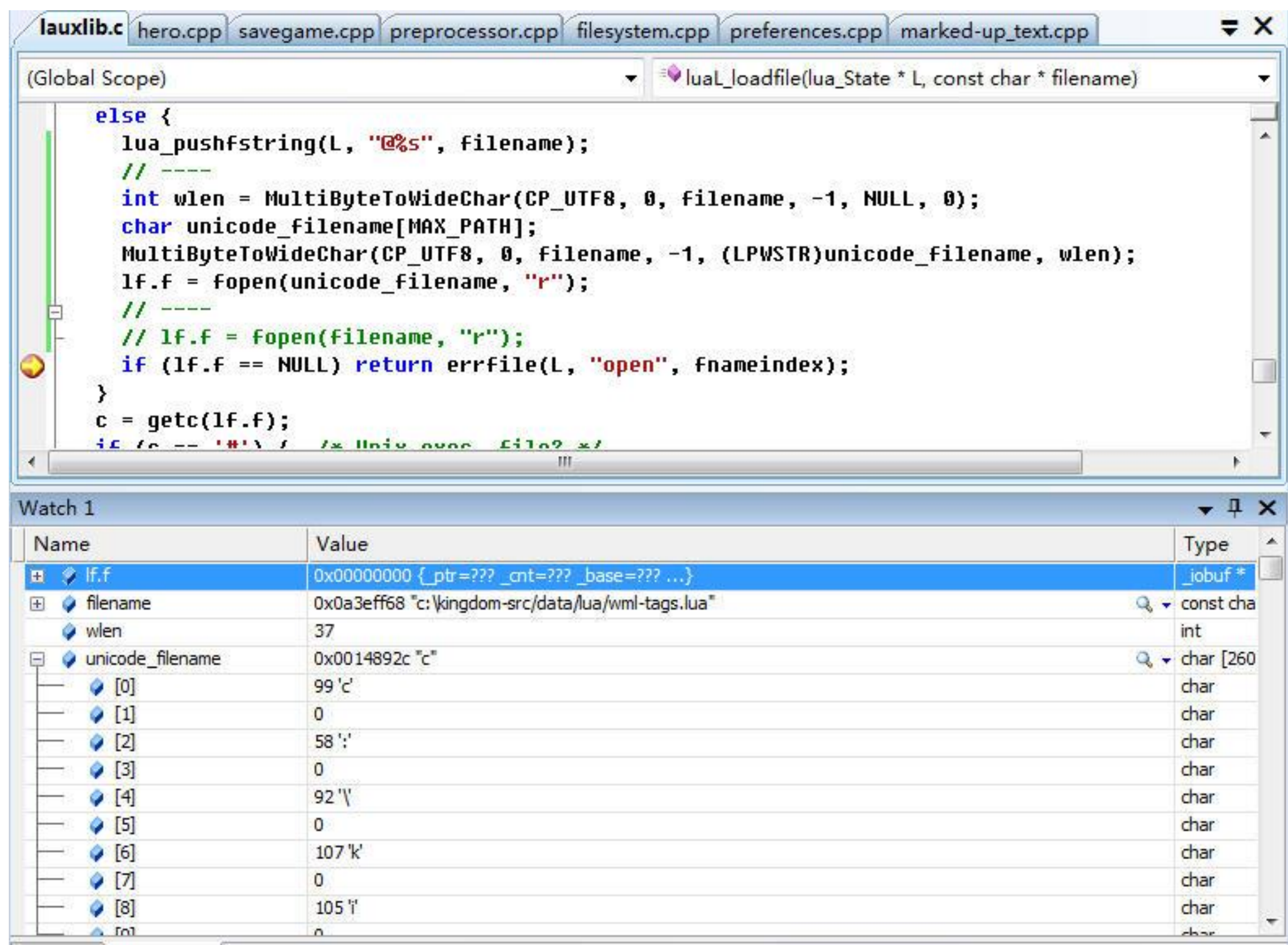


图 8-6 luaL_loadfile

修改代码目的是使传入 fopen 函数的文件名参数由 ANSI 字符串（单字节表示 ASCII 字符）变成 UNICODE 字符串（双字节表示 ASCII 字符）。在 Watch 窗口能看到 fopen 返回 NULL，fopen 失败。换句话说 fopen 不支持 UNICODE 字符串作为它的参数。

C 运行库中不仅仅是 fopen，其它涉及到字符作为参数的函数，像 remove，它们都不支持 UNICODE 字符串。

不可否认，C/C++经过数十年发展，C 运行库中函数大多已有其它也是跨平台的函数可替换，

像 STL 中的 std::ifstream 可替掉 fopen。但很多“历史悠久”程序，它们到现在依旧在用 C 运行库，而且 C 运行库非常轻量级，不涉及到任何 C++概念，是 STL 这些 C/C++库无法比拟的。

要让 C 运行库支持 UNICODE，就必须改 C 运行库。这个改法有两种。

- 保持原函数名，修改函数内部实现。
- 原函数名时功能不变，增加 UNICODE 版本函数名。

很显然，不管是兼容性还是跨平台考虑，第一种方案要优于第二种，但现实是 C 运行库在解决 UNICODE 问题上是采用第二种，至少 Microsoft 是这样。

TCHAR.H routine	_UNICODE & _MBCS not defined	_MBCS defined	_UNICODE defined
_tfopen	fopen	fopen	_wfopen

从以上可以看出 Microsoft 解决 ANSI/UNICODIE 基本思路：程序代码中不要用 fopen，改用_tfopen，调用_tfopen 前要“# include <tchar.h>”。对 UNICODE 程序，Microsoft C/C++会把_tfopen 映射到_wfopen，即 UNICODE 版本，非 UNICODE 程序则会映射到传统的 fopen，即 ANSI 版本。

Microsoft 这种改法对要用 C 运行库编写跨平台程序是个悲哀：代码中需要写_tfopen，可其它平台不认_tfopen！

在此不能抱怨 Microsoft 没做什么，它在为 C/C++发展做的贡献是谁也无法比的，况且对 C 运行库发展方向有它自已考虑，像提供原函数加“_s”后缀安全版本。让人欣慰的 Microsoft 在处理 STL 库是采用第一个方案，即同一个函数名既可以接受 UNICODE 参数也可以接受非 UNICODE（但不支持 UTF-8），函数内部对它们进行甄别。std::ifstream 打开的文件名既可以是 ANSI 也可以是 UNICODE。

知道 C 运行库处理 UNICODE 问题出在不接受 UNICODE 字符串，那会想到个办法，程序还是用 UNICODE，只是在传给 C 运行库函数时先转换成 ANSI！对此能不能不忘漏转是次要的，问题是这种方案就不可行。让考虑“8.1.1 Windows 下的 UNICODE、非 UNICODE 程序”中的“用户目录”，假如把地区码设到 Big5，它没收录“户”和“录”，那它由 UNICODE 转换为 ANSI 时出来的结果就是错的，fopen 自然失败。

要做到“完美”国际化，fopen 必须能接受一个统一码表示的字符串。当统一码是 UNICODE 时，至少当前 C 运行库是不支持，另外它还有存储效率低，像一定要用两字节表示一个 ASCII 字符，可高字节就没用的。那么是否存在用一个字节表示 ASCII 字符的统一码？——UTF-8。

UTF-8 是种统一码，以一个字节表示 ASCII 字符，但它也遇到 UNICODE 一样问题，fopen 不支持 UTF-8 字符串！程序中以 UTF-8 表示字符至少有一点强于 UNICODE，当文件名全是 ASCII 时，它不须要转换。编程时选择哪种方案需要权衡，既然运行库就不接受统一码，何不找个至少看去“好”点的统一码，UTF-8。

采用 UTF-8 下如何更好解决国际化问题。C 运行库不认 UTF-8，这不认是建立在参数是非 ASCII 之上，也就是说如果文件名都是 ASCII 字符，运行库处理 UTF-8 还是挺“完美”的。专业不行就用行政手段：强行要求用户在安装时把程序放在纯英文目录下！程序运行时需要数据一般分两个部分：资源包和用户数据。对 Win 系统，资源包一般放在“x:\Program Files”目录，用户数据则在<My Document>目录，对这两个目录，程序读取往往集中在资源包，用户目录往往就是存些配置数据而已。一旦资源包目录做到纯英文，UTF-8 加上 C 运行库就可毫无阻碍地去处理，至于<My Document>目录，它无法要求用户做到纯英文，这里就用 UNICODE，处理时也不要用 C 运行库，而是用支持 UNICODE 的 STL 或操作系统 API，像 CreateFileW。

9.1.3 UTF-8

UTF-8 不是独立编码系统，它基于 UNICODE。字面上 UTF-8 由 UTF 和 8 组成，UTF 是“Unicode Transformation Format”的缩写，可以翻译成通用转换格式，8 则是表示它以字节为单

位对 UNICODE 进行编码。

Unicode 到 UTF-8

Unicode 编码(16 进制)	UTF-8 字节流(二进制)
000000 - 00007F	0xxxxxxx
000080 - 0007FF	110xxxxx 10xxxxxx
000800 - 00FFFF	1110xxxx 10xxxxxx 10xxxxxx
010000 - 10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx[

UTF-8 的特点是对不同范围的字符使用不同长度的编码。对于 0x00-0x7F 之间的字符，UTF-8 编码与 ASCII 编码完全相同。UTF-8 编码的最大长度是 4 个字节。从上表可以看出，4 字节模板有 21 个 x，即可以容纳 21 位二进制数字。Unicode 的最大码位 0x10FFFF 也只有 21 位。

例 1："汉"字的 Unicode 编码是 0x6C49。0x6C49 在 0x0800-0xFFFF 之间，使用 3 字节模板：1110xxxx 10xxxxxx 10xxxxxx。将 0x6C49 写成二进制是：0110 1100 0100 1001，用这个比特流依次代替模板中的 x，得到：11100110 10110001 10001001，即 E6 B1 89。

例 2：Unicode 编码 0x20C30 在 0x010000-0x10FFFF 之间，使用用 4 字节模板：11110xxx 10xxxxxx 10xxxxxx 10xxxxxx。将 0x20C30 写成 21 位二进制数字（不足 21 位就在前面补 0）：0 0010 0000 1100 0011 0000，用这个比特流依次代替模板中的 x，得到：11110000 10100000 10110000 10110000，即 F0 A0 B0 B0。

UTF-8 迭代器

以 UTF-8 处理程序中字符，文件路径是一个方面，更多还是遇到的杂七杂八字符，像从一个文本文件读出的字符流。要处理字符流时，首先要能分隔出一个一个字符，对 UNICODE 两字节一个字符，但 UTF-8 可能 1 字节，也可能 2、3、4 字节，这时要设计出一个好的 UTF-8 迭代器。

UTF-8 迭代器类似 STL 迭代器，支持 begin()、end()、operator++，operator*。假设要处理字符流是 std::string utf8_text，以下是 UTF-8 迭代器通用调用逻辑。

```
utils::utf8_iterator ch(utf8_text)
for (utils::utf8_iterator end = utils::utf8_iterator::end(utf8_text); ch != end; ++ch) {
    *ch 读出是以 UNICODE 码表示的当前字符
}
```

经过 UTF-8 迭代器读出的已是程序需要的 UNICODE 字符，对数据处理来说，UNICODE 才是需要的结果值。为更进一步了解 UTF-8 迭代器，让深入 utf8_iterator::update 函数（begin()、operator++都会调用它），同时学会用 C/C++代码把 UTF-8 转换成 UNICODE。

1、修改 serialization\string_utils.c 的 utf8_iterator::update()函数，增加代码。

```
std::string::const_iterator my_itor = current_substr.first + 1;
unsigned char ch1 = *my_itor;
my_itor ++;
unsigned char ch2 = *my_itor;
```

2、在"if (size != 1)"处设断点。

3、运行时选"Debug"——"Starting Debug"。

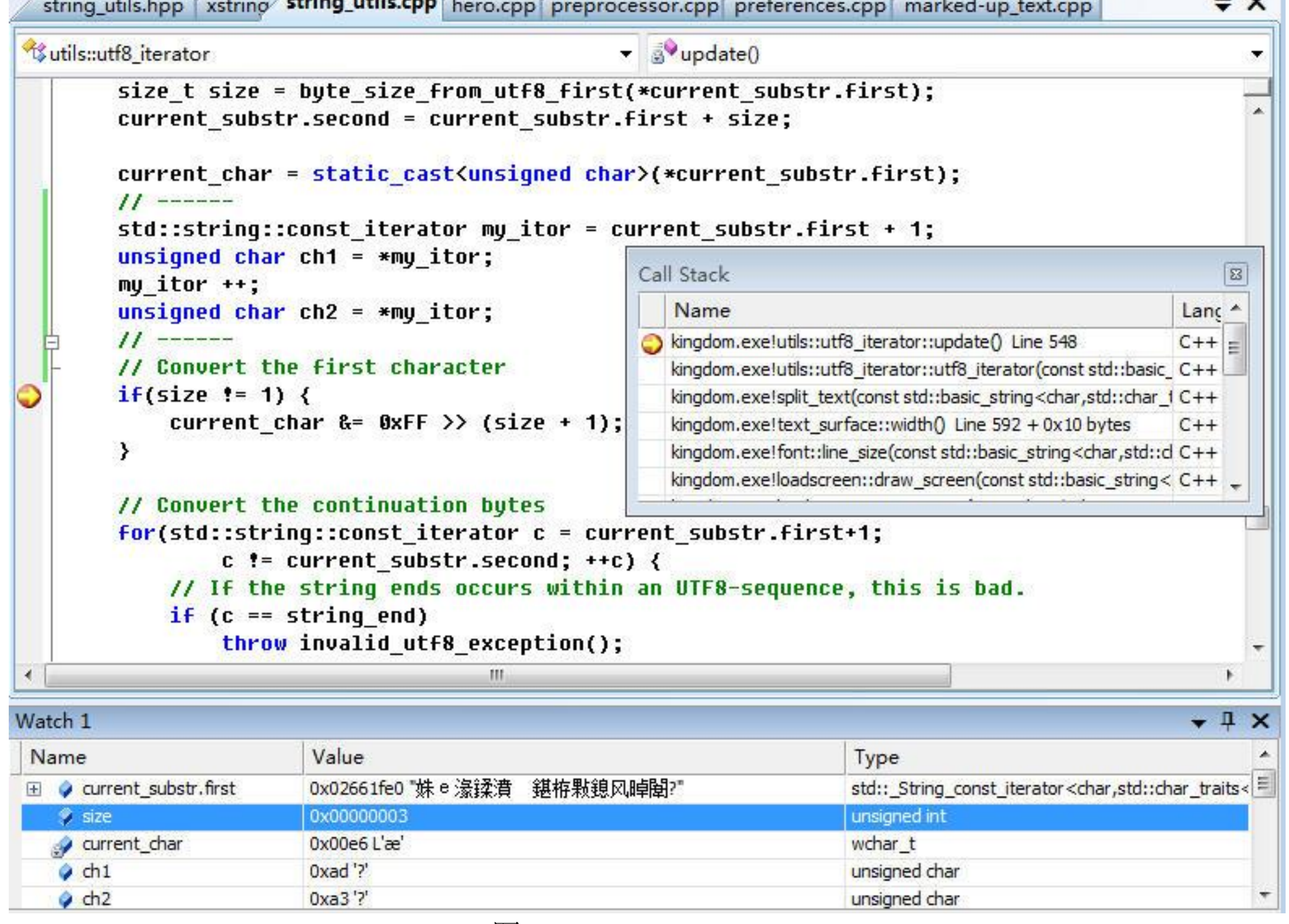


图 8-7 utils::utf8_iterator

增加代码目的是为更直观看到字节流（UTF-8 字符流）中正要处理的第二、第三字节，已存的 current_char 指示的则是正要处理的第一个字节。

由图 8-7，结合 utf8_iterator::update()可总结出 UTF-8 转换成 UNICODE 逻辑。

步骤一：得到此个 UTF-8 字符要占用字节数

byte_size_from_utf8_first 实现这个操作，依据 UTF-8 编码特点，依据第一个字节就能得到此个字符的字节数。

```
static int byte_size_from_utf8_first(unsigned char ch)
{
    int count;

    if ((ch & 0x80) == 0)
        count = 1;
    else if ((ch & 0xE0) == 0xC0)
        count = 2;
    else if ((ch & 0xF0) == 0xE0)
        count = 3;
    else if ((ch & 0xF8) == 0xF0)
        count = 4;
    else if ((ch & 0xFC) == 0xF8)
        count = 5;
    else if ((ch & 0xFE) == 0xFC)
        count = 6;
    else
        throw invalid_utf8_exception(); // Stop on invalid characters

    return count;
}
```

此个例子中第一个字节是 0xe6，“(ch & 0xF0) == 0xE0”，得到字节数是 3。

步骤二：转换出第一个字节

```
if (size != 1) {
    current_char &= 0xFF >> (size + 1);
}
```

此个例子转换出第一个字节参与值是 0x6。

步骤三：转换出后续字节，由移位操作总合出以 UNICODE 码表示的字符

```
for (std::string::const_iterator c = current_substr.first+1;
        c != current_substr.second; ++c) {
    // If the string ends occurs within an UTF8-sequence, this is bad.
    if (c == string_end)
        throw invalid_utf8_exception();

    if ((*c & 0xC0) != 0x80)
        throw invalid_utf8_exception();

    current_char = (current_char << 6) | (static_cast<unsigned char>(*c) &
0x3F);
}
```

字节流中值（十六进制）	e6	ad	a3
抽出的有效部分（二进制）	0110	10 1101	10 0011

移位后得到的 UNICODE 码：6b63；查 UNICODE 码表 0x6b63 指向“正”这个汉字。

作为参考，再贴个一次性就把一 UTF-8 字符串转成 UNICODE 函数。摘自 SDL_ttf 库的 SDL_ttf.c。

```
static Uint16 *UTF8_to_UNICODE(Uint16 *unicode, const char *utf8, int len)
{
    int i, j;
    Uint16 ch;

    for ( i=0, j=0; i < len; ++i, ++j ) {
        ch = ((const unsigned char *)utf8)[i];
        if ( ch >= 0xF0 ) {
            ch  =  (Uint16)(utf8[i]&0x07) << 18;
            ch |=  (Uint16)(utf8[++i]&0x3F) << 12;
            ch |=  (Uint16)(utf8[++i]&0x3F) << 6;
            ch |=  (Uint16)(utf8[++i]&0x3F);
        } else
        if ( ch >= 0xE0 ) {
            ch  =  (Uint16)(utf8[i]&0x0F) << 12;
            ch |=  (Uint16)(utf8[++i]&0x3F) << 6;
            ch |=  (Uint16)(utf8[++i]&0x3F);
        } else
        if ( ch >= 0xC0 ) {
            ch  =  (Uint16)(utf8[i]&0x1F) << 6;
            ch |=  (Uint16)(utf8[++i]&0x3F);
        }
        unicode[j] = ch;
    }
    unicode[j] = 0;

    return unicode;
}
```

字库

多个字模组成字库，每个字模占等量字节，要显示时以“地址=码值 x 每字模占用字节数”来寻址字库中特定字模。UTF-8 是非顺序编码，当中存在好多“空洞“，不存在什么 UTF-8 字库。UTF-8 基于的 UNICODE 则是顺序编码，要显示 UTF-8 字符要先转换成 UNICODE 码，然后以后者寻址 UNICODE 字库。用于显示字符串的 SDL_ttf.dll 正是按此个逻辑，因而以 UTF-8

为参数的 TTF_RenderUTF8_Blended 执行效率低于以 UNICODE 为参数 TTF_RenderUNICODE_Blended，内部实现上，前者通过先把参数转成 UNICODE 再调用后者来实现渲染。

9.2 gettext

gettext 是个 GNU 开源项目，它使用 locale 设计出一种可让一个程序包“存储”并切换多语言的机制。在继续之前请最好粗略回顾下“1.3.1 要翻译字符串”。

要翻译字符串：根据当前语言不同会显示出不一样结果的字符串。要翻译字符串可存在两个地方，WML 代码中、C/C++代码中。WML 指定一个字符串是要翻译字符串的方法是在字符串前加个“_”字符，像 id=_"Campaign"。C/C++代码指定一个字符串是要翻译字符串是在字符串外加“_()”，像 str << _("Local Player")。在叫法上，把 WML 中的加上“_”前缀、C/C++中加上“_()”字符串叫做别名，_"Campaign"是要翻译字符串 Camapign 的别名。

9.2.1 gettext 翻译流程

步骤一：生成*.pot 文件

.pot 文件是程序员产生的文件，是源代码中要翻译字符串的模板，它汇集程序要用到的所有要翻译字符串。.pot 是文本格式文件，常用文本编辑器都可以创建、编辑它。打开<kingdom-res>\po\wesnoth-lib\wesnoth-lib.pot，看下*.pot 文件中内容。

```
......
#. [grid]
#: data/gui/default/window/title_screen.cfg:224
msgid "Campaign"
msgstr ""

#. [grid]
#: data/gui/default/window/title_screen.cfg:224
msgid "Start a new single player campaign"
msgstr ""
......
```

- “#”字符开始的是注释行。
- msgid：指示此个要翻译字符串。字符串值就是写 WML 代码、C/C++代码别名中的值。
- msgstr：此处一律留空。

*.pot 就是以(msgid, msgstr)格式，一个一个地把要翻译字符串集中到一块。

<kingdom-res>\po 目录存在多个 pot，这是因为程序用到要翻译字符串太多，主观把它进行分类，像主题中要翻译字符串放在 wesnoth.pot，窗口中要翻译字符串放在 wesnoth-lib.pot，多人对战则放在 wesnoth-multiplayer.pot。分开多个 pot 有利于维护，同时因为关键字少了，有利于提高程序运行时匹配效率。

程序发布时不必包含*.pot 文件。

步骤二：生成*.po 文件

.po 是翻译员需要产生的文件。.pot 和特定语言无关，*.po 特定于某种语言。*.po 是文本格式文件，常用文本编辑器都可以创建、编辑它，但为统一等原因，推荐用专门 po 编辑器，像 Poedit(http://www.poedit.net)，poedit 很小，也是开源软件。进入<kingdom-res>\po\wesnoth-lib 目录，让对比下 en_GB.po、zh_CN.po、zh_TW.po 文件内容。

<kingdom-res>\po\wesnoth-lib\en_GB.po

```
......
#. [grid]
#: data/gui/default/window/title_screen.cfg:224
```

```
msgid "Campaign"
msgstr "Campaign"

#. [grid]
#: data/gui/default/window/title_screen.cfg:224
msgid "Start a new single player campaign"
msgstr ""
......
```

<kingdom-res>\po\wesnoth-lib\zh_CN.po

```
#. [grid]
#: data/gui/default/window/title_screen.cfg:224
msgid "Campaign"
msgstr "战役"

#. [grid]
#: data/gui/default/window/title_screen.cfg:224
msgid "Start a new single player campaign"
msgstr ""
```

<kingdom-res>\po\wesnoth-lib\zh_TW.po

```
#. [grid]
#: data/gui/default/window/title_screen.cfg:224
msgid "Campaign"
msgstr "戰役"

#. [grid]
#: data/gui/default/window/title_screen.cfg:224
msgid "Start a new single player campaign"
msgstr ""
```

- “#”字符开始的是注释行。
- msgid：指示此个要翻译字符串。字符串值就是写 WML 代码、C/C++代码别名中的值。
- msgstr：特定于该语言翻译出的字符串。若置空则意味着翻译出字符串就是 msgid 中内容。

对比*.pot、*.po，会发现*.pot 说白了就是 msgstr 一律置空的*.po！事实的确如此，gettext 要实现翻译是可以不需要*.pot。那为什么要存在*.pot，——维护需要。可以想象下，一个人要完成一种语言翻译或许还有可能，要实现全语种翻译，那可说“绝对”不可能。每种语言翻译员就要得到一份内容相同的要翻译字符串集，如何保证这个相同，靠的就是给它们同一个*.pot。基于*.pot 作用就是保证各翻译员得到的是同一份要翻译字符串集，它常被称为翻译模板。

让尝试在 Poedit 使用翻译模板，同时把标题屏幕中“战役”按钮改为显示“我的战役”。

1、文本编辑器打开<kingdom-res>\po\wesnoth-lib\zh_CN.po，删除“Campaign”这个要翻译字符串，即以下这两行。

```
msgid "Campaign"
msgstr "战役"
```

2、保存，关闭 zh_CN.po。Poedit 打开 zh_CN.po，进入主界面后执行“类目”——“从 pot 文件更新...”，选择 wesnoth-lib.pot。Poedit 会弹出“更新摘要”对话框，“新建字符串”页指示已存在的 zh_CN.po 缺少“Campaign”这个要翻译字符串。同样，“过时字符串”页会指示 zh_CN.po 有、wesnoth-lib.po 没有的要翻译字符串。

3、按“确认”关闭“更新摘要”对话框。“原文”中找到“Campaign”，在底下翻译区输入“我的战役”。按“保存”，修改会被存入 zh_CN.po，同时生成 zh_CN.mo。

程序发布时不必包含*.po 文件。

步骤三：生成*.mo 文件

.mo 是程序运行时“唯一”依赖的语言包文件。它由.po 一对一生成。

针对每个 pot，每种语言各有一个 po，基于这种结构大概可猜出 gettext 是如何翻译字符串。gettext 内部有个存储当前操作系统正使用语言的字符串变量 locale，英文时 locale=en_GB，这时 gettext 就以 msgid 搜 en_GB.po，得到的 msgstr 就是英文下要显示的内容，简体中文时

locale=zh_CN，于是搜 zh_CN.po，繁体中文则搜 zh_TW.po。既然*.po 已可以实现切换语言，似乎语言资源包做到*.po 这一层就可以了，但为效率考虑只到*.po 是不够的。

- 要优化匹配等操作。为支持翻译，运行时最多的是字符串匹配操作，*.po 没对这种操作做任何优化，就是简单地把 msgid 内容按字母排下序，对搜索也是极大优化。
- *.po 存在大量注释。注释不仅浪费内存还降低搜索效率。
- *.po 是文本格式。文本格式易于编辑，但不适合发布。

正因为*.po 对运行时翻译存在诸多不足，gettext 机制要求做语言资源时还要把 po 生成 mo，mo 是个优化的、更适合搜索操作的 po。mo 和 po 一一对应。

“步骤二：生成*.po 文件”已叙述过如何用 Poedit 生成 mo，接下让看如何使用 mo。把生成的<kingdom-res>\po\wesnoth-lib\zh_CN.mo 改名为 wesnoth-lib.mo，并复制到<kingdom-res>\translations\zh_CN\LC_MESSAGES，覆盖掉 LC_MESSAGES 下原 wesnoth-lib.mo。再运行游戏程序，语言设为简体中文，进入标题屏幕后“战役”按钮应显示为“我的战役”。

程序发布时须要包含*.mo 文件。在存放上，gettext 对“众多”*.mo 有一定存放要求。

图 8-9 mo

- kingdom-res/translations 是所有 mo 包的根目录，translations 这个目录名由应用程序自定义。
- <translations>根下是按语种分类目录，en_GB、zh_CN、zh_TW 这些叫法是内定的，来自 locale。
- 语种目录根下是 LC_MESSAGES 目录，LC_MESSAGES 是个内定目录名。
- en_GB\LC_MESSAGES 存放所有属于英文的 mo 文件，同样，zh_CN\LC_MESSAGES 存放所有属于简体中文的 mo 文件。
- mo 文件扩展名必须是 mo，主文件名由应用程序自定义，每个主文件名对应一个域名。

pot、po、mo 小结

	pot	po	Mo
编写人	WML、C/C++程序员	语言翻译员	语言翻译/发布员
格式	文本	文本	二进制

内容来源	MOD、C/C++程序	来自*.pot	来自*.po
编辑工具	常用文本编辑器	Poedit	Poedit
包含于发布包	否	否	是

9.2.2 xgettext 和自动生成 pot

xgettext 是 gettext 系统提供的一个实用程序，用它去收集 msgid，进而生成 pot。它的工作原理是根据给定的关键字（keyword）和 msgid 位置对，搜索源码文件。举个例子，关键字是 gettext、位置是 1，就可收集出用 gettext 翻译的 msgid；关键字是 dsgettet、位置 2 则是那些用 dsgettet 翻译的字符串。在自动生成 pot 上，Rose 也采用 xgettext，但由于要支持库和应用相分离、以及一个应用可能存在多个翻译域，限定了代码要如何调用翻译函数。

“库和应用相分离”指的是 Rose 库使用“rose-lib”，基于它的应用使用“<appid>-lib”，假设应用 appid 是 studio，那么它的默认翻译域是“studio-lib”。“一个应用可能存在多个翻译域”指应用层除默认域外还存在其它域，比如战棋游戏，为清晰把武将姓名、列传单独放在不是默认的“hero.pot”。不论哪一种，都会导致应用存在多个翻译域，为支持多个采用以下规则。

- 默认翻译域是“<appid>-lib”。
- 不使用默认翻译域的源文件要定义“GETTEXT_DOMAIN”宏，宏值是翻译域。
- 翻译函数统一使用“_”或“vgettext2”。

按照以上规则，“rose-lib”不可能是默认翻译域，所以 rose 库中源文件都有预定义 GETTEXT_DOMAIN。为更了解以上规则，让深入 poedit 如何自动收集要翻译字符串。

Rose 使用 xgettext 生成 pot，实现它是通过 poedit，正是后者调用 xgettext 来自动收集要翻译字符串。poedit 的收集过程要设置两个参数，一是要搜索的路径，二是要搜索的关健字及位置。在菜单操作上，它们都是“编目”——“属性”，弹出“编目属性”对话框，前者对应“源路径”页，后者是“源关健字”。

源路径。源路径一般为两类，一是 C/C++源代码所在路径，二是存放 cfg 涉及到的、以及其它自定义路径，它固定使用<res>/po/cfg-cpp/<appid-lib>。以 rose-lib.pot 为例，C/C++源代码所在路径是<src>/librose，存放 cfg 涉及到的、以及其它自定义路径是<res>/po/cfg-cpp/rose-lib。第二种路径中文件分为两部分，一是由编辑器把*.cfg 汇编成某个 xwml/*.bin 时自动生成的文件，像 rose-lib 会来自 data.bin、gui.bin，它就自动会生成 data.cpp、gui.cpp。二是源码、*.cfg 都搜不出的 msgid，举个例子，有些武将姓名的 msgid 既不出现在源码也不出现在*.cfg，那就集中放在一个 cpp，命名为_bonus.cpp，然后额外写上“_”调用让 xgettext 发现这些 msgid。

源关健字。此处设置关健字（keyword）和位置对，要注意的是，Rose 不建议使用 gettext、dsgettext，而是以下两个关键字，位置都是“1”。

- _：翻译不带参数的 msgid。例子：_("msgid")。
- vgettext2：翻译带参数的 msgid。例子：vgettext2("msgid $p", symbols)。

9.2.3 API

gettext 被封装成 dll，沿袭一惯命名规则，gettext 生成的 dll 叫 intl.dll，intl 是 international 缩写。接下分析应用程序如何使用 gettext 提供的 API。以下是一段使用 gettext 代码。

```
#include <string>
#include <clocale>
#include <libintl.h>

int main(int argc, char* argv[])
{
    setlocale(LC_ALL, "");

    bindtextdomain("wesnoth", "C:\kingdom-res\translations");
    bind_textdomain_codeset("wesnoth", "UTF-8");
    bindtextdomain("wesnoth-lib", "C:\kingdom-res\translations");
    bind_textdomain_codeset("wesnoth-lib", "UTF-8");
    bindtextdomain("wesnoth-hero", "C:\kingdom-res\translations");
```

```
    bind_textdomain_codeset("wesnoth-hero", "UTF-8");
    textdomain("wesnoth-lib");

    std::string tstr = gettext("Campaign");
    return 0;
}
```

域

gettext 是以<msgid，msgstr>这样一个映射实现翻译，多个映射项被放在一个 mo 文件中，这一个 mo 文件就称为域。bindtextdomain("wesnoth-lib", "C:\kingdom-res\translations")功能是创建一个域：wesnoth-lib，同时把“C:\kingdom-res\translations”下的文件 wesnoth-lib.mo 绑定到这个域。wesnoth-lib.mo 文件名是由 gettext 内嵌规则得出，命令规则是“域名+.mo”。

把映射项依主观定的规则分类存放在多个文件无疑是有好处的。一来可以维护上模块化，二来程序执行时加快翻译（在那一 mo 文件中映射项少，搜索快了）。这使得一个程序可能存在多个域，也就是多个文件，例如游戏中全局信息，像“坐标”、“点击鼠标”这样一些操作信息是放在 wesnoth.mo 文件中，而像兵种信息，像“骑兵”、“市场”则放在 wesnoth-units.mo。

既然存在多个域，程序在翻译具体某一 msgid 时就要有“当前域”概念。textdomain("wesnoth-lib")功能把 wesnoth-lib 设当前域，即指示接下要翻译 msgid 时都默认到 wesnoth-lib.mo 中找。

翻译

gettext()实现翻译动作。以上 gettext("Campaign")执行以 msgid=Campaign 为关键字在 wesnoth-lib.mo 中找查。

gettext()参数中没有域，所以它在当前域中查找。除了 gettext()，gettext 另外提供了一个可以给出域参数版本函数：dgettext。在一些场合 dgettext 要比 gettext()方便，例如程序中可能大量使用 wesnoth-lib.mo，偶尔会用下 wesnoth-units.mo，这时就可用 dgettext("wesnoth-units", <msgid>)，dgettext 退出前会自动把当前域设到它进入之前的当前域。

选择输出字符集

gettext 要的输入是 ANSI 字符串（msgid 值都是 ASCII 字符），输出却可以指定各式各样。bind_textdomain_codeset("wesnoth-lib", "UTF-8")用于指定域 wesnoth-lib 输出的是 UTF-8 字符集。

当没有为一个域调用过 bind_textdomain_codeset 时，输出字符集是 ANSI。

gettext()没有在当前域中搜到 msgid 对应的 msgstr 时，输出字符集强制是 ANSI。举个例子，msgid 是 Campaign，但在 wesnoth-lib.mo 中没有以 Campaign 为 msgid 的映射，这时 gettext 返回的不是空字符串而是 Campaign，而且 Campaign 是 ANSI 字符集，即使在 wesnoth 域上已经调用过 bind_textdomain_codeset("wesnoth-lib", "UTF-8")。

FAQ：gettext 最多支持多少个域

bindtextdomain 用于创建域，并把<translations>\<locale>\LC_MESSAGES 目录下 mo 文件绑定到这个域。当程序中有多个域要被“一块”（任一时刻当然只能是一个，“一块”指的是一段时内多个域轮换）使用时，gettext 要换域时并不须重新调用一次 bindtextdomain，也就是说 gettext 对已注册的域有记忆功能。那么能记住域数目有多大？记忆时间有多长？

要知道这答案得知道 gettext 是如何管理域。让进入 gettext 库，用设置断点等方式理解 bindtextdomain 是如何工作的。

1. 在 kingdom 工程中打开 gettext 工程中的 bindtextdom.c。
2. 在 set_binding_values 函数内设断点。
3. 运行时选“Debug”——“Starting Debug”。

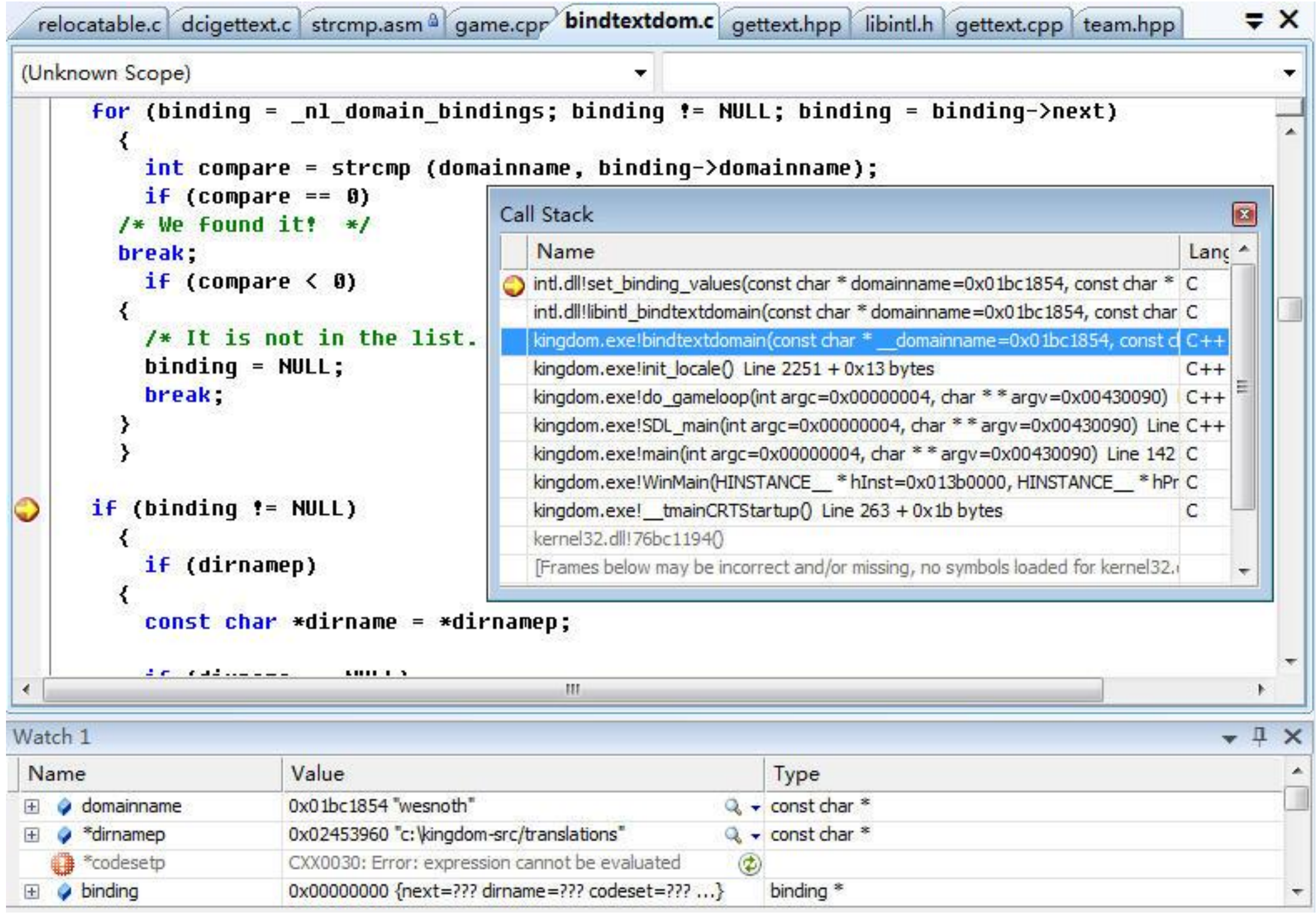


图 8-10 set_binding_values

从 Call Stack 可看到 bindtextdomain 调用 set_binding_values 来实现创建域、绑定 mo 目录；不仅 bindtextdomain，bind_textdomain_codeset 也是靠调用 set_binding_values 来实现设置输出字符集。

```
static void set_binding_values(const char *domainname, const char **dirnamep,
const char **codesetp)
```

- doaminname：要创建或修改的域的名称。例如 wesnoth。
- dirnamep：域目录。例如 c:\kingdom-res/translations。bind_textdomain_codeset(...)时这参数传 NULL。
- codesetp：域输出字符集。例如 UTF-8。bindtextdomain(...)时这参数传 NULL。

gettext 如何管理域？图 8-9 中有个看似熟悉的 for 循环。

```
for (binding = _nl_domain_bindings; binding != NULL; binding = binding->next)
```

这循环直觉会猜去是实现逐个访问以_nl_domain_bindings 为头的链表中结点，bingding 指示读出的结点，结点中有的 next 变量指向下一个节点。——没错，此个 for 循环就是实现访问链表中结点，此链表就是 gettext 管理域的数据结构，链表结点的 C/C++类型是 struct binding。

```
struct binding {
    struct binding *next;
    char *dirname;
    char *codeset;
    char domainname[ZERO];
};
摘自<gettext>/gettext/gettext-runtime/intl/gettextP.h

struct binding *_nl_domain_bindings;
摘自<gettext>/gettext/gettext-runtime/intl/dcigettext.c
```

结点没有 prev 字段，应该是个单向链表。域链表除去单向这特征外，继续看

set_binding_values(...)，当新域名没在链表中、set_binding_values 要把新创建域（new_binding）链接进链表用的是以下代码。

```
if (_nl_domain_bindings == NULL
    || strcmp (domainname, _nl_domain_bindings->domainname) < 0) {
    new_binding->next = _nl_domain_bindings;
    _nl_domain_bindings = new_binding;
} else {
    binding = _nl_domain_bindings;
    while (binding->next != NULL
        && strcmp (domainname, binding->next->domainname) > 0)
    binding = binding->next;

    new_binding->next = binding->next;
    binding->next = new_binding;
}
```

由代码可看出新节点插入链表时它被按域名进行了排序，而_nl_domain_bindings 这个全局变量“永远”指向链表头。由于_nl_domain_bindings 初始值是 NULL，以上代码同时也解决链表中最后一个结点的 next 字段赋 NULL 值。

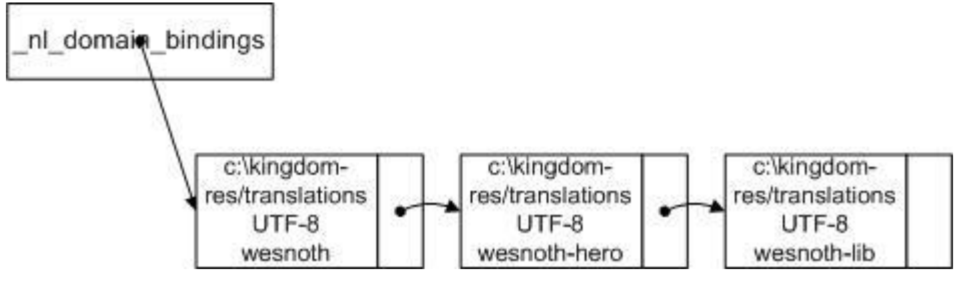


图 8-11 域链表

图 8-11 是依据以上规则形成、存放了三个域的链表，三个域的输出字符集都是 UTF-8，语言包路径都是“c:\kingdom-res\translations”。

综上所述，gettext 用单向链表来管理域，每个节点表示一个域。链表中节点被按域名升序存放，支持的域数目理论上无限个，一旦创建它就一直有效。

FAQ：带参数别名

让考虑这么种情况，你正在写个三国回合策略游戏，游戏中有数个势力，在每一回合轮到某个势力行动时要给玩家个提示，“轮到马超的回合了”，“轮到刘备的回合了”，“轮到曹操的回合了”。三个要翻译字符串除了势力名不一样，基它都是一样，要实现这个翻译，可以写三个 msgid，可这种方法既费时又很难难做到全面（能把所有势力名都包括了？），这时就可以写个带参数的可翻译字符串，C/C++代码表现出来就是个带参数别名。

如何实现带参数别名，让进入代码级调试。

1、打开 src/playsingle_controller.cpp，在 playsingle_controller::show_turn_dialog 内设断点。

2、运行时选“Debug”——“Start Without Debuging”。进入标题屏幕后，“首选项”——“常规”，打钩“回合对话框”。回到标题屏幕，语言选择英文（让 Watch 窗口能清晰看到翻译出的是什么字符串）。

3、“Debug”——“Attach to Process”，选择 kingdom.exe 进程。回到标题屏幕后，“战役”，一路“确认”直到触发断点。

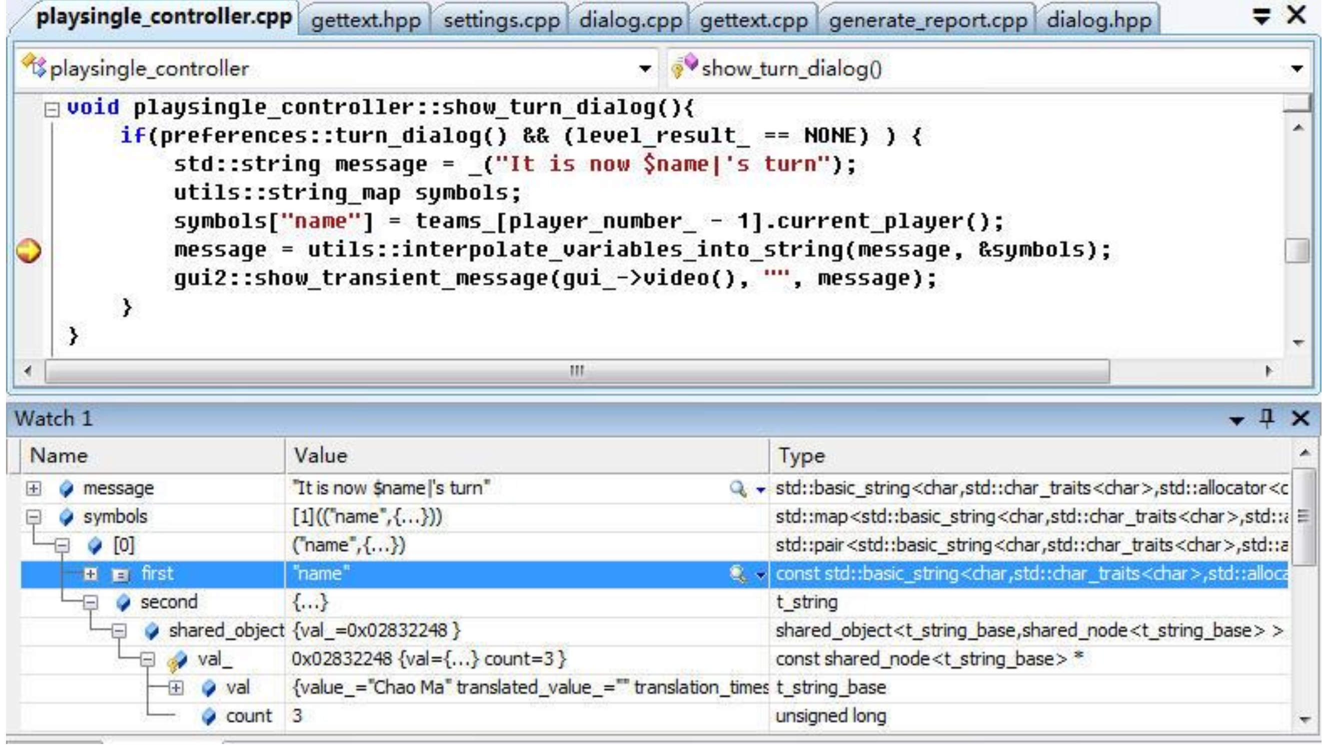


图 8-12 show_turn_dialog

图 8-12 中 Watch 窗口看到 message 变量值是“It is now $name|'s turn”，但只要一执行完“message = utils::interpolate_variables_into_string(message, &symbols)”，message 就会改为“It is now Chao Ma's turn”，也就是说原来的“$name|”被改成了 C/C++代码根据当前程序运行状态得到势力名（symbols[0].second）：Chao Ma。依据图 8-12，大概可得出 C/C++是如可实现带参数的别名。

1. 对要翻译字符串 msgid 调用 gettext API 得到 msgstr。用到的翻译函数和不带参数别名没区别，还是 gettext/dgettext。
2. 填充 utils::string_map symbols。symbols 项的 first 是参数名，second 是 C/C++代码根据当前运行状态得到的、参数对应的值。如果带参数别名中有 N 个参数，symbols 就需要 N 项。
3. 调用 utils::interpolate_variables_into_string(message, &symbols)，依据 symbols 以值替换掉 message 中所有参数。

为支持以上逻辑，po 在翻译时至少要原封不动保留 pot 中的参数名。以下是和此个要翻译字符串相关的是 zh_CN.po 部分。

```
msgid "It is now $name|'s turn"
msgstr "轮到$name|的回合了"
```

“|”符号用于清晰指示此个参数名结束，具体过程参考 utils::interpolate_variables_into_string。

对于要格式化带参数字符串，有人会想到用 sprintf。

```
sprintf(msgstr, gettext("It is now %s's turn"), "Chao Ma")
```

这语句乍看似方便，但没法实用，sprintf 存在和 fopen 同样问题，只接受 ANSI 字符串，像地区码是 Big5 时就不能正常格式化 Big5 没有收录的字符，像汉字中“户”，“录”。

综上所述，gettext 没给出带参数别名的专门翻译函数，要靠应用程序自个去实现。可采用约定以特殊符号（$）为前缀的是参数，运行时用<参数，值>这样配对以“值”替换掉字符串中的“参数”。

API 小结

gettext 向外曝露的 API 没几个，它们大致可被分为两类：域相关函数、翻译函数。

1：域相关函数

```
char* bindtextdomain(const char *__domainname, const char *__dirname);
```

创建一个名叫__domainname 的域，同时把__dirname”下的文件 wesnoth-lib.mo 绑定到这个域。

```
char* textdomain(const char *__domainname);
```

把__domainname 设置为当前域。

```
char*   bind_textdomain_codeset(const   char   *__domainname,   const   char
*__codeset);
```

用于指定域__domainname 输出的是__codeset 字符集

2：翻译函数

```
char* gettext(const char *__msgid);
```

在当前域以__msgid 为关键字搜出对应的 msgstr，返回这个 msgstr。

```
char* dgettext (const char *__domainname, const char *__msgid);
```

在__domainname 域以__msgid 为关键字搜出对应的 msgstr，返回这个 msgstr。退出前会自动把当前域设到它进入之前的当前域。

9.3 locale

之前在猜测 gettext 会以什么方式选择 po 时用了以下的话："内部有个存储当前操作系统正使用语言的字符串变量 locale，英文时 locale=en_GB，这时 gettext 就以 msgid 搜 en_GB.po，得到的 msgstr 就是英文下要显示的内容，简体中文时 locale=zh_CN，于是搜 zh_CN.po，繁体中文则搜 zh_TW.po。"虽然逻辑大致是这样，但描述太笼统，为进一步分析就须要问更多问题，像 locale 变量是什么？en_GB、zh_CN、zh_TW 这些标识是怎么来的？还有，程序如何根据 locale 变量运行时切换语言？ 回答它们就须要简单知道下 locale 历史。

9.3.1 历史

（以下内容摘自互联网，当中一些概念和之上说的有些差异，但作为不同人不同观点，不作修改放在此处，请读者自行理解。）

locale 是国际化与本土化过程中的一个非常重要的概念，个人认为，对于中文用户来说，通常会涉及到的国际化或者本土化，大致包含三个方面：看中文，写中文，与 window 中文系统的兼容和通信。从实际经验上看来，locale 的设定与看中文关系不大，但是与写中文，及 window 分区的挂载方式有很密切的关系。本人认为就像一个纯英文的 Windows 能够浏览中文，日文或者意大利文网页一样，你不需要设定 locale 就可以看中文。那么，为什么要设定 locale 呢？什么时候会用到 locale 呢？

一、为什么要设定 locale

正如前面我所讲的，设定 locale 与你能否浏览中文的网页没有直接的关系，即便你把 locale 设置成 en_US.ISO-8859-1 这样一个标准的英文 locale 你照样可以浏览中文的网页，只要你的系统里面有相应的字符集（这个都不一定需要）和合适的字体（如 simsun），浏览器就可以把网页翻译成中文给你看。具体的过程是网络把网页传送到你的机器上之后，浏览器会判断相应的编码的字符集，根据网页采用的字符集，去字体库里面找合适的字体，然后由文字渲染工具把相应的文字在屏幕上显示出来。

在下文本人会偶尔把字符集比喻成密码本，个人觉得对于一些东西比较容易理解，假如你不习惯的话，把全文 copy 到任何文本编辑器，用字符集替换密码本即可。

那有时候网页显示乱码或者都是方框是怎么回事呢？个人认为，显示乱码是因为设定的字符集不对(或者没有相应的字符集)，例如网页是用 UTF-8 编码的，你非要用 GB2312 去看，而系统根据 GB2312 去找字体，然后在屏幕上显示，当然是一堆的乱码，也就是说你用一个错误的密码本去翻译发给你的电报，当然内容那叫一个乱；至于有些时候浏览的网页能显示一部分

汉字，但有很多的地方是方框，能够显示汉字说明浏览器已经正确的判断出了网页的编码，并在字体库里面找到了相应的文字，但是并不是每个字体库都包含某个字符集全部的字体的缘故，有些时候会显示不完全，找一个比较全的支持较多字符集的字体就可以了。

既然我能够浏览中文网页，那为什么我还要设定 locale 呢？

其实你有没有想过这么一个问题，为什么 gentoo 官方论坛上中文论坛的网页是用 UTF-8 编码的（虽然大家一直强烈建议用 GB2312 编码），但是新浪网就是用 GB2312 编码的呢？而 Xorg 的官方网页竟然是 ISO-8859-15 编码的，我没有设定这个 locale 怎么一样的能浏览呢？这个问题就像是你有所有的密码本，不论某个网站是用什么字符集编码的，你都可以用你手里的密码本把他们翻译过来，但问题是虽然你能浏览中文网页，但是在整个操作系统里面流动的还是英文字符。所以，就像你能听懂英语，也能听懂中文。 最根本的问题是：你不可以写中文。

当你决定要写什么东西的时候，首先要决定的一件事情是用那种语言，对于计算机来说就是你要是用哪一种字符集，你就必须告诉你的 linux 系统，你想用那一本密码本去写你想要写的东西。知道为什么需要用 GB2312 字符集去浏览新浪了吧，因为新浪的网页是用 GB2312 写的。

为了让你的 Linux 能够输入中文，就需要把系统的 locale 设定成中文的(严格说来是 locale 中的语言类别 LC_CTYPE)，例如 zh_CN.GB2312、zh_CN.GB18030 或者 zh_CN.UTF-8。很多人都不明白这些古里古怪的表达方式。这个外星表达式规定了什么东西呢？这个问题稍后详述，现在只需要知道，这是 locale 的表达方式就可以了。

二、到底什么是 locale?

locale 这个单词中文翻译成地区或者地域，其实这个单词包含的意义要宽泛很多。Locale 是根据计算机用户所使用的语言，所在国家或者地区，以及当地的文化传统所定义的一个软件运行时的语言环境。

这个用户环境可以按照所涉及到的文化传统的各个方面分成几个大类，通常包括用户所使用的语言符号及其分类(LC_CTYPE)，数字 (LC_NUMERIC)，比较和排序习惯(LC_COLLATE)，时间显示格式(LC_TIME)，货币单位(LC_MONETARY)，信息主要是提示信息,错误信息, 状态信息, 标题, 标签, 按钮和菜单等(LC_MESSAGES)，姓名书写方式(LC_NAME)，地址书写方式(LC_ADDRESS)，电话号码书写方式 (LC_TELEPHONE)，度量衡表达方式(LC_MEASUREMENT)，默认纸张尺寸大小(LC_PAPER)和 locale 对自身包含信息的概述(LC_IDENTIFICATION)。

所以说，locale 就是某一个地域内的人们的语言习惯和文化传统和生活习惯。一个地区的 locale 就是根据这几大类的习惯定义的，这些 locale 定义文件放在/usr/share/i18n/locales 目录下面，例如 en_US, zh_CN and de_DE@euro 都是 locale 的定义文件，这些文件都是用文本格式书写的，你可以用写字板打开，看看里边的内容，当然出了有限的注释以外，大部分东西可能你都看不懂，因为是用的 Unicode 的字符索引方式。

对于 de_DE@euro 的一点说明，@后边是修正项，也就是说你可以看到两个德国的 locale：/usr/share/i18n/locales/de_DE@euro /usr/share/i18n/locales/de_DE 打开这两个 locale 定义，你就会知道它们的差别在于 de_DE@euro 使用的是欧洲的排序、比较和缩进习惯，而 de_DE 用的是德国的标准习惯。

上面我们说到了 zh_CN.GB18030 的前半部分，后半部分是什么呢？大部分 Linux 用户都知道是系统采用的字符集。

三、什么是字符集？

字符集就是字符，尤其是非英语字符在系统内的编码方式，也就是通常所说的内码，所有的字符集都放在/usr/share/i18n/charmaps，所有的字符集也都是用 Unicode 编号索引的。Unicode 用统一的编号来索引目前已知的全部的符号。而字符集则是这些符号的编码方式，或者说是在网络传输，计算机内部通信的时候，对于不同字符的表达方式，Unicode 是一个静态的概念，字符

集是一个动态的概念，是每一个字符传递或传输的具体形式。就像 Unicode 编号 U59D0 是代表姐姐的“姐”字，但是具体的这个字是用两个字节表示，三个字节，还是四个字节表示，是字符集的问题。例如：UTF-8 字符集就是目前流行的对字符的编码方式，UTF-8 用一个字节表示常用的拉丁字母，用两个字节表示常用的符号，包括常用的中文字符，用三个表示不常用的字符，用四个字节表示其他的古灵精怪的字符。而 GB2312 字符集就是用两个字节表示所有的字符。需要提到一点的是 Unicode 除了用编号索引全部字符以外，本身是用四个字节存储全部字符，这一点在谈到挂载 windows 分区的时候是非常重要的一个概念。所以说你也可以把 Unicode 看作是一种字符集（我不知道它和 UTF-32 的关系，反正 UTF-32 就是用四个字节表示所有的字符的），但是这样表述符号是非常浪费资源的，因为在计算机世界绝大部分时候用到的是一个字节就可以搞定的 26 个字母而已。所以才会有 UTF-8，UTF-16 等等，要不然大同世界多好，省了这许多麻烦。

四、zh_CN.GB2312 到底是在说什么？

Locale 是软件在运行时的语言环境, 它包括语言(Language), 地域 (Territory) 和字符集(Codeset)。一个 locale 的书写格式为: 语言[_地域[.字符集]]. 所以说呢，locale 总是和一定的字符集相联系的。下面举几个例子：

1、我说中文，身处中华人民共和国，使用国标 2312 字符集来表达字符。 zh_CN.GB2312＝中文_中华人民共和国＋国标 2312 字符集。

2、我说中文，身处中华人民共和国，使用国标 18030 字符集来表达字符。 zh_CN.GB18030＝中文_中华人民共和国＋国标 18030 字符集。

3、我说中文，身处中华人民共和国台湾省，使用国标 Big5 字符集来表达字符。zh_TW.BIG5=中文_台湾.大五码字符集

4、我说英文，身处大不列颠，使用 ISO-8859-1 字符集来表达字符。 en_GB.ISO-8859-1=英文_大不列颠.ISO-8859-1 字符集

5、我说德语，身处德国，使用 UTF-8 字符集，习惯了欧洲风格。 de_DE.UTF-8@euro＝德语_德国.UTF-8 字符集@按照欧洲习惯加以修正

注意不是[email]de_DE@euro.UTF[/email]-8，所以完全的 locale 表达方式是 [语言[_地域][.字符集] [@修正值]

生成的 locale 放在/usr/lib/locale/目录中，并且每个 locale 都对应一个文件夹，也就是说创建了[email]de_DE@euro.UTF[/email]-8 locale 之后，就生成/usr/lib/locale/de_DE@euro.UTF-8/目录，里面是具体的每个 locale 的内容。

五、怎样去自定义 locale

在 gentoo 生成 locale 还是很容易的，首先要在 USE 里面加入 userlocales 支持，然后编辑 locales.build 文件，这个文件用来指示 glibc 生成 locale 文件。 很多人不明白每一个条目是什么意思。 其实根据上面的说明现在应该很明确了。

File: /etc/locales.build en_US/ISO-8859-1 en_US.UTF-8/UTF-8

zh_CN/GB18030 zh_CN.GBK/GBK zh_CN.GB2312/GB2312 zh_CN.UTF-8/UTF-8

上面是我的 locales.build 文件，依次的说明是这样的：

en_US/ISO-8859-1：生成名为 en_US 的 locale，采用 ISO-8859-1 字符集，并且把这个 locale 作为英文_美国 locale 类的默认值，其实它和 en_US.ISO-8859-1/ISO-8859-1 没有任何区别。

en_US.UTF-8/UTF-8：生成名为 en_US.UTF-8 的 locale，采用 UTF-8 字符集。

zh_CN/GB18030：生成名为 zh_CN 的 locale，采用 GB18030 字符集，并且把这个 locale 作为中文_中国 locale 类的默认值，其实它和 zh_CN.GB18030/GB18030 没有任何区别。

zh_CN.GBK/GBK：生成名为 zh_CN.GBK 的 locale，采用 GBK 字符集。zh_CN.GB2312/GB2312：生成名为 zh_CN.GB2312 的 locale，采用 GB2312 字符集。zh_CN.UTF-

8/UTF-8：生成名为 zh_CN.UTF-8 的 locale，采用 UTF-8 字符集。

关于默认 locale，默认 locale 可以简写成 en_US 或者 zh_CN 的形式，只是为了表达简单而已没有特别的意义。

Gentoo 在 locale 定义的时候掩盖了一些东西，也就是 locale 的生成工具：localedef。 在编译完 glibc 之后你可以用这个 localedef 再补充一些 locale，就会更加理解 locale 了。具体的可以看 localedef 的 manpage。

$localedef -f 字符集 -i locale 定义文件 生成的 locale 的名称 例如 $localedef -f UTF-8 -i zh_CN zh_CN.UTF-8

上面的定义方法和在 locales.build 中设定 zh_CN.UTF-8/UTF-8 的结果是一样一样的。

六、locale 的五脏六腑

刚刚生成了几个 locale，但是为了让它们生效，必须告诉 Linux 系统使用那(几)个 locale。这就需要对 locale 的内部机制有一点点的了解。在前面我已经提到过，locale 把按照所涉及到的文化传统的各个方面分成 12 个大类，这 12 个大类分别是： 1、语言符号及其分类(LC_CTYPE) 2、数字(LC_NUMERIC) 3、比较和排序习惯(LC_COLLATE) 4、时间显示格式(LC_TIME) 5、货币单位(LC_MONETARY) 6、信息主要是提示信息,错误信息, 状态信息, 标题, 标签, 按钮和菜单等(LC_MESSAGES) 7、姓名书写方式(LC_NAME) 8、地址书写方式(LC_ADDRESS) 9、电话号码书写方式(LC_TELEPHONE) 10、度量衡表达方式(LC_MEASUREMENT) 11、默认纸张尺寸大小(LC_PAPER) 12、对 locale 自身包含信息的概述(LC_IDENTIFICATION)。

其中，与中文输入关系最密切的就是 LC_CTYPE， LC_CTYPE 规定了系统内有效的字符以及这些字符的分类，诸如什么是大写字母，小写字母，大小写转换，标点符号、可打印字符和其他的字符属性等方面。而 locale 定义 zh_CN 中最最重要的一项就是定义了汉字(Class “hanzi”)这一个大类，当然也是用 Unicode 描述的，这就让中文字符在 Linux 系统中成为合法的有效字符，而且不论它们是用什么字符集编码的。

```
LC_CTYPE % This is a copy of the "i18n" LC_CTYPE with the following modifications: - Additional classes: hanzi

copy "i18n"

class "hanzi"; / % ..;/ ..;/ ;;;;;;;;/ ;;;;;;;;/ ;;;; END LC_CTYPE
```

在 en_US 的 locale 定义中，并没有定义汉字，所以汉字不是有效字符。所以如果要输入中文必须使用支持中文的 locale，也就是 zh_XX，如 zh_CN，zh_TW，zh_HK 等等。

另外非常重要的一点就是这些分类是彼此独立的，也就是说 LC_CTYPE，LC_COLLATE 和 LC_MESSAGES 等等分类彼此之间是独立的，可以根据用户的需要设定成不同的值。这一点对很多用户是有利的，甚至是必须的。例如，我就需要一个能够输入中文的英文环境，所以我可以把 LC_CTYPE 设定成 zh_CN.GB18030，而其他所有的项都是 en_US.UTF-8。

七、怎样设定 locale 呢？

设定 locale 就是设定 12 大类的 locale 分类属性，即 12 个 LC_*。除了这 12 个变量可以设定以外，为了简便起见，还有两个变量：LC_ALL 和 LANG。它们之间有一个优先级的关系：LC_ALL>LC_*>LANG 可以这么说，LC_ALL 是最上级设定或者强制设定，而 LANG 是默认设定值。 1、如果你设定了 LC_ALL＝zh_CN.UTF-8，那么不管 LC_*和 LANG 设定成什么值，它们都会被强制服从 LC_ALL 的设定，成为 zh_CN.UTF-8。2、假如你设定了 LANG＝zh_CN.UTF-8，而其他的 LC_*=en_US.UTF-8，并且没有设定 LC_ALL 的话，那么系统的 locale 设定以 LC_*=en_US.UTF-8。 3、假如你设定了 LANG＝zh_CN.UTF-8，而其他的 LC_*，和 LC_ALL 均未设定的话，系统会将 LC_*设定成默认值，也就是 LANG 的值 zh_CN.UTF-8 。 4、假如你设定了 LANG＝zh_CN.UTF-8，而其他的 LC_CTYPE=en_US.UTF-8，其他的 LC_*，和 LC_ALL 均未设定的话，那么系统的 locale 设定将是：LC_CTYPE=en_US.UTF-8，其余的 LC_COLLATE，

LC_MESSAGES 等等均会采用默认值，也就是 LANG 的值，也就是 LC_COLLATE＝LC_MESSAGES＝……＝ LC_PAPER＝LANG＝zh_CN.UTF-8。

所以，locale 是这样设定的： 1、如果你需要一个纯中文的系统的话，设定 LC_ALL= zh_CN.XXXX，或者 LANG= zh_CN.XXXX 都可以，当然你可以两个都设定，但正如上面所讲，LC_ALL 的值将覆盖所有其他的 locale 设定，不要作无用功。 2、如果你只想要一个可以输入中文的环境，而保持菜单、标题，系统信息等等为英文界面，那么只需要设定 LC_CTYPE＝zh_CN.XXXX， LANG= en_US.XXXX 就可以了。这样 LC_CTYPE＝zh_CN.XXXX， 而 LC_COLLATE＝LC_MESSAGES＝……＝ LC_PAPER＝LANG＝en_US.XXXX。 3、假如你高兴的话，可以把 12 个 LC_*一一设定成你需要的值，打造一个古灵精怪的系统： LC_CTYPE＝zh_CN.GBK/GBK(使用中文编码内码 GBK 字符集)；LC_NUMERIC=en_GB.ISO-8859-1(使用大不列颠的数字系统) LC_MEASUREMEN=de_DE@euro.ISO-8859-15(德国的度量衡使用 ISO-8859-15 字符集) 罗马的地址书写方式，美国的纸张设定……。估计没人这么干吧。 4、假如你什么也不做的话，也就是 LC_ALL，LANG 和 LC_*均不指定特定值的话，系统将采用 POSIX 作为 lcoale，也就是 C locale。

9.3.2 LC_MESSAGES

locale 是根据计算机用户所使用的语言、所在国家或者地区、以及当地的文化传统所定义的一个软件运行时的语言环境，为方便把这个定义称为广义 locale；而把格式是“[语言[_地域][.字符集] [@修正值] ”形成的 locale 值称为狭义 locale，当说到 locale 是 zh_CN.GB2312 时，它指的是狭义 locale，因为专门指值，狭义 locale 也被称为 locale 值。

locale 被分 12 大类，即 12 个 LC_*，前文说到 gettext 根据 locale 值选择相应 po/mo 包，由于 12 个 LC_*可以有不同 locale 值，多个不一样的 locale 值自然得不到唯一结果，要得到唯一结果只能选择 12 个 LC_*中的一个 locale 值作为它的判定依据。从图 8-8 的 mo 目录结构大概已能猜到 getext 选的是 LC_MESSAGES，LC_MESSAGES 负责管理“提示信息，错误信息，状态信息，标题，标签，按钮和菜单等”。

C/C++从哪得到 LC_MESSAGE 的 locale 值？既然 gettext 要由这个 locale 计算出 mo 所在目录，让进入 gettext 看它是怎么得到 locale。

1. 在 kingdom 工程中打开 gettext 工程中的 localname.c。
2. 在_nl_locale_name_posix 函数内设断点。
3. 运行时选“Debug”——“Starting Debug”。
4. 如果第 3 步没触发断点（选择的是“系统默认语言”时不会触发），进入标题屏幕后把语言改一种非系统默认语言，例如“English（GB）”，修改后会触发断点。

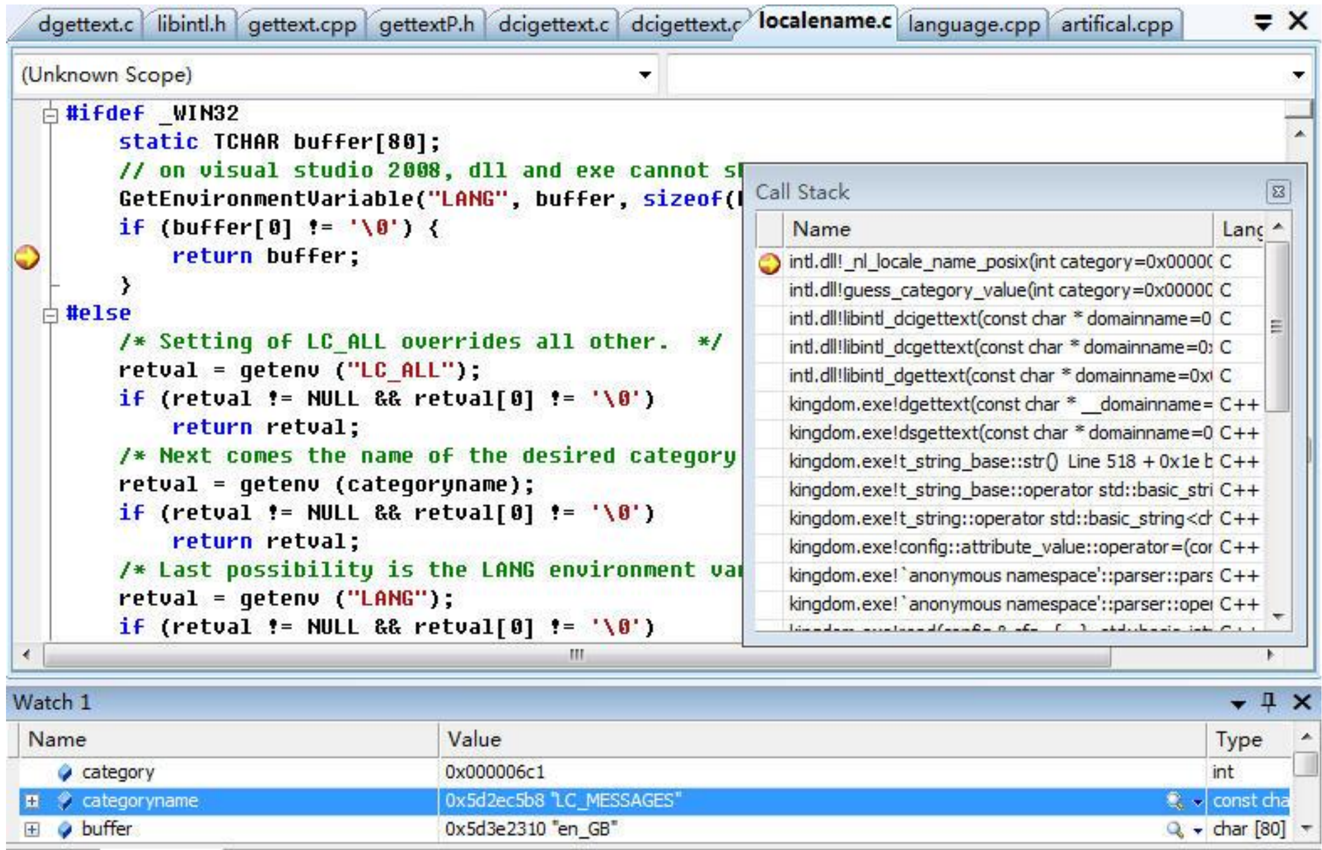


图 8-13 _nl_locale_name_posix

图 8-13 显示在 Window 系统下是从 Win32 进程环境变量区取 LANG 变量，LANG 变量值就是 locale；而在非 Windows 系统，它是从 C 运行库环境变量区中取，优先级是 LC_ALL > LC_MESSAGES > LANG，这个优先级符合“**8.3.1 历史**”中叙述的 LC_ALL>LC_*>LANG。

这里要叙述下我使用 gettext 历史。基于韦诺之战修改代码后，发现不能实现运行时切换语言（韦诺却可以），由于在用的 gettext 和韦诺不一样，怀疑 gettext 有问题。为找问题就要进入 gettext 执行源码调试，可惜从 GNU 下载的 gettext 不支持在 Visual Studio .Net 下编译，只好下载源码自己创建出一个新的 gettext 工程，也就是现在这个在用的工程。进入源码调式发现运行时不能切换语言是由于 Visual Studio 2010 已不支持 exe 和 dll 共享 C 运行库环境变量区，也就是说在 2010 下 getenv/setenv 已无效！那要如何解决这个问题，说来这里目的就是要让 exe 和 dll 共享一个变量，想到用 Win32 进程环境变量区，即 GetEnvironmentVariable/SetEnvironmentVariable 函数。由于这库说来也就是自个在用，在这里只是读取“LANG”变量，忽略 LC_ALL、LC_MESSAGES，反正只要应用程序那边记得设的是“LANG”就行，类似以下代码。

```
void wesnoth_setlocale(int category, std::string const &slocale, std::vector<std::string> const *alternates)
{
    std::string locale = slocale;
    ......
    if (category == LC_MESSAGES) {
        SetEnvironmentVariable("LANG", locale.c_str());
        return;
    }
    ......
}
```

下载的 gettext 没有把_WIN32 单独处理，是我加的。至于 getenv/setenv 无效问题，官方最新 gettext 应该已经解决，用的可能还是其它办法，但一来 gettext 库源码很乱（满篇不规则代码对齐），二来我想等验证过更多平台，至少要支持 Windows、iOS、Android，到时再和官方 gettext 版本进行同步。

图 8-13 显示出 gettext 是从变量区取变量值办法得到 locale，如果变量区不存在 LC_ALL、LC_MESSAGES、LANG 或是空值呢？这种情况是极可能会出现，像《王国战争》程序中选择的是“系统默认语言”，这时 gettext 将调用_nl_locale_name_default 得到默认 locale 值。

1. 在 kingdom 工程中打开 gettext 工程中的 localname.c。
2. 在_nl_locale_name_default 函数内设断点。
3. 运行时选“Debug”——“Starting Debug”，哪果没触发断点进入标题屏幕后把语言改系统默认语言，修改后会触发断点。

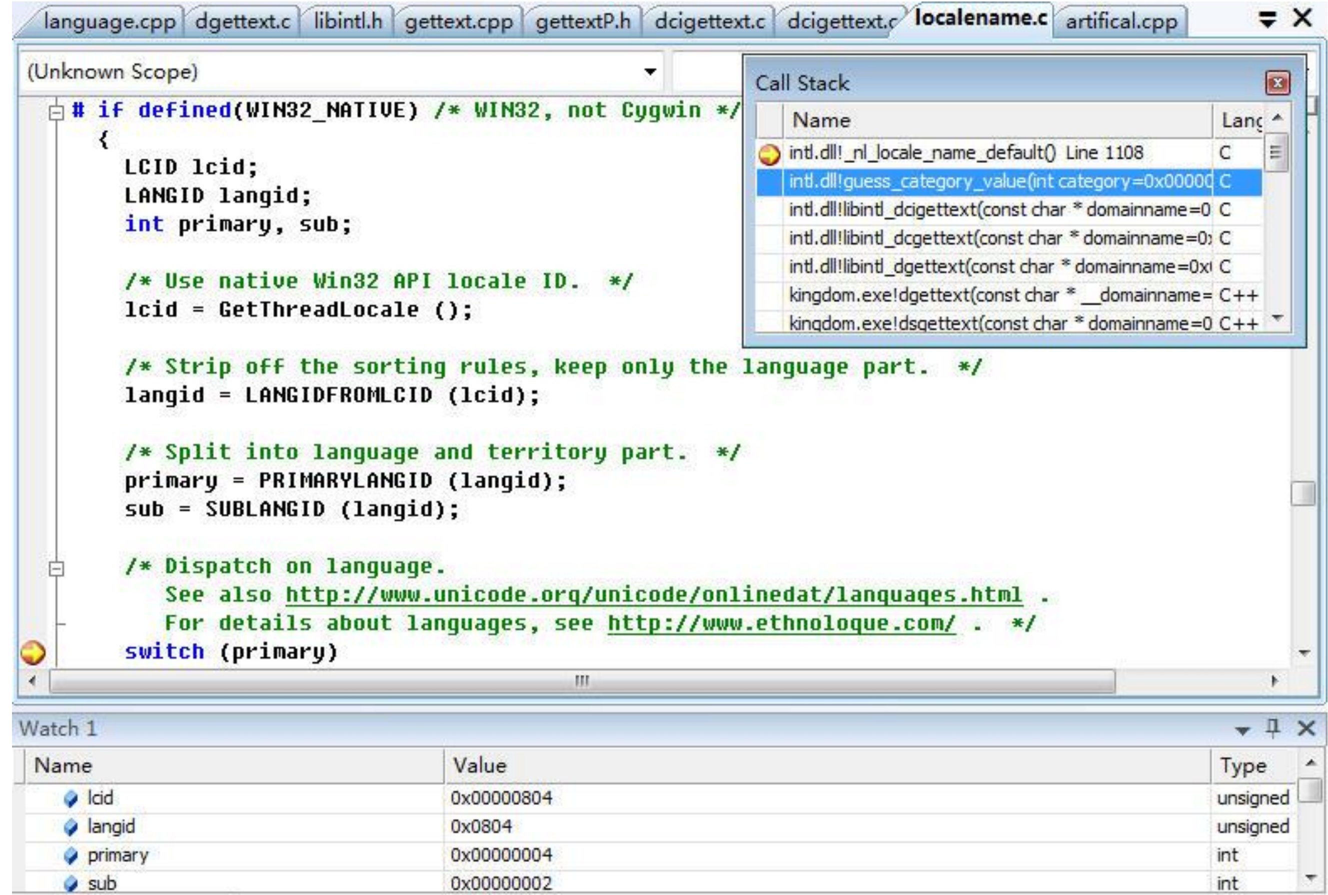


图 8-14 _nl_locale_name_default

在 Win32 下 gettext 是调用 GetThreadLocale()函数得到 locale 值。图 8-12 中得到语言主码值是 4（LANG_CHINESE），次码值是 2（SUBLANG_CHINESE_SIMPLIFIED），由紧接的 switch 大开关可得到<primary, sub>=(4, 2)对应 locale 值是“zh_CN”。

小结下 gettext 如何获取当前 locale 值及优先级：环境变量 LC_ALL ＞ 环境变量 LC_MESSAGES＞ 环境变量 LANG＞ 进程默认 locale。以下代码片断显示_nl_locale_name_posix、_nl_locale_name_default 搜索次序。

```
char* guess_category_value (int category, const char *categoryname)
{
    ...
    locale = _nl_locale_name_posix (category, categoryname);
    locale_defaulted = 0;
    if (locale == NULL) {
        locale = _nl_locale_name_default ();
        locale_defaulted = 1;
    }
    ...
    return locale;
}
```

知道如何读取 locale，接下要回答 gettext 是如何利用此个 locale 得到 mo 具体路径。让依旧进入代码调试。

1、在 kingdom 工程中打开 gettext 工程中的 dcigettext.c。
2、在 DCIGETTEXT 函数内设断点。（参考图 8-13）。
3、运行时选“Debug”——“Starting Debug”。

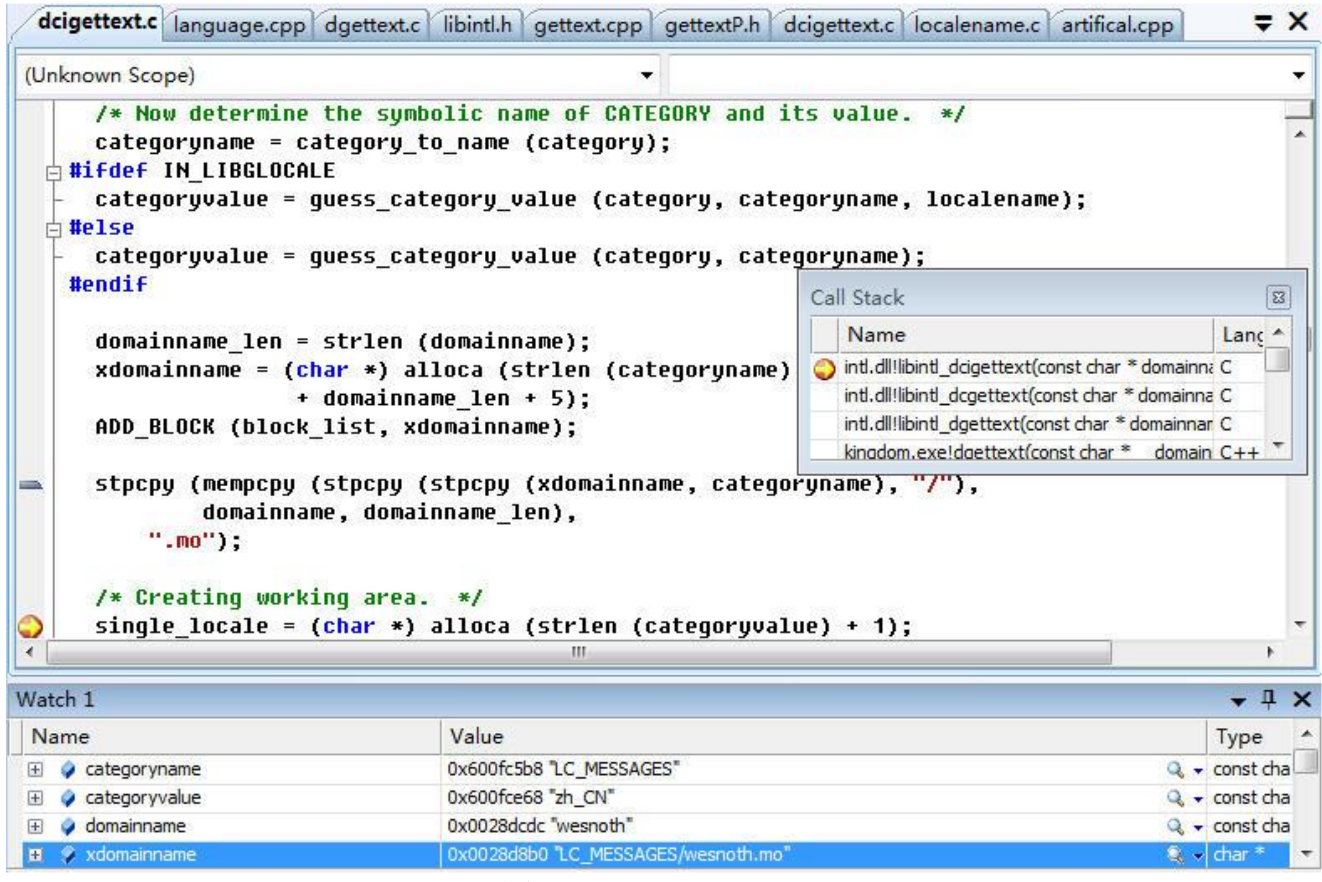


图 8-15 libintl_dcigettext

图 8-14 指示出 guess_category_value 得出 locale 后立即被用于和域名、“mo”连接，三部分组成 mo 文件“全”路径，即 Watch 窗口中 xdomainname 指示的“LC_MESSAGES/wesnoth.mo”。

到此可以小结下 gettext 对发布包中<translations>根下目录的命名要求：gettext 没“严格”规定<translations>根下是什么目录名，应用程序可以用设置 LC_ALL/LC_MESSAGES/LANG 环境变量实现自定义！应用程序设置值时最好符合 locale 规范，格式一般是：[语言[_地域][@修正值]，即去掉“字符集”的 locale 格式，像 ca_ES@valencia，en_GB、zh_CN、zh_TW。“严格”加双引号是因为一旦应用程序没设置 LC_ALL/LC_MESSAGES/LANG 环境变量，那 gettext 就会自己定义那些个目录名，即默认时 locale 值，此时命名规则参考<gettext>/gettext_runtime/intl/localname.c 中的_nl_locale_name_default 函数。

为全面补充说下 C/C++的 setlocale 函数。C/C++代码要得到当前 locale，使有运行库/进程环境变量区/自然不是 C/C++推荐方法（说白了就是在系统区开个变量让 exe 和 dll 能共享的老旧手段），甚至不是 GetThreadLocale()，C/C++推荐使用 C 运行库中的 setlocale，STL 库中的 std::setlocale（首选）。gettext 为什么不使用这两种方法？——对 C 运行库 setlocale，Microsoft 的 C 运行库只支持 LC_ALL/LC_COLLATE/LC_MONETRAY/LC_NUMRIC/LC_TIME，不支持 LC_MESSAGES，至少 Microsoft 已不能用此个 setlocale；对 STL 库 std::setlocale，gettext 是个纯 C 库，不希望引入 STL。

9.3.3 运行时切换语言

沿着图 8-13 会发现 gettext 在翻译每个字符串前都要去重新读取当前 locale，也就是说只要从切换语言那一刻开始改变 locale，gettext 接下翻译时就会以这个新 locale 进行翻译。这个机制使应用程序要实现运行时切换语言变得很简单：改成新语言配对 locale 值！根据“8.3.2

LC_MESSAGES”分析出的 gettext 如何读取 locale，设置新语言可分为两种办法。

- 要设为默认语言时，删除或置空 LC_ALL/LC_MESSAGES/LANG 环境变量。
- 要设为特定语言时，把该语言对应 locale 设置为 LC_ALL/LC_MESSAGES/LANG 的值。

运行时切换语言可能会遇到另一个问题：应用程序为提高翻译效率往往缓存已翻译出的结果！很显然，一旦切换到新语言，应该清掉原缓存。如果缓存是类似池这种集合，一个 clear 函数就够了，而缓存要是分散在一个个变量中，就像“王国战争”是用 class t_string 来封装要翻译字串，一个 t_string 封装了 msgid、msgstr，在切换语言那一刻系统中可能已形成许多到处分散的 t_string 对象，这时如何告知每个 t_string：语言变了，原来翻译出的已过时？

如何解决让进入代码调试。

1、打开 tstring.cpp。
2、在 t_string_base::str 函数内设断点。
3、运行时选“Debug”——“Starting Debug”。进入标题屏幕后选择“语言”，“确认”修改到新语言。

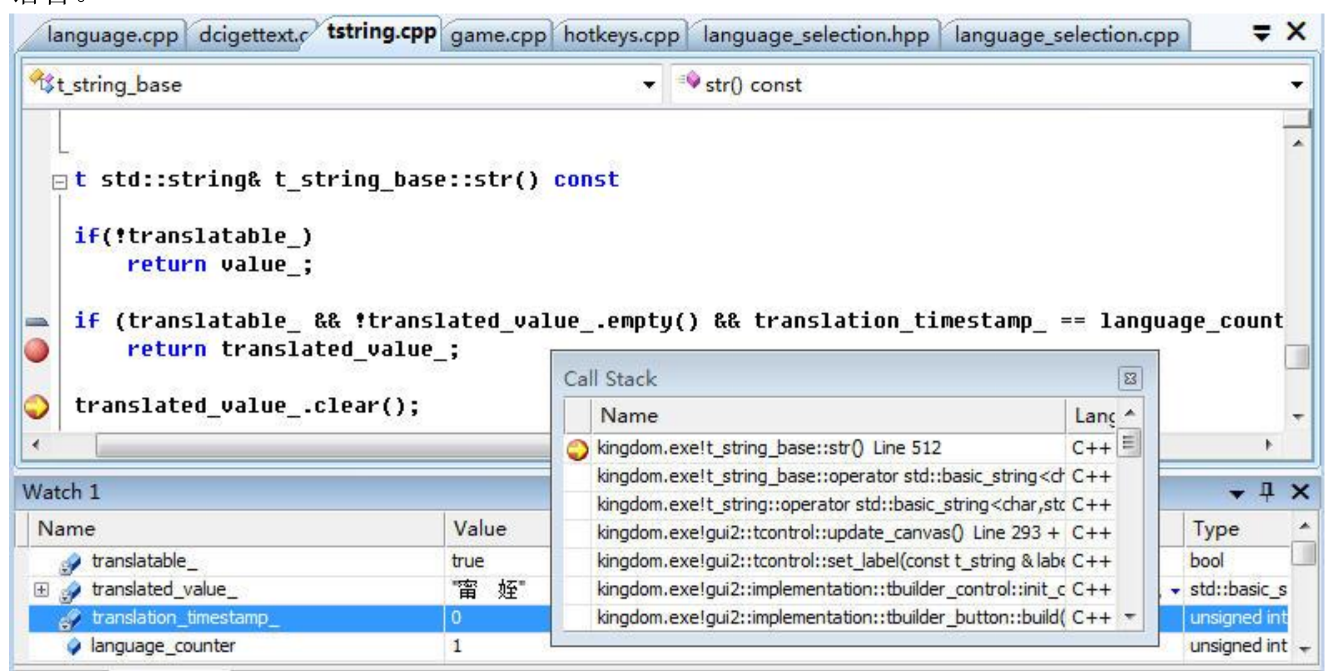


图 8-16 t_string_base::str

t_string_base 是 t_string 基类。translated_value_非空指示此个要翻译字符串已翻译过，translated_value_就是翻译出的值。t_string_base::str()为解决“自动”重新翻译问题加了个 translation_timestamp_成员变量！当它和“全局”变量 language_counter 一致时表示已翻译出的是“最新”语言，即 translated_value_可作为返回值直接传给上层程序，一旦不一致就认为已翻译内容根据的不是“最新”语言，需要重新翻译。

基于 t_string_base.str()此种新旧“语言”判断机制，切换语言时只要改下“全局”变量 language_counter，所有 t_string 会自动作废翻译缓存。

9.4 深入 gettext

9.4.1 输出编码

对 gettext 翻译返回的字符串，应用程序可能直接要以它为索引去搜 UNICODE 表，这时希望 gettext 输出是 UNICODE 格式；可能希望和其它 UTF-8 字符相连形成新字符串，这时希望输出是 UTF-8；可能和其它 GB2312 字符相连形成新字符串，这时希望是 GB2312，叙述 API 时已知道 gettext 是通过提供 bind_textdomain_codeset 让应用程序把翻译输出设置为它想要的编码格

式。那 gettext 是如何实现输出各种格式？

gettext 要支持输出多种格式，它必须要通过格式转式。gettext 以 msgid 搜出 msgstr 后，这个 msgstr 只一种格式，它须要把这单一格式转为 bind_textdomain_codeset 设定的多种格式。既然是转换，肯定存在转换前和转换后两种编码，在此让讨论下这前后编码是什么个格式。

转换前编码

转换前编码也叫 mo 编码、po 编码，它是 mo/po 文件中编码 msgstr 字符串的格式。注意：这种编码可能存在多种格式，可能他是以 UTF-8 格式存储 po，可能他是以 UTF-16 格式存储 po。

或许有人会问，搜索 msgid 会不会碰到要识别 mo 编码？——搜索 msgid 过程不须在意 msgstr 是什么编码，取出 msgstr 是个字节“复制”过程。Poedit 把 po 生成 mo 时会把所有 msgstr 集中放置在一块区域（一般在 mo 文件尾，UlterEdit 打开个 mo 可看到个大概），把它称为 msgstr 区，msgid 匹配后是以着地址偏移方式读出相应 msgstr。举个例子，msgid="Campaign"对应的是 msgstr 区中偏移 N 开始、长度是 M 那段字节，getext 取 msgstr 时就从偏移 N 开始读出 M 字节；msgid="Tutorial"对应的是 msgstr 中 J 开始、长度是 K 那段字节，于是“Tutorial”的 msgstr 就从 J 开始读出 K 字节。搜索 msgid 只管“复制”M、K 字节，不关心那些字节是什么格式，由于是“复制”，原来什么格式就是什么格式。

转换后编码

转换后编码就是应用程序调用 bind_textdomain_codeset 设置的输出编码。转换后编码须要是支持多种格式。

在此已能发现 gettext 的编码转换过程须支持输入编码是多样，输出编码也是多样，要如何实现这种转换？——引入中间格式，转换分两个步骤，第一步把输入格式转成中间格式，第二步把中间格式转成输出格式。

要实现以上转换过程，中间格式应该是种常用的统一码，自然而然会想到是 UNICODE。接下让进入源码看下转换时状态变量，进而得出 gettext 如何实现转换。

1、注释掉 game.cpp 中 init_loale()函数的 bind_textdomain_codeset (PACKAGE, "UTF-8")。
2、注释掉 gettext.cpp 中 dsgettext(...)函数的 bind_textdomain_codeset(domainname, "UTF-8")。
3、重编译 kingdom.exe。
4、在 kingdom 工程中打开 gettext 工程中的 loop_unicode.h。
5、在 unicode_loop_convert 函数内设断点。
6、运行时选“Debug”——“Starting Debug”。

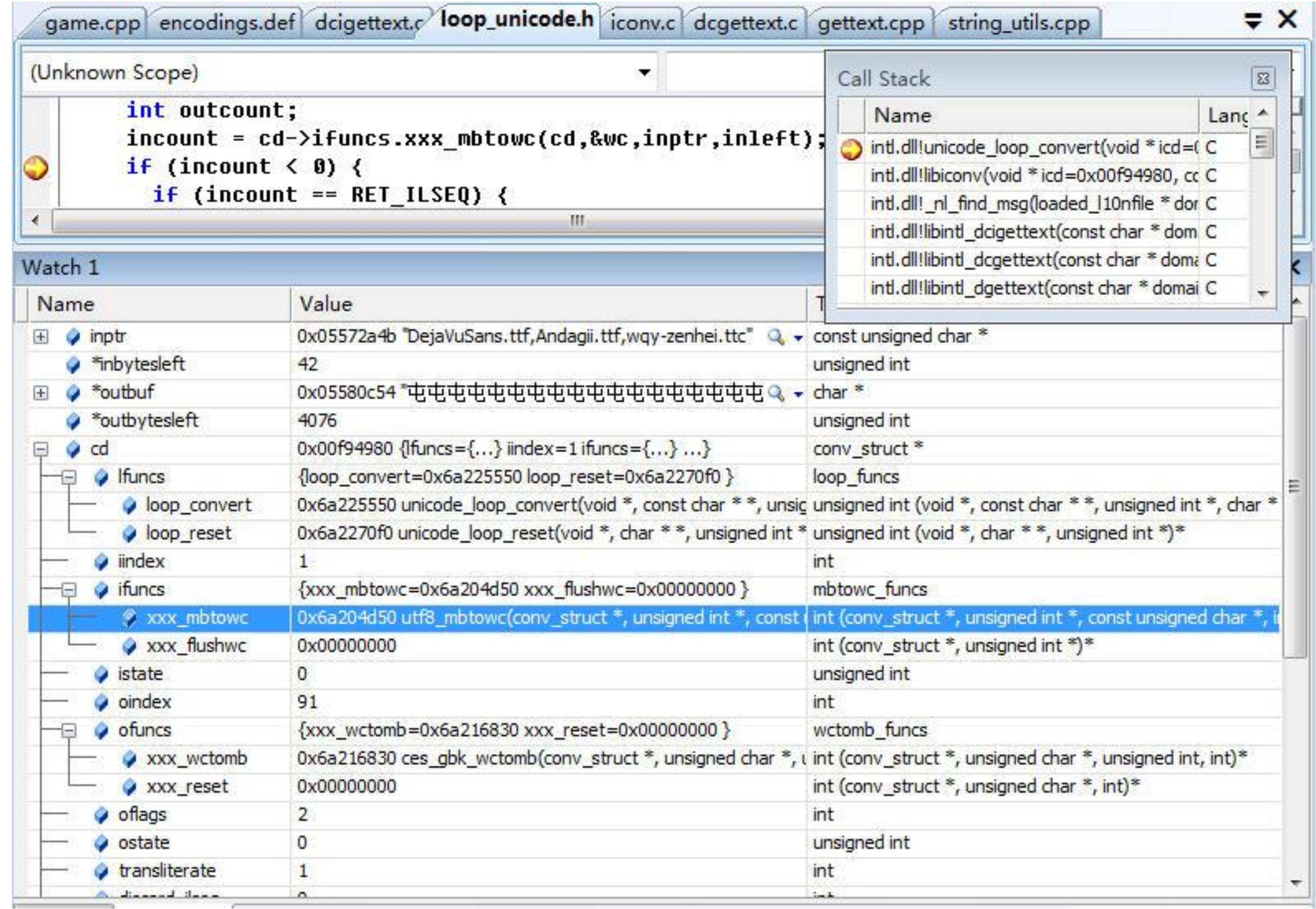


图 8-17 unicode_loop_convert

图 8-17 中，“incount = cd->ifuncs.xxx_mbtowc(...);”语句是调用 xxx_mbtowc 函数，Watch 窗口可看出这个 xxx_mbtowc 是个函数指针，真正执行的函数是 utf8_mbtowc。让看下 utf8_mbtowc 代码。——代码看去有点眼熟，不错，它就是用于把一个 UTF-8 字符转化成 UNICODE 字符，转换出的 UNICODE 放在 pwc，并返回此个 UTF-8 字符占用字节数。

此次要执行的转换要求是输入格式 UTF-8，输出格式 GBK（GB2312），把 UTF-8 转成 UNICODE 是借用中间格式方法中第一步，即把输入格式转成中间格式。至于第二步，往下看 unicode_loop_convert(...)会发现它要调用“outcount = cd->ofuncs.xxx_wctomb(cd, outptr, wc, outleft)”，通过 Watch 窗口知道 cd->ofuncs.xxx_wctomb 也是一个函数指针，真正执行的函数是 ces_gbk_wctomb，进去 ces_gbk_wctomb 会发现它执行的把一个 UNICODE 字符转化成 GBK 字符，正是中间格式方法中第二步，即把中间格式转成输出格式。

通过以上叙述大概能知道此个转换逻辑。

1. 输入字符串中还有要转换的字符吗？如果有执行 2，否则转换结束。
2. 取出第一个要转换字符，把它由输入格式转成中间格式，再由中间隔格式转成输出格式。
3. 要转换字符串去掉第一字符，跳去执行 1。

再看图 8-17 中 Watch，除了 ifuncs、ofuncs 存放函数指针，还有 lfuncs，它用于存储转换函数的指针，也就是此刻正在执行的 unicode_loop_convert。基于这三个函数指针，大致能猜到 gettext 正是通过针对不同格式环境让函数指针指向不同转换函数来实现支持多格式转换。例如要把输入格式 ISO-8859-6 转成输出格式 UTF-8，那 xxx_mbtowc、xxx_wctomb 则分别是 iso8859_6_mbtowc、utf8_wctomb。

由以上分析可得出 gettext 一大任务要根据输入、输出格式把相应函数挂接到 lfuncs、ifuncs、ofuncs 内指针，更准确说是要形成显示在 Watch 中类型是 conv_struct 的 cd 变量。如何形成

conv_struct，让再次使用调试。

1、注释掉 game.cpp 中 init_loale()函数的 bind_textdomain_codeset (PACKAGE, "UTF-8")。
2、注释掉 gettext.cpp 中 dsgettext(...)函数的 bind_textdomain_codeset(domainname, "UTF-8")。
3、重编译 kingdom.exe。
4、在 kingdom 工程中打开 gettext 工程中的 dcigettext.c。
5、在_nl_find_msg 函数内设断点。
6、运行时选“Debug”——“Starting Debug”。

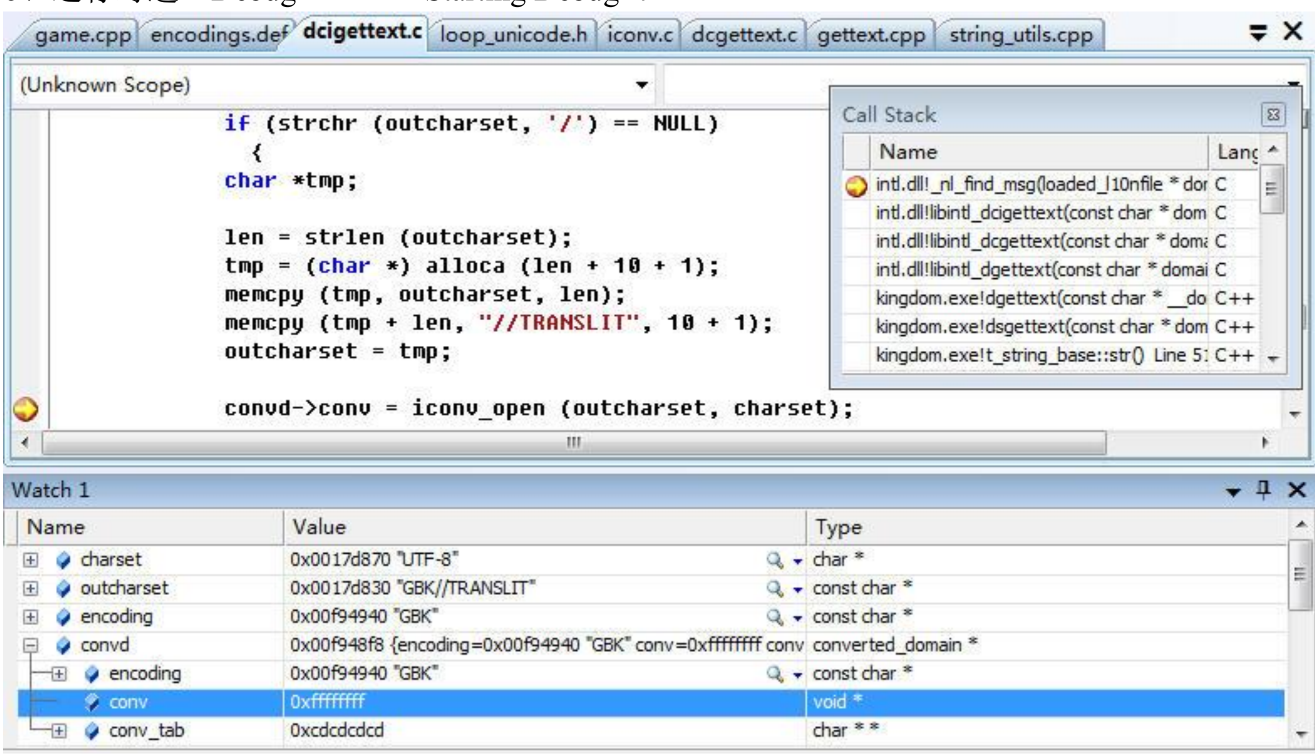


图 8-18 _nl_find_msg

图 8-18 中 convd->conv 就是要形成的 struct sonv_struct，此时它是无效的，但只要一执行完断点所指的 iconv_open 就会生效，像 struct 内所有函数指针都指向相应转换函数。charset 指示输入格式，outcharset 指示输出格式，iconv_open 根据输入、输出格式生成 conv_struct。接下让分析 charset、outcharset 是怎么来的。

charset 来自 mo 文件的 msgid=""对应的 msgstr（见_nl_find_msg(domain_file, domainbinding, "", 0, &nullentrylen)），它一般被放在头部，内容中有类似“charset=UTF-8”，它指示了此 mo 文件是 UTF-8 编码。假如 msgid=""对应的 msgstr 中找不到“charset=”，或 mo 中不存在 msgid="" 这个要翻译字符串，gettext 将不执行转换，同时忽略应用程序设的输出编码，原封不动向上返回从 mo 文件中读出的内容。

outcharset 来自 encoding + "//TRANSLIT"，encoding 来自 domain 中的 codeset 变量，也就是应用程序调用 bind_textdomain_codeset 设的输出格式。如果 bind_textdomain_codeset 设的输出格式是 NULL，gettext 会调用 localcharset.c 中的 local_charset()得到输出格式，得到就是地区码，像 GBK（简体中文），即 GB2312。此次调试前要注释掉两个 bind_textdomain_codeset 就是故意要让 codeset 是 NULL，促使 gettext 调用 local_charset()得到地区码，否则输入、输出都是 UTF-8 就不再执行转换。codeset 是 NULL 时会调用 local_charset()逻辑参考 dcigettext.c 中的 get_output_charset(...)。

小结

● 格式转换引入中间格式，分两个步骤，第一步把输入格式转成中间格式，第二步把中间格式

转成输出格式。

- 输入格式来自 po，具体是 msgid=""对应的 msgstr 中的“charset=”，输出格式来自 bind_textdomain_codeset 或 local_charset 得到的地区码，即 ANSI 编码。
- 输入格式和输出格式一致时，不执行转换。（官方 gettext 要转，我改为不转，实在想不出为啥要转）。

9.4.2 iconv

gettext 工程由两个项目组成：gettext、iconv，它们划分标准是什么？这里又要说下我使用 gettext 历史，官方 gettext 源码不包含 iconv，iconv 也是个 GNU 开源库，gettext 以着动态链接库形式调用 iconv.dll 来实现转码。那时要构建能在 Visual Studio .Net 编译的 gettext 工程时，感觉分成两个库不易于调试，发布时也会带来麻烦，就把它们合二为一，原来的 gettext 归入 gettext 项目，原来的 iconv 则归入 iconv 项目。这种合二为一作法同样也被带到了自建的 Xcode 下的 gettext。

iconv 具体实现什么功能？上文讨论 gettext 如何支持多种输出编码时已碰到两个函数，iconv_open(...)根据输入、输出格式生成 conv_struct，iconv(...)根据构建出的 conv_struct 执行转换一字符串，这两个函数由 iconv 提供，iconv 实现的功能正是对 gettext 搜索出的 msgstr 按应用程序要求进行转码。在功能定位上，GNU 项目 iconv 作用就是在多种国际编码格式之间进行文本内码的转换。

iconv 项目编译两个源文件：iconv.c、relocatable.c。relocatable.c 可认为没做事，也就是说 iconv 的实现都在 iconv.c。除去 iconv.c、relocatable.c，GNU 的 iconv 还有个很重要源文件 localcharset.c，乍看这名称有点眼熟，是的，上文说到的当 bind_textdomain_codeset 设的输出格式是 NULL 时，gettext 会调用 local_charset()得到地区码，正是这个 localcharset.c 实现了 local_charset()。GNU iconv 也要能检测出地区码，像 iconv_open(...)发现参数指示的输入格式是空时，于是作为独立工程 GNU iconv 须要 localcharset.c，加上 gettext 也有一个 localcharset.c，两个 c 文件实现同样功能（就是向外输出 local_charset()），它们会出现冲突。以下是 gettext 工程解决 localcharset.c 冲突办法。

- 整工程的 locale_charset()实现由 gettxt 项目提供，iconv 项目只是引入。
- iconv 项目下还是保留 locale_charset 的声明头文件，localcharset.h，它放在 <gettext>\libiconv\libcharset\include 目录下，这个文件内容和<gettext>\gettext\gettext-runtime\intl\localcharset.h 一样。如此设置使 iconv 要调用 locale_charset()时只要#include 本地目录（<gettext>\libiconv）的 localcharset.h，不涉及 gettext 目录（<gettext>\gettext），同时能更好和 GNU 的 iconv 保持一致，缺点是要存在两个 localcharset.h。
- 为减少规则 2 导致存在两个 localcharset.h 可能给复制造成的误操作，Visual Studio 在 iconv 项目“Build Events”——“Pre-Build Event”中增加条 copy 命令。“**copy $(SolutionDir)\..\gettext\gettext-runtime\intl\localcharset.h $(SolutionDir)\..\libiconv\libcharset\include\localcharset.h**”命令使得每次要编译 iconv 项目前会自动把 gettext 项目中的 localcharset.h 复制到 iconv 下的 localcharset.h，基于这个操作流程，如果要修改 localcharset.h，只要而且必须是改动 gettext 下那个。Visual Studio 使用 IDE 提供方法避免 localcharset.h 混乱，其它 IDE 则须要手动复制，像 Xcode。

9.4.3 内存泄漏

gettext 用单向链表来管理域，每个节点表示一个域，当要创建一个新节点时，用 malloc 从堆中分配一块内存。

```
struct binding* new_binding = (struct binding *)malloc(offsetof(struct binding, domainname) + len);
摘自 bindtextdom.c 中 set_binding_values(...)
```

C/C++代码得有相应 free 来释放这些内存。gettext 何时调用 free 来释放这些内存？——节

点内存要一直有效，释放要等到 dll 退出时，通常这个时刻就是退出应用程序时刻。要捕抓到退出时刻，Windows 提供 DllMain 的“case DLL_PROCESS_DETACH”，于是要释放节点内存只要在此个入口写上相关代码。Windows 能支持，其它系统就不那么幸运了。

无法捕捉到退出时刻的系统该怎么办？——可以让 gettext 向外提供个 API，应用程序在退出前调用它，这函数功能就是要 gettext 释放申请的内存资源。这样做的确可以实现这个目的，但回头再想想，即使退出前不释放内存又会造成什么恶果？无非就是退出前释放不干净，它不会降低程序运行速度、多使用内存，而且一旦退出，系统会自动回收该程序所有资源，自然包括那些没释放干净的内存。

官方 gettext 在这问题上没做处理，甚至包括能支持捕抓退出时刻的 Windows 系统。gettext 会造成内存泄漏，虽然不处理也没危害，但作为开发员得知道 gettext 会造成多少处内存泄漏，这样至少当你在代码中加入能检测“内存泄漏”功能的代码时可以分清哪些泄漏是“安全”的。

- _nl_current_default_domain 不等于 _nl_default_default_domain 内容时指向的内存。_nl_current_default_domain 指向当前域的域名称，它初始时等于_nl_default_default_domain，一旦应用程序设定一个新当前域名后，它就会从复制出新域的域名称，它指向这块内存。
- 注册域时分配的内存。注册指的是要向系统登记域时，把该域相关相息“形成”一个节点，并加入域链表，在“形成”结点过程中会从堆中分配数块内存。代码角度来说，“形成”结点是针对一个域形成一个 struct bingding 结构变量，分配出的数块内存是形成该结构的特定字段。

  dirname 内存。dirname 存放 mo 资源包目录，gettext 使用 strdup 从参数“复制”出 dirname。
  codeset 内存。codeset 存放输出格式，gettext 使用 strdup 从参数“复制”出 codeset。
  节点自身占用内存。(struct binding *)malloc(offsetof(struct binding, domainname) + len)分配的内存。一个域名一个。
- 加载域时分配的内存。加载指的是要翻译具体 msgid 时，把该 msgid 归属的域从磁盘文件读到内存，在“读”的过程中会从堆中分配数块内存。对某个域，整个运行过程中只会“读”一次。代码角度来说，“读“是针对一个域形成一个 struct loaded_l10nfile 结构变量，分配出的数块内存是形成该结构的特定字段。

  一个 struct loaded_l10nfile 结构变量。
  filename：文件名（域名）。
  data。域数据，类型 struct loaded_domain。既然是把磁盘文件读到内存，很自然会想得程序得开一个大小是文件字节数的数据缓存，读时把文件内存读入这缓存中，但 data 并不是这个缓存，而是 loaded_domain.data。释放时 loaded_domain 除了要释放 loaded_domain.data，还有转换结构 loaded_domain.conversions，以及为支持复数而引入的结构 loaded_domain.plural。在释放 data 中内存后，也要释放 data 自身。
  struct loaded_l10nfile 这个结构变量自身。

内存泄漏上再注意下 alloca，该函数在 Windows 平台时对应的是_alloca，其它则是 alloca，它的功能是在函数栈中分配内存。用 alloca 分配内存时，一旦调用它的函数退出了，系统会自动回收内存，因而不必也不要在函数退出前调用 freea。由于它这种函数一退出便自动释放特性，用它分配小内存还有益处的，像 DCIGETTEXT 中的 xdomainname、single_locale。

第十章 构造规则

本章目标

- 如何编写正六边形的构造规则
- 放置全局图像中原点对齐、中心点对齐
- 地形匹配逻辑
- 如何减少规则匹配判断次数
- 如何渲染出运行时大地图

程序中地图不是一张大位图，而是由一系列小格子图像拼凑而成。地图要模拟现实中地貌，它就需要显示各样地形，像山地、丘陵、平原、浅水、深水，还有在各样地形间须要有过渡带。

一个格子上地形一般有三种：基本层、覆盖层和过渡带。每种地形都需要一个图像，也就是说一个格子可能至少须要叠加三个图像，这时就须要按一定规则显示这些图像。

有时一种地形可能不只覆盖一个格子，这时就要把这地形在多个格子间拆开显示。

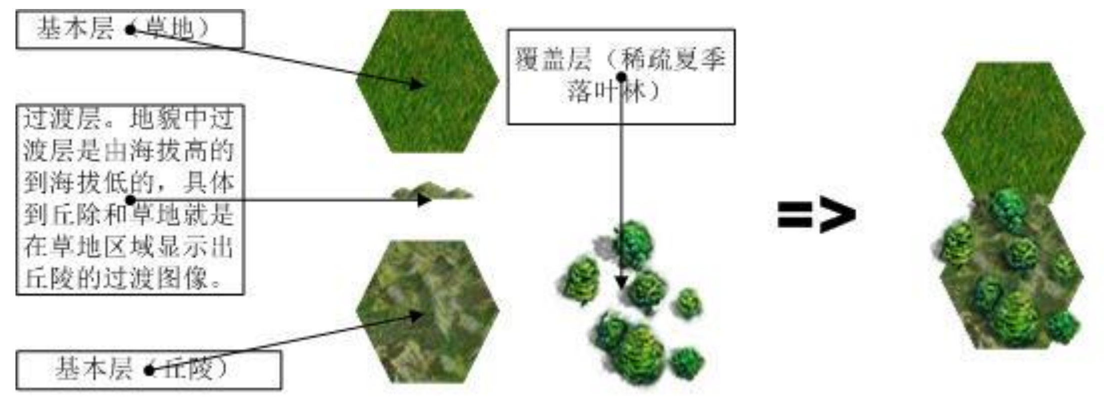


图 4-1 地形层

10.1 通过一个例子直观理解地形系统

10.1.1 地图编辑器画地图

生成画有“夏季落叶覆盖的丘陵”地形的地图

1. 运行程序，进入标题屏幕后选“地图编辑器”。
2. 菜单“地图”，打勾“绘制格子坐标”，打勾“绘制地形编码”。“改变地图大小”，新宽度和新高度置为 5x3。
3. 在格子(1, 1)处画上“夏季落叶覆盖的丘陵”地形，地形码是 Hh^Fds。

图 4-2 画夏季落叶林

4. 打开<kingdom-src>/data/core/terrain.cfg，搜索关键字“Hh^Fds”，找到它所在[terrain_type]块。这个 terrain_type 块指示了“夏季落叶林”这种地形信息，包括这种地形叫什么；它的地形标识（地形标识参考 **4.3.1 约束匹配**）；它在地图编辑器中属于哪个组。
5. “文件”——“保存地图”，文件名设置为“test”。地图文件默认放在<Documents>/My Games/kingdom/editor/maps。用文本编辑器对打开“test”，对比下地图和文件中内容。

```
border_size=1
usage=map

Gg  , Gg  , Gg    , Gg  , Gg  , Gg  , Gg
Gg  , Gg  , Gg    , Gg  , Gg  , Gg  , Gg
Gg  , Gg  , Hh^Fds, Gg  , Gg  , Gg  , Gg
Gg  , Gg  , Gg    , Gg  , Gg  , Gg  , Gg
Gg  , Gg  , Gg    , Gg  , Gg  , Gg  , Gg
```

“test”存储着此个地图各个格子地形（包括边界），把“test”称作地形配置文件。

找出和“Hh^Fds”地形关联的构造规则

构造规则用于关联图像和地形，描述了何样地形画哪样图像。

1、打开<kingdom-src>/data/core/terrain-graphics.cfg，搜索关键字“Fds”。找出两行：

```
{NEW:FOREST H*^Fds,M*^Fds (C*,K*,X*,Q*,W*,Ai,M*,*^V*,*^B*)  forest/deciduous-summer-sparse}
{NEW:FOREST    *^Fds    (C*,K*,X*,Q*,W*,Ai,M*,*^V*,*^B*)    forest/deciduous-summer}
```

这两行就是和该地形关联的所有构造规则。

2、打开<kingdom-src>/data/core/terrain-graphics/new-macros.cfg。搜索“#define NEW:FOREST”。

```
#define NEW:FOREST TERRAINLIST ADJACENT IMAGESTEM
    # This assumes centered images. Places the images named
    # {IMAGESTEM}-small[1-11].png on all {TERRAIN} adjacent to {ADJACENT}, and
    # {IMAGESTEM}[1-11].png on all others.

    [terrain_graphics]
        map="
,  2
*,   *
,  1
*,   *
,  *"
        [tile]
            pos=1
            type={TERRAINLIST}
            no_flag=forest
            set_flag=forest,small_forest
        [/tile]

        [tile]
            pos=2
            type={ADJACENT}
        [/tile]

        rotations=n,ne,se,s,sw,nw

        [image]
            name={IMAGESTEM}-small@V.png
            variations=";2;3;4;5;6;7;8;9;10;11"
            layer=0
            base=90,144
            center=90,144
        [/image]
    [/terrain_graphics]

    [terrain_graphics]
        map="
,  *
```

```
*, *
, 1
*, *
, *"
        [tile]
            pos=1
            type={TERRAINLIST}
            set_no_flag=forest
        [/tile]

        [image]
            name={IMAGESTEM}@V.png
            variations=";2;3;4;5;6;7;8;9;10;11"
            layer=0
            base=90,144
            center=90,144
        [/image]
    [/terrain_graphics]
#enddef
```

NEW:FOREST 定义了两种构造规则。由于使用了模板，每种规则各能产生 6 条规则，因而一个 NEW:FOREST 宏能产生 12 条规则。

第一种规则是当满足条件

1. (1, 1)格子地形匹配“H*^Fds,M*^Fds”。
2. (1, 1)格子没有置 forest 标志。
3. (1, 0)格子地形匹配(C*,K*,X*,Q*,W*,Ai,M*,*^V*,*^B*)。

它就执行

1. 在(0, 0)格子随机画一个图像，图像是 forest/deciduous-summer-sparse-small.png、forest/deciduous-summer-sparse-small2.png、forest/deciduous-summer-sparse-small3.png、forest/deciduous-summer-sparse-small4.png、forest/deciduous-summer-sparse-small5.png、forest/deciduous-summer-sparse-small6.png、forest/deciduous-summer-sparse-small7.png、forest/deciduous-summer-sparse-small8.png、forest/deciduous-summer-sparse-small9.png、forest/deciduous-summer-sparse-small10.png、forest/deciduous-summer-sparse-small11.png 中随机取一个。
2. 在(1, 1)格子设置 forest、small_forest 标志。

第二种规则是当满足条件

1. (1, 1)格子地形匹配“*^Fds”。
2. (1, 1)格子没有置 forest 标志。

它就执行

1. 在(0, 0)格子随机画一个图像，图像是 forest/deciduous-summer-sparse.png、forest/deciduous-summer-sparse2.png、forest/deciduous-summer-sparse3.png、forest/deciduous-summer-sparse4.png、forest/deciduous-summer-sparse5.png、forest/deciduous-summer-sparse6.png、forest/deciduous-summer-sparse7.png、forest/deciduous-summer-sparse8.png、forest/deciduous-summer-sparse9.png、forest/deciduous-summer-sparse10.png、forest/deciduous-summer-sparse11.png 中随机取一个。
2. 在(1, 1)格子设置 forest 标志。

因为(1, 1)格子处的 forest 标志，这两种规则是互斥的，也就是说即使真要画图像，在(1，1)处也只会画 deciduous-summer-sparse-small@V.png 或 deciduous-summer-sparse@V.png

terrain-graphics.cfg 中有两个满足条件的 NEW:FOREST，就是说构造规则集合中符合条件有四种规则（24 条），但由于 forest 标志原因，这四种规则是互斥的。实际生成地图时使用哪种规则就是看谁先被定义，因为谁先被定义就指示谁将先被判定能不能使用，而一旦判断定出它可以使用它就抢占设置了 forest 标志。

画上毗邻地形

步骤 1：(1, 0)格子画上“礁石”（Wwr）地形。

图 4-3 画礁石地形

会发现(1, 1)处图像中的树变希疏了。

造成变化的原因是由原来满足 NEW:FOREST 第二种规则变为满足了第一种，于是叠加的图像文件由 deciduous-summer-sparse@V.png 改为 deciduous-summer-sparse-small@V.png。

步骤 2：(1, 0)格子按鼠标右健，让地形恢复回“平原”（Gg）地形。

Gg 不满足规则一中的地形符合(C*,K*,X*,Q*,W*,Ai,M*,*^V*,*^B*)，致使画上的图像文件又由 deciduous-summer-sparse-small@V.png 改回 deciduous-summer-sparse@V.png。

步骤 3：(1, 0)格子画上“沼泽”（Ss）地形。

沼泽和平原一样不符合地形条件(C*,K*,X*,Q*,W*,Ai,M*,*^V*,*^B*)，导致即使在（1,0）画上沼泽后(1, 1)中图像不产生变化。

小结

- 地形系统有两个输入一个输出（参考图 4-20），两个输入是地形配置文件、构造规则集，一个输出是用户最终看到的地图。
- 可以这么认为：地形系统形成地图的过程就是依次扫描地形配置文件中各个格子；针对每个格子，扫描构造规则集中每条规则，一旦判断出满足就应用上该规则，而构造规则叙述了在此地形上如何画图像，由此就画出了最终用户看到的地图。
- 构造规则在集合中优先级会影响最终用户看到的地图，而定义次序就决定了构造规则在集全中优先级，越早定义优先级越高。

10.1.2 格子上图像形成过程

针对地图中某个格子，一条构造规则就可能向它叠加数个图像；而且，可能两条甚至更多条构造规则会应用在该格子，由此导致结果是该格子会叠加数个图像。这里让总结下会叠加在图 4-2 中（1，1）格子上的所有图像。（要分析哪条规则叠加了哪图像，这不是那么简单的，你看完此章后通可自行去分析。）

（1，1）格子地形标识是 Hh^Fds，它包括两种“单”地形，基本层：丘陵（Hh），覆盖层：夏季落叶林（Fds）。通过代码中设置断点，可察看到叠加在该格子上的所有 12 个图像。（表格中图像已被按渲染时渲染次序排序。）

序号	图像文件	约束偏移	Center
0	terrain/grass/green7.png	(1, 2)	(90,144)
1	terrain/grass/green6.png	(2, 2)	(90,144)
2	terrain/grass/green6.png	(0, 2)	(90,144)
3	terrain/grass/green.png	(2, 1)	(90,144)
4	terrain/hills/regular2.png	---	(0, 0)局部图像
5	terrain/hills/regular-n.png	(1, 0)	(90,144)
6	terrain/hills/regular-ne.png	(2, 1)	(90,144)
7	terrain/hills/regular2-se.png	(2, 2)	(90,144)
8	terrain/hills/regular2-s.png	(1, 2)	(90,144)
9	terrain/hills/regular-sw.png	(0, 2)	(90,144)
10	terrain/hills/regular-nw.png	(0, 1)	(90,144)
11	terrain/forest/deciduous-summer-sparse3.png	(1, 1)	(90,144)

查看每叠加一个图像后格子(1, 1)上图像变化

1. 没叠加一个图像。主图面是不透明的纯黑色。

图 4-4 未叠加任何图像时格子

2. 只叠加#0、#1、#2、#3 图像。
   格子(1, 1)显示出图像和上步“没叠加一个图像”时一样。（从#0、#1、#2、#3 裁剪出、要叠加到(1, 1)格子的都是空图像）。
3. 叠加#4 图像：terrain/hills/regular2.png。

图 4-5 叠加基本层地形：丘陵

4. 叠加#5 图像。
   格子(1, 1)显示出图像和第 3 步结束时一样。（从#5 裁剪出、要叠加到(1, 1)格子的是空图像）
5. 叠加#6 图像。terrain/hills/regular-ne.png。
   看上去，叠加#6 后格子(1, 1)显示出图像和第 4 步结束时一样，但其实是叠加了的，只是因为较难分辨。
6. 叠加#7 图像。terrain/hills/regular2-se.png。
   看上去，叠加#7 后格子(1, 1)显示出图像和第 5 步结束时一样，但其实是叠加了的，只是因为较难分辨。
7. 叠加#8、#9、#10 图像。
   格子(1, 1)显示出图像和第 6 步结束时一样。（从#8、#9、#10 裁剪出、要叠加到(1, 1)格子的

都是空图像）。

8. 叠加#11 图像：terrain/forest/deciduous-summer-sparse3.png。

图 4-6 叠加覆盖层地形：夏季落叶林

A：为什么叠加的图像文件有的不带数字，像 regular-n.png，有的却带数字，像 regular2.png？

Q：构造规则文件名中用的了类似 name+ variations 命名方式（参考 **4.2.9 [image]块中文件名**）。

```
name={IMAGESTEM}-small@V.png
variations=";2;3;4;5;6;7;8;9;10;11"
```

这时@V 部分取什么值依赖于紧跟的“variations”字段值，由于是随机在“variations”取值，造成每次取到的文件名可能不一样。即使同一个程序，不同时刻运行出来的图像也可能不一样。

A：为什么不显示 green.png、green4.png、green5.png、green6.png、green7.png 这些图像？

Q：green.png、green4.png、green5.png、green6.png、green7.png，包括之下的 regular-n.png、regular-sw.png、regular-nw.png、regular2-se.png、regular2-s.png，它们是全局图像，从设置了 Center 看，它们放置时都使用中心对齐方式，显示时使用中心对齐方式下全图像显示规则。这些图像实际尺寸是 72x72，但根据左上角坐标值（约束偏移和 Center 共同作用）计算出的“裁剪矩形”内是空图像导致不显示。关于全局图像参考“**4.2.7 绘制全局图像**”。

注：它们的约束偏移若改为(1, 1)就会显示。

A：为什么不显示一个图像时，在它的周围也存在类似丘陵的图像？

Q：因为构造集合中还在这条规则。

```
{TRANSITION_COMPLETE_L (Mm,Hh) (!,Hh,W*,S*) -180 hills/regular}
```

当(1, 1)格子是丘陵时(Hh)时它会在周围画上过渡带图像。

10.2 构造规则（build rule）

[terrain_graphics]标签用于关联图像和地形，这个关联关系称为地形图形规则（构造规则）。构造规则实现了如何把地图数据（以地形字符串表示）变成一系列玩家看到的位图图像。

在 WML 层次上，[teerain_graphics]既可以是顶层标签，也可以是 scenarios 下的直接标签。

10.2.1 定义一条规则

只须要定义一个[terrain_graphics]块就可定义一条构造规则。最简单的构造规则看起来是这样的。

```
[terrain_graphics]
[/terrain_graphics]
```

当然，这条规则就是空的，它没什么作用。

10.2.2 向规则添加约束

为让规则做点有意义事，首先须向规则添加条件，它指出了这规则将匹配地图上哪（些）个格子。WML 中的[tile]块用于定义条件。在本文档余下部分，把一个[tile]称为一条“约束”。

例如，下面的 WML 代码向空规则添加一条约束：

```
[terrain_graphics]
   [tile]
```

```
        x=0
        y=0
        type=Gg
    [tile]
[terrain_graphics]
```

让看下在这个 WML 标签中的元素：

- x 和 y 定义该条约束在该条规则内偏移（简称**约束偏移**）。
- type 定义约束应该匹配的地形。

假设地形地图是下面这样（格子内的字母代表地形码）

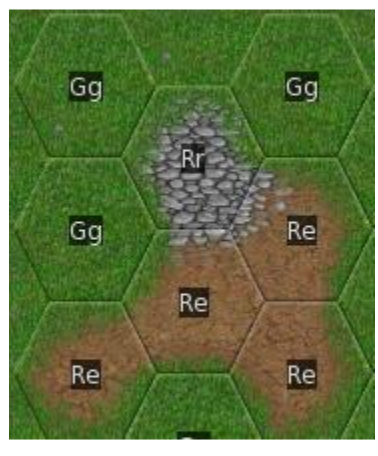


图 4-10 约束 I

以上定义的规则就匹配该地图上所有地形是 Gg 的格子。

那么如何使用“偏移”？偏移可以使规则要匹配的不仅仅是单个格子，而是把几个格子联合成“单个”地形，使得规则的条件成为要和这“单个”地形匹配。举个例子，对要能匹配以下两个格子组成的“单个“地形：

图 4-11 约束 II

要实现目标，须要定义一条类似下面的构造规则。

```
[terrain_graphics]
    [tile]
        x=0
        y=0
        type=Gg
    [/tile]
    [tile]
        x=1
        y=0
        type=Rr
    [/tile]
[/terrain_graphics]
```

这条规则有两个约束：

- 第一个约束，偏移(0, 0)处地形应该是“Gg”
- 第二个约束，编移(1, 0)处地形应该是“Rr”

10.2.3 六角坐标系统

为说明构造规则系统中两个坐标之间关系，需引入程序使用的栅格坐标模型。**模型强制性**

定义：左上角坐标被编号为(0, 0)，它的东南角格子则被编为(1, 0)，而(1,0)的东北角则编号为(2, 0)，等等。在表述上，常把位在东南角格子称为紧接的格子，像紧接(0, 0)的格子是(1, 0)。以下是用图片表示六角坐标系统。**把(0, 0)称为规则原点，位在原点的约束称原点约束。**

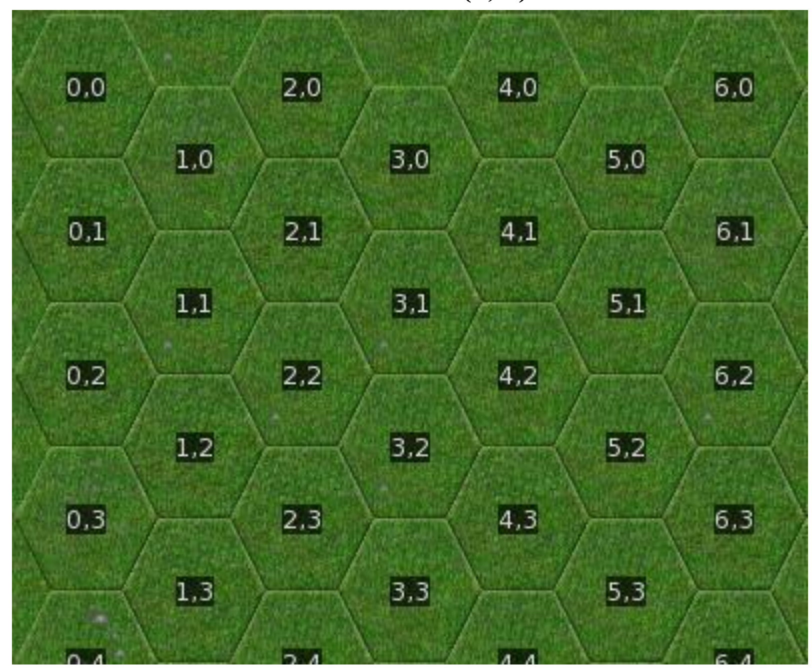


图 4-12 六角坐标系统

当我们说一个从(xf, yf)偏移了(xo, yo)格子，指的就是如果(xf,yf)是(0,0)，那(xo,yo)就是那个格子。在此，得出的格子并不总是两个格子的坐标和。例如从(2,0)偏移了(1,0)格子是(3,0)，但从(3,0)偏移了(1,0)格子却是(4,1)！

10.2.4 type 字符串

如何写一个能够匹配数种地形类型的约束？——只要把希望匹配的地形写成列表作为[tile]块中“type”的值。例如，以下规则将匹配任何或是深水（Wo）或是丘陵（Hh）地形。

```
[terrain_graphics]
    [tile]
        x=0
        y=0
        type=Wo,Hh
    [/tile]
[/terrain_graphics]
```

type 字符串可使用的和地形相关特殊符号：

- “.”：占位符。用于计算各约束偏移坐标、计算构造规则矩形等，不用于做任何判断。
- “*”（START）：通配符。它既可以作为独立地形，也出现在分量中。作为独立地形时，它匹配所有地形。出现在分量时，它可以匹配四字符、三字符、二字符或一字符。如 H*，它可以匹配 H 开头的所有标识，像 Hh、Hhd。
- “!”（NOT）：非符。它不能作为独立地形，且须要和其它地形结合使用。要匹配地形不出现在之后的地形列表中时，匹配结果返回真。(!,*^_fme)，要匹配地形不满足*^_fme 时，(!,*^_fme)返回真。

类型列表格式：以逗号分隔各地形，像(!,*^_fme)，(*^Br\,*^Br|,*^Br/)。

10.2.5 向地形添加图像

到现在，已知道如何在规则中写匹配条件，接下要为规则做些实用的事，那就是向地形添加图像。这添加是在规则中添加[image]块，基本[image]语法是下面格式。

```
[image]
```

```
    name=<name>
[/image]
```

<name>指示要显示的图像文件。在处理上，程序会自动向它加前缀“terrain/“，后缀“.png”，由它们来合成“最后”文件名。例如，name=forest/deciduous-summer-sparse，“最后”形成的文件名是 terrain/forest/deciduous-summer-sparse.png。

[image]既可以出现在约束根下也可以出现在规则根下，约束根下的称作局部[image]，规则根下的称作全局[image]。

约束中的局部[image]

如果约束中有[image]，指定图像须是单六边形尺寸图像（当前就意味着尺寸需是 72x72）。一旦判断出规则匹配（满足该规则中所有约束），那么图像就会被添加到约束对应格子的坐标处。例如，以下规则。

```
[terrain_graphics]
    [tile]
        x=0
        y=0
        type=g
    [/tile]
    [tile]
        x=1
        y=0
        type=c
        [image]
            name=foobar
        [/image]
    [/tile]
[/terrain_graphics]
```

对于以下地形：

```
 __    __
/g \__/g \
\__/m \__/
/g \__/c \
\__/c*\__/
/c \__/c \
\__/  \__/
```

“terrain/foobar.png”将画在坐标(1, 0)（标有星号）格子，而规则中涉及到另一个格子不会画任何图像，尽管(0, 0)对应着规则中满足条件的一条约束。

单个规则中可以绘制多个图像，多个图像的书写格式是<name>字段值写成图像列表，例如：village/dark1-A01.png:200,village/drake1-A02.png:200,village/drake1-A03.png:200。

规则根下的全局[image]

如果[image]放在规则根下，这时可以指定一个尺寸大于 72x72 图像，此时的图像称全局图像，为什么称全局呢？因为在放置这图像时可能会涉及到该规则中所有约束。全局图像可以是大于 72x72 尺寸的大图像（相比于 72x72 小图像），绘画大图像不是只“画”一次原图就够了，而是把大图像拆成数个格子尺寸（72x72）的小图像，然后依次画这些小图像。拆分过程等同图像处理术语中的“裁剪”（Crop）。**注**：拆分过程不会涉及缩放（Scale）。

- 构造规则矩形：构造规则中总合各约束涉及到矩形。
- 图像矩形：图像文件中图像的矩形。

```
[terrain_graphics]
    map="
,  *
*,   *
,  1
*,   *
```

```
,  *"
   ......
[/terrain_graphics]
```

对于这个构造规则它涉及到 7 个格子，它的构造规则矩形尺寸是 180x252。

既然要裁剪就要计算出裁剪矩形参数：左上角坐标、尺寸。参数中尺寸是固定的，即格子尺寸 72x72，剩下就是要计算左上角坐标，根据左上角坐标计算方式不同，放置全局全像有两种方式：原点对齐和中心点对齐。**采用哪种方式的判断标志是[image]块中是否存在有效“center”字段**（有效指的存在 center 字段，并且有有效 x、y 值。）。

图像放置方式一：原点对齐

原点对齐时，构造规则矩形像素坐标(0,0)对应图像矩形的像素坐标(0,0)，矩形中其它像素基于这个对应关系进行平移。以下是一条要放置全局图像的规则，[image]块中没有“center”，因而它是采用原点对齐。

```
[terrain_graphics]
    [tile]
        x=0
        y=0
        type=g
    [/tile]
    [tile]
        x=1
        y=0
        type=c
    [/tile]
    [image]
        name=foobar
    [/image]
[/terrain_graphics]
```

foobar.png 尺寸是 126x108，它看起来是这样的：

```
  _____
| /     \        |
|/   A   \____   |
|\ (0,0) /    \  |
| \_____/  B   \ |
|       \ (1,0)/|
|        \____/_|
```

地形则是和上面一样：

因此，图像 A 部分将画在标有“+”格子，B 则画在“*”格子。

但是，如果希望下面叠加效果，（欲叠加图像尺寸 126x144）

图 4-13 原点对齐

要达到以上说的目的，让看下面这条规则：

```
[terrain_graphics]
    [tile]
        x=1
```

```
        y=0
        type=Ea
    [/tile]
    [image]
        name=shadow-village
    [/image]
[/terrain_graphics]
```

但是，上面规则没法正常工作，甚至不能正确显示(1,0)处村庄！叠加全局图像要满足：**如果规则包含了约束，那么子图像将被画到该约束对应的相应格子处；相应地，子图像若要画在该格子，则规则中必须存在该格子对应的约束。**由于规则中没有(0,1)处约束，因此不会在(0,1)处显示子图像。要让工作需要修改规则。

```
[terrain_graphics]
    [tile]
        x=1
        y=0
        type=Ea
    [/tile]
    [tile]
        x=0
        y=1
        type=Ea
    [/tile]
    [image]
        name=shadow-village
    [/image]
[/terrain_graphics]
```

注意：原点对齐时，构造规则矩形像素坐标(0,0)对应图像矩形的像素坐标(0,0)，因而图像一定是被从原点约束起进行平移，甚至规则中就没定义原点约束。

小结。对比全局、局部图像。

	全局[image]	局部[image]
出现位置	位在[terrain_graphics]根下	位在[tile]根下
[image]中字段	会检查 center 字段	忽略 center 字段
图像	可超过 72x72 尺寸	最大只能是格子尺寸 72x72

图像放置方式二：中心点对齐

中心点对齐时，构造规则矩形中心像素坐标对应图像矩形的中心像素坐标，矩形中其它像素基于这个对应关系进行平移。对各约束来说，各自的约束偏移和全局[image]块中的“center”字段共同决定了它的左上角坐标。

为描述如何计算左上角坐标，让例举在此个 180x252 上叠加尺寸是 144x144 图像，即实现完成图 4-14 的叠加过程。

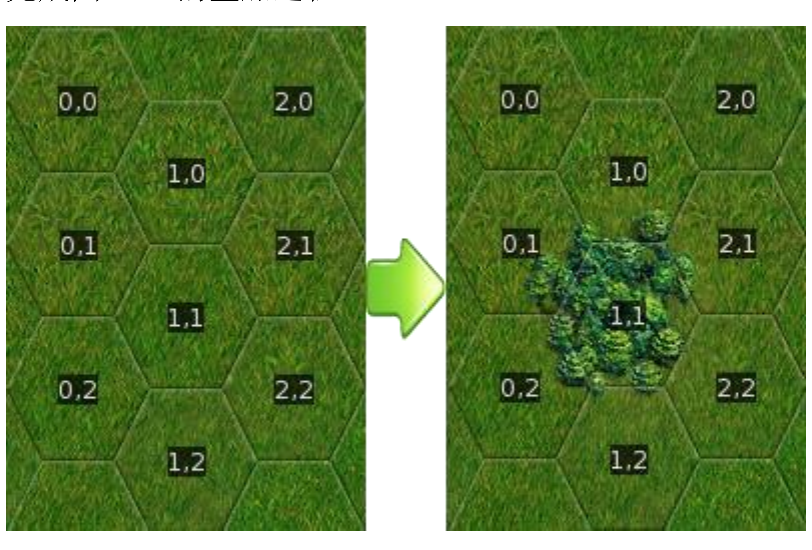


图 4-14 中心点对齐

要计算左上角坐标分两个步骤，1）计算图像尺寸等于构造规则尺寸时坐标；2）用实际图像尺寸时计算出坐标。

当图像尺寸等于构造规则矩形尺寸，依据图 4-14 或图 4-15，由心算就可得出内中各格子坐标。

- (0, 0)：(0, 0, 72, 72)。
- (0, 1)：(0, 72, 72, 72)。
- (0, 2)：(0, 144, 72, 72)。
- (1, 0)：(54, 36, 72, 72)。
- (1, 1)：(54, 108, 72, 72)。
- (1, 2)：(54, 180, 72, 72)。
- (2, 0)：(108, 0, 72, 72)。
- (2, 1)：(108, 72, 72, 72)。
- (2, 2)：(108, 144, 72, 72)。

程序不能靠心算，它须要公式，通过概括要得到以上各格子左上角坐标、尺寸可统一为以下公式。

```
terrain_graphics_x = ((tile_size*3) / 4) * val_.loc_.x;
terrain_graphics_y = tile_size * val_.loc_.y + (tile_size / 2) * (val_.loc_.x % 2);
w = tile_size;
h = tile_size;

注：
val_.loc.x/val_.loc.y：格子对应的约束在构造规则中的偏移。(0, 0)格子，val_.loc.x=0，val_.loc.y=0；(2, 1)格子 val_.loc.x=2，val_.loc.y=1。
tile_size：格子尺寸。即 72。
```

当图像尺寸不等于构造规则矩形尺寸

构造规则矩形尺寸只可能>=图像尺寸。当存在两列时这个不等式一样成立，等于出现在一列时。因为把图像放到构造规则矩形不涉及到缩放，要正确显示图像，只要对齐两个矩形的中心点。构造规则矩形中心点在(1, 1)格子中心，它的像素坐标(90, 144)。图像尺寸是 144x144，它的像素中心坐标是(72, 72)。

要能在(1, 1)格子正确显示图像，由“当图像尺寸等于构造规则矩形尺寸”计算出左上角坐标经过中心偏移坐标后转换后的值。(90, 144)到(72, 72)的偏移值是(-18, -72)。

```
54 - 18 = 36
108 - 72 = 36
```

调整后的(1, 1)格子裁剪矩形左上角坐标(36, 36)。以上计算可用一个公式进行计算：

```
adjust_center_x = png_w / 2 - terrain_graphics_center_x;
adjust_center_h = png_h / 2 - terrain_graphics_center_y;
adjusted_x = terrain_graphics_x + adjust_center_x;
adjusted_y = terrain_graphics_y + adjust_center_y;

terrain_graphics_center_x/terrain_graphics_center_y：构造规则矩形中心像素坐标。这个值并不总是等于(terrain_graphics_w/2, terrain_graphics_h / 2)。
png_w/png_h：图像尺寸。(png_w/2, png_h/2)是图像矩中心坐标。
```

要利用上述公式，MOD 须指定 terrain_graphics_center_x 和 terrain_graphics_center_y。这个字段就是[image]下的 center，像以上这个构造规则就是“center=90,144”，由于构造规则集中大量存在这种矩形，也就意味着集合中存在大量 center=90,144 的构造规则。

小结

- 约束偏移和 center 字段共同决定小图像的裁剪矩形左上角坐标。
- 构造矩形尺寸和图像尺寸不同时，需正确设置 center 字段。

- center 字段值只和构造规则矩形有关，和要绘画的图像尺寸无关。

10.2.6 标志（flag）

在判断本约束是否可以应用前，可以用标志来辅助判断约束是否能被使用。标志的发生作用一般是使用互斥作用。标志就是个可以叠加到映射的字符串。对 MOD 来说，一旦定义标志后它就不可见了，程序用它计算 terrain graphics。

设置标志

[tile]块下的 set_flag 语句用于设置标志。定义 set_flag 时跟后逗号隔开的标志列表。一旦规则匹配格子后这些标志就会设置向这个约束对应的地形格子。

测试标志

[tile]块下的 has_flag 语句用于测试标志。定义 has_flag 时跟后逗号隔开的标志列表。用了 has_flag 后指示地形上必须有 has_flag 后跟的标志后才能匹配所在约束。

[tile]块下的 no_flag 语句也用于测试标志。定义 no_flag 时跟后逗号隔开的标志列表。用了 no_flag 后指示地形上必须没有 no_flag 后跟的标志后才能匹配所在约束。

测试并设置标志

set_no_flag = <flag>：判断匹配时，约束对应的格子不能存在<flag>；可一旦匹配后，约束对应的格子设上<flag>。

[terrain_graphics]下的标志

set_flag、has_flag、no_flag 都可位在[terrain_graphics]块根下，它等同该规则中的所有约束都必须满足这些标志判断。

标志一般用于定义互斥规则。例如如果有一条定义两格的草地地形，另有一条是一格地形，我们不想这同一个格子同时应用这两条规则，这时可以用以下来定义这两条规则。

```
[terrain_graphics]
    [tile]
        x=0
        y=0
        type=Gg
    [/tile]
    [tile]
        x=0
        y=1
        type=Gg
    [/tile]
    [image]
        name=2-tile-grassland
    [/image]
    set_flag=base
[/terrain_graphics]

[terrain_graphics]
    [tile]
        x=0
        y=0
        type=Gg
    [/tile]
    [image]
        name=1-tile-grassland
    [/image]
    no_flag=base
[/terrain_graphics]
```

规则应用规则是谁先定义谁就先使用。如果匹配第一条规则，在对应两个格子就会设置“base”标志。因此第二个条规则就不会匹配两个格子中的任何一个，因为它有“no_flag=base”语句。

10.2.7 使用映射字符串（Map string）

随着约束中的格子数增多，定义多格子地形会变得较容易混淆。为简化定义，引入一个快捷

标注：映射字符串。

terrain_graphics 标签下的一个“map”元素定义一个映射字符串。

映射字符串格式

映射字任串通常是个多行字符串，它符合以下规则：

- 扔掉起始的空行；
- 如果第一非空行以一个空格开始，给它行号 1，否则给行号 0；
- 如果是奇数号的行，扔掉它的初始两个字符；
- 行内剩余字符被分裂为 4 字符块（4-character chunks）（4 字符块并不总是 4 字符，只是最多是 4 个字符）；
- 对每一个四字符块：
  + 如果块是句点，忽略该块；
  + 如果块是由除数字外的其它字符组成，创建新的规则；
  + 如果块是数字，创建新的锚点；
  + 处理一块后列号增加 1；

映射使用的规则等于下面的 WML 块：

```
[tile]
    x=<x>
    y=<y>
    type=<type>
[/tile]
```

<type>是 4 当前块中的 4 字符字符串，以下等式则计算出 x 和 y。

```
x = (lineno % 2) + colno * 2
y = lineno / 2
```

注：“%”是 C/C++中的模操作，“/“是 C/C++中的整数除法操作。

例如，以下两个块具有同样功能。

映射字符串形式

```
[terrain_graphics]
    map="
    g    !c
      mh
    gd"
[/terrain_graphics]
```

原始形式

```
[terrain_graphics]
    [tile]
        x=0
        y=0
        type=g
    [/tile]
    [tile]
        x=1
        y=0
        type=mh
    [/tile]
    [tile]
        x=2
        y=0
        type=!c
    [/tile]
    [tile]
        x=0
        y=1
        type=gd
```

```
    [/tile]
[/terrain_graphics]
```

锚点

正如前面提到的，为了代替指定地形，映射可以使用“锚点”。锚点不是定义真实规则，但设置锚点后，任何[terrain_graphics]内的[tile]就可以使用这个锚点，从而代替偏移中的“x”和“y”。WML 元素 pos=<anchor>中，ahchor 是锚点索引，可取值从 0 到 9。以锚点定义的规则和通常规则没什么两样，除了它的偏移是锚点形式。

使用锚点，一个较让人感兴特色是一个锚写可以被指定多次。当一个[tile]使用这个锚点，它将一次就创建数条约束。例如下面两个块在功能上是一样的：

使用了锚点格式

```
[terrain_graphics]
    map="
      1
    1   1
      2
    1   1
      1"
    [tile]
        pos=1
        type=m
        set_flag=adjacent_to_foo
    [/tile]
    [tile]
        pos=2
        type=m
        no_flag=adjacent_to_foo
    [/tile]
[terrain_graphics]
```

传统格式

```
[terrain_graphics]
    [tile]
        x=1
        y=0
        type=m
        set_flag=adjacent_to_foo
    [/tile]
    [tile]
        x=0
        y=1
        type=m
        set_flag=adjacent_to_foo
    [/tile]
    [tile]
        x=1
        y=1
        type=m
        no_flag=adjacent_to_foo
    [/tile]
    [tile]
        x=2
        y=1
        type=m
        set_flag=adjacent_to_foo
    [/tile]
    [tile]
        x=0
        y=2
        type=m
        set_flag=adjacent_to_foo
    [/tile]
    [tile]
        x=2
```

```
        y=2
        type=m
        set_flag=adjacent_to_foo
    [/tile]
    [tile]
        x=1
        y=2
        type=m
        set_flag=adjacent_to_foo
    [/tile]
[/terrain_graphics]
```

常见映射字符串中锚点转换到偏移值

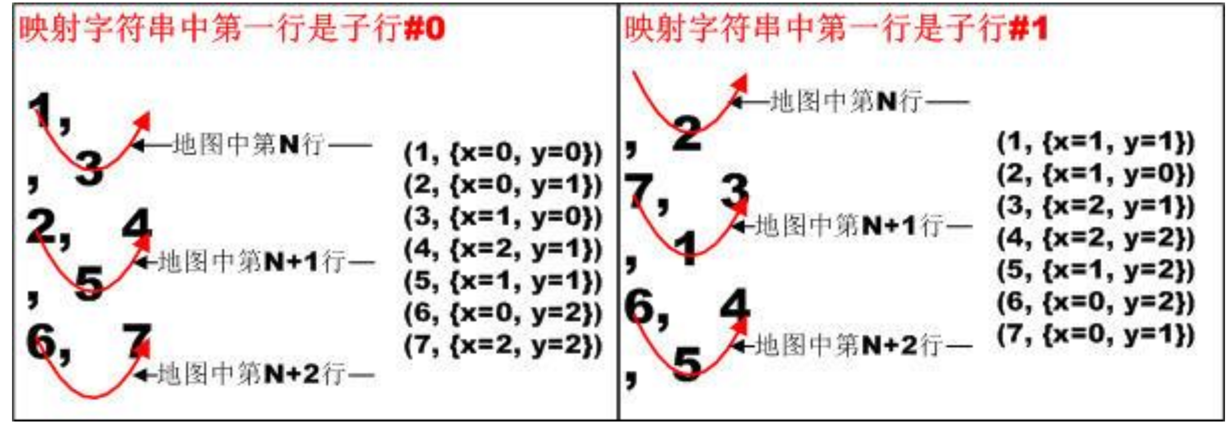


图 4-15 映射字符串锚点到编移

10.2.8 使用旋转模板（rotation templates）

对一些相似规则，A 可能只是 B 的旋转版本。为简定定义这类规则，引入旋转模板。模板使用一个[terrain_graphics]却一下定义最多 6 条规则，每条规则旋转 Pi/3 弧度。

如何定义旋转模板：定义和通常[terrain_graphics]规则，除了该规则含“rotations”字段，这个字段值是 6 个逗号分隔的字符串。

规则或它的约束中，它们的图像名或标志字符串可能含有从模板来的“@Rn”字符串，n 是 0 到 5 数字。

一个模板最多产生 6 条规则，这些规则和模板相似，除了：

- 约束的字符串映射也会随着旋转一个角度，从 0 到 5pi/6。注意，约束的相对位置也会随着旋转，因此一般把“基本”图像看作是垂直的。
  旋转后，所有约束集都被旋转，因此：
  +不能存在负坐标约束。
  +至少存在一个 x=0 的约束。
  +至少存在一个 y=0 的约束。
  满足以上条件把很重要，例如，旋转要对于占用多个格子约束也要适用。
- 对于旋转 0 度约束，模板字符串变量@R0，@R1，@R2，@R3，@R4，@R5，应该被 rotations 字段中的值 r0，r1，r2，r3，r4，r5 进行替换。
- 对于旋转 pi/6 度约束，模板字符串变量@R0，@R1，@R2，@R3，@R4，@R5，应该被 rotations 字段中的值 r1，r2，r3，r4，r5，r0 进行替换。
- 对于旋转 2pi/6 度约束，模板字符串变量@R0，@R1，@R2，@R3，@R4，@R5，应该被 rotations 字段中的值 r2，r3，r4，r5，r0，r1 进行替换。

一个模板约束产生的约束数是由谁决定的？
——rotations 中的值数目。

10.2.9 [image]块中文件名，可动画地形

[image]块中文件名可有三种形式：

- 一个固定文件名。
- 一组文件名。
- 可实现动画的文件名。

1、一个固定文件名

```
[image]
   name=forest/deciduous-summer-sparse.png
   ......
[/image]
```

2、一组文件名

```
[image]
   name=forest/forest/pine-sparse-small@V.png
   variations=";2;3;4;5;6;7;8;9;11"
   ......
[/image]
```

WML 让要[image]块下必须同时有 name 和 variations 字段的方法来找述一组文件名。

假设 get_variations(variant.image_string, variant.variations)来实现把 WML 转为实际值。代入以上例子中 get_variations 输入就是

- variant.image_string: forest/pine-sparse-small@V.png
- variant.variations: ;2;3;4;5;6;7;8;9;11

get_variations 输出（@V 被分号之间的字符串带替）

- forest/pine-sparse-small.png
- forest/pine-sparse-small2.png
- forest/pine-sparse-small3.png
- forest/pine-sparse-small4.png
- forest/pine-sparse-small5.png
- forest/pine-sparse-small6.png
- forest/pine-sparse-small7.png
- forest/pine-sparse-small8.png
- forest/pine-sparse-small9.png
- forest/pine-sparse-small10.png
- forest/pine-sparse-small11.png

采用此处格子表示文件名时，它最后出来的只有一个文件，只是这个文件名是从以上一组文件名中随机的选出的。

3、可实现动画效果的文件名

```
[image]
   name=village/darke1-A01.png:200,village/darke1-
A02.png:200,village/darke1-A03.png:200
   variations=";2;3;4;5;6;7;8;9;10;11"
[/image]
```

注：variations 字段没有发挥作用，写在这里只不过是为了指示要是 name 中有“@V”字符，variations 就会和 name 共同作用决定文件名。[/code]此个[image]创建了一个周期是 600(200*3)毫秒的动画，第一个 200 毫秒显示 village/darke1-A01.png:200，第二个 200 毫秒显示 village/darke1-A02.png:200，第 3 个 200 毫秒显示 village/darke1-A03.png:200，之后它又回到头部，即第 4 个 200 毫秒显示 village/darke1-A01.png:200，依此类推。

要知道更多地形动画细节参考“**7.4 地形动画**”。

10.2.10 用于形成地图的构造规则集

构造规则实现了如何把地图数据（以地形字符串表示）变成一系列玩家看到的位图图像，程序形成地图时逐个扫描数据，针对每个数据应用上相应构造规则，扫描结束后一幅运行时地图也就出来了。用于形成地图的构造规则集指的就是此中的“相应构造规则”。

资源包中的<res>/data/core/terrain-graphic.cfg 存放构造规则集，用“SLG Maker”的“构造规则”页可直观查看各规则顺序。要通过这个集合就画出表征现实地貌的地图，如何编写自然不是那么简单，而且不同人可能还有不同构造框架，以下写些个人认为的通用步骤。

在宏观上分为三步骤：按海拔把各地形分层、绘制基本层、绘制覆盖层。

第一步：按海拔把各地形分层

按海拔由高到低，把常见地貌分山岭、丘陵、平地（包括草原、道路）、水。这个顺序不仅决定了画具体地形图像时的高低呈现，更重要是决定了过渡带如何过渡。过渡带作用是让从一种地形“无缝”过渡到其它地形，它的存在就是要让各地形间过渡不显得突凸，使画出更漂亮地图。但要注意，两地形相邻时过渡带不是相互过渡，它只从海拔高的向低的过渡，图像画在海拔低的格子。举个例子，山岭和水相邻，会出现由海拔高的山岭向海拔低的水过渡，过渡带图像则画在水格子，而不会出现由水向山岭过渡。

同一海拔的不同地形谁向谁过渡问题，像同属草原的 Gg 和 Gll，当它们相邻时是 Gg 向 Gll 过渡还是相反，则视哪种漂亮采用那种，只要遵循只一侧过渡原则，即或全是 Gg 过渡到 Gll、或全是 Gll 过渡到 Gg。另外，同地形码的地形间不会画过渡带。

构造规则中有个“layer”字段，该字段指示具体一格子上各图像叠加次序。要画出漂亮地图，需注意设置各规则的“layer”值，但该值的设置可说和“按海拔把各地形分层”基本无关，如何设置该值参考“第二步：绘制基本层”。

第二步：绘制基本层

基本层概念是相对于覆盖层，是可表征地貌的“底层”地形，像山岭、丘陵、草原、道路、浅水、深水，是地形字符串中“^”左侧部分。覆盖层是覆盖在基本层上地形，像桥梁、村庄、夏季落叶林，是地形字符串中“^”右侧部分。地图上任一格子可以没覆盖层，但必须有基本层。

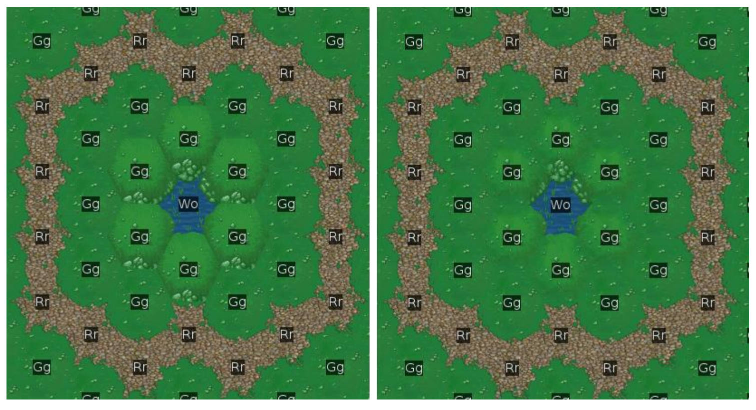
在要使用的图像上，要表示出基本层至少需要两类图像：基本层图像、过渡到其它地形的过渡带图像。像某种风格草原（为多样性，程序往往会支持多种风格草原），它首先需要一张主图像，命名为 grass.png，除此之外有过渡到其它地形的过渡带图像，一正六边形格子会六个方向向外过渡，累计须要 6 张，grass-n.png、grass-ne.png、grass-se.png、grass-s.png、grass-sw.png、grass-nw.png。

如果只画以上七张图像，那是假设了该地形向其它地形过渡时都使用同一风格的过渡带图像，而要最终出来漂亮地图，向所有地形使用同一套过渡带往往是不行的，一种常见情况就是得区分开向水的过渡和向水之外的过渡。

上图草原（Gg）到水（Wot）和道路（Rr）已使用两套过渡带，相比于过渡到水，过渡到道路使用了更宽的过渡带，原因很简单，在水中使过和道路一样宽的过渡带时，那水格子就要被过渡带全盖住，看不出水了。

上面是过渡到水、道路的过渡带解决方案，但这种方案个人认为需要进一完改善，原因是图中和水的过渡不漂亮，要让草原过渡到水呈现出更好效果。那要如何出来更好效果，图中给画的过渡带就那么窄，以这么窄的空间不可能画出漂亮过渡，为此要修改规则，目的是使得水边格子就开始呈现出向水的过渡效果，即图中黑框涉及到的六个 72x72 也归入“过渡带”图像。

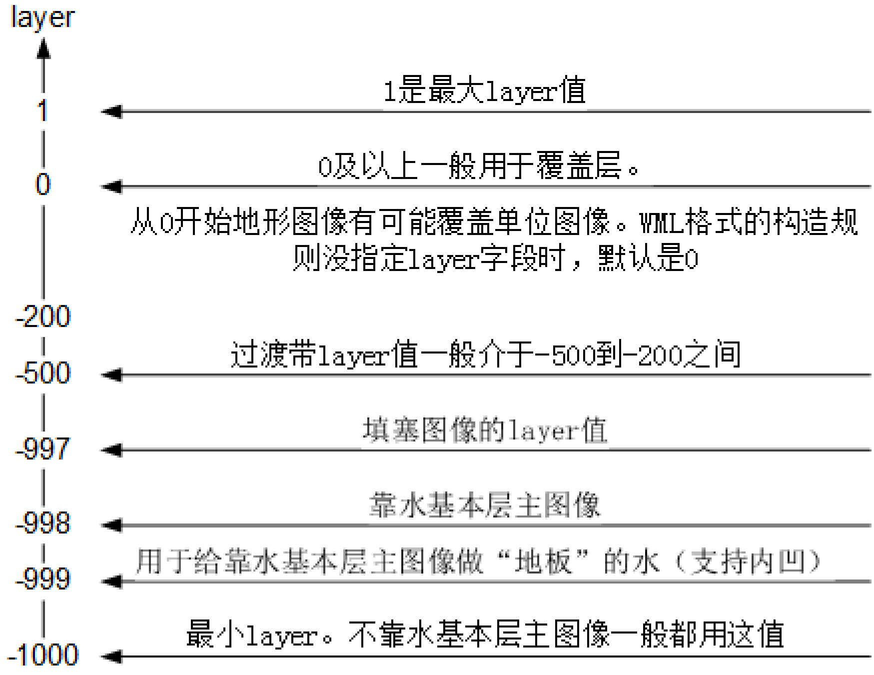
到此主图像须被分为两种，周围没水的（grass.png）、周围有水的（grass-water.png），而且过渡带也要被分为两套，新加一套只针对向水的过渡，包括 grass-water-ne.png 等六张，已有的那套就用于向除水之外地形的过渡。



图中 Gg 使用两套主图像，并使用两套过渡带。上图（左）明显存在个问题，两主图像间的过渡不自然，须要有图像来缝合它们，为此须要再画一套图像（右），这套图像须要三个功能。一：弥合同地形码的两主图像；二：弥合不同地形码、从靠水的到纯陆的过渡；三：要显得邻陆的和邻水的衔接不突兀。此图像是画在靠水格子，使用目的是让出来图像效果和纯陆的一致，为此把它们叫填塞图像，因为有六个方向可能需要填塞，每一图像也须六张。像 grass-stuff-n.png，它是当格子北侧不接水时要填塞的图像，填塞图像总体看去是个顶角在格子中心的三角形。

到此一地形须要 20 张图像，两张主地形、六张水过渡、六张和水之外过渡、六张填塞。

最终出来的地图，它每个格子一般不会只画一张图像。以一个靠水草原（Gg）格子为例，它须一张靠水 Gg 主图像，如果周围存在非水格子，它在那些方向上要画填塞图像，如果周围有比它海拔高地形，像山岭，还须要画山岭到它的过渡带图像。采用一旦冲突、后叠加的会覆盖已叠加的这一常用处理规则，这三种图像在叠加顺序上应该是主图像、填塞图像、过渡带图像。如果在草原格子还存在树，那么须画上树这一覆盖层图像，树的顺序在三种之后。构造规则中的“layer”字段就用于决定叠加次序。



以上定义的 layer 系统，最小值是-1000，最大值是 1。要画出好地图，每套构造规则系统都须事先定义出一套 layer。

第三步：绘制覆盖层

相对于基本层，覆盖层就要简单多了，它基本上就是按着现实中地貌画出各样图像，像森林、白云覆盖的山岭。

10.2.11 构造规则实例：城墙

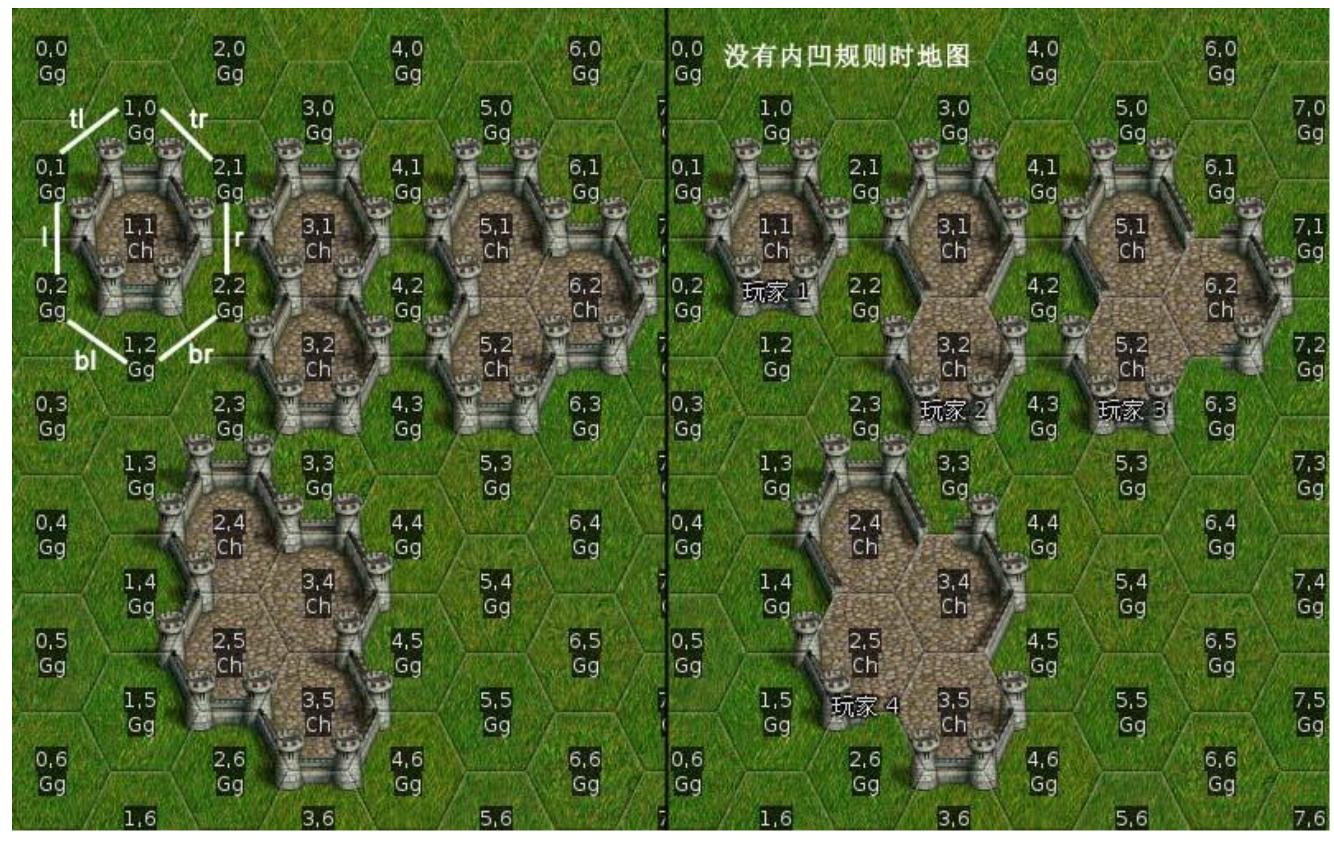


图 4-18，城墙地图

多种城堡。城堡由城墙、地面组成。当相邻格子是城堡时就要把它们连成同一个城堡，在实

现图像显示上说，连通的实质是让在连接处不画城墙。

城墙规则之旋转模板

```
{NEW:WALL   Ch,Cv,Co,Cd   (!,C*,K*)   castle/castle}
```

接下让逐步分析 NEW:WALL 宏。

```
#define NEW:WALL TERRAINLIST ADJACENT IMAGESTEM
    {NEW:WALL_P {TERRAINLIST} {ADJACENT} 100 {IMAGESTEM}}
#enddef
=>
{NEW:WALL_P Ch,Cv,Co,Cd (!,C*,K*) 100 castle/castle
```

NEW:WALL 是 NEW:WALL_P 中的 P=100 的版本，P 指的是[terrain_graphics]中的 probability 字段。probabiltiy=100 指示该规则一旦约束匹配后 100%会应用该规则。NEW:WALL_P 宏定义了两种模板，每模板 6 条规则，也就是说 NEW:WALL_P 一共会生成 12 条规则。

```
#define NEW:WALL_P TERRAINLIST ADJACENT PROB IMAGESTEM
    [terrain_graphics]
        map="
2
,  3
1"
        [tile]
            pos=1
            type={TERRAINLIST}
            set_no_flag=wall-@R0
        [/tile]
        [tile]
            pos=2
            type={ADJACENT}
            set_no_flag=wall-@R2
        [/tile]
        [tile]
            pos=3
            type={ADJACENT}
            set_no_flag=wall-@R4
        [/tile]

        rotations=tr,r,br,bl,l,tl
        probability={PROB}

        [image]
            layer=0
            base=54,72
            name={IMAGESTEM}@V-convex-@R0.png
            variations=";2;3;4;5;6"
        [/image]
    [/terrain_graphics]

    [terrain_graphics]
        map="
2
,  3
1"
        [tile]
            pos=1
            type={ADJACENT}
            set_no_flag=wall-@R0
        [/tile]
        [tile]
            pos=2
            type={TERRAINLIST}
            set_no_flag=wall-@R2
        [/tile]
        [tile]
            pos=3
            type={TERRAINLIST}
            set_no_flag=wall-@R4
        [/tile]
```

```
        rotations=tr,r,br,bl,l,tl
        probability={PROB}

        [image]
            layer=0
            base=54,72
            name={IMAGESTEM}@V-concave-@R0.png
            variations=";2;3;4;5;6"
        [/image]
    [/terrain_graphics]
#enddef
```

NEW:WALL_P 定义的构造规则存在“rotations”，使用了旋转模板。旋转模板基于规则模板进行修改形成新规则，这个修改体现在两个方面。

- 约束偏移。规则内的约束编移要随着旋转角度而变化。
- 标记中的@R0、@R1、@R2、@R3、@R4、@R5，要以当前 r0、r1、r2、r3、r4、r5 值执行替换。

生成规则一：@R0=r0，即@R0=tr 时

约束偏移

此条约束旋转角度是 0，即不执行旋转，它的约束编移就是模板中的约束偏移。具体到图 4-18，这条约束对应“tr”那条线。以下是规则中的映射字符串以及三描点对应的偏移。

```
        map="
2
,  3
1"
```

锚点	约束偏移	地图中坐标
1	(0, 1)	(1, 1)
2	(0, 0)(原点)	(1, 0)
3	(1, 0)	(2, 1)

pos=2 的约束偏移是(0, 0)，它是此条规则的原点。

替换标记

@R0=tr，要使用的 rotations 字段值从 0 到 5 依次是 tr、r、br、bl、l、tl。

“set_no_flag=wall-@R0”变为“set_no_flag=wall-tr”；“name={IMAGESTEM}@V-convex-@R0.png”变为“name={IMAGESTEM}@V-convex-tr.png”。

生成规则二：@R0=r1，即@R0=r 时

约束偏移

此条约束旋转角度是 pi/6，它的约束编移是规则模板以 pos=1 这格子为旋转中心、顺时针旋转 pi/6 后得到约束偏移。具体到图 4-18，这条约束对应“r”那条线。以下是规则中的映射字符串以及三描点对应的偏移。

```
            map="
.,
,  2
1,
,  3"
```

这个映射字符串比模板中的多了一个句点四字块，这只是我为以下叙述方便，即使不写这个四字块此个映射字符串也是合法的。旋转不会增加、减少一个四字块。

锚点	约束偏移	地图中坐标
1	(0, 1)	(1, 1)
2	(1, 0)	(2, 1)

3	(1, 1)	(2, 2)

三个锚点的约束都不是(0, 0)，它们都不是此条规则的原点。规则原点是在映射字符串顶行的句点处，对应地图中坐标是(1, 0)。

替换标记

@R0=r，要使用的 rotations 字段值从 0 到 5 依次是 r、br、bl、l、tl、tr。

“set_no_flag=wall-@R0”变为“set_no_flag=wall-r”;“name={IMAGESTEM}@V-convex-@R0.png”变为“name={IMAGESTEM}@V-convex-r.png”。

生成规则三：@R0=r2，即@R0=br 时

约束偏移

此条约束旋转角度是 2pi/6，它的约束编移是规则模板以 pos=1 这格子为旋转中心、顺时针旋转 pi/3 后得到约束偏移。具体到图 4-18，这条约束对应“br”那条线。以下是规则中的映射字符串以及三描点对应的偏移。

```
            map="
1,
,  2
3"
```

锚点	约束偏移	地图中坐标
1	(0, 0)(原点)	(1, 1)
2	(1, 0)	(2, 2)
3	(0, 1)	(1, 2)

pos=1 的约束偏移是(0, 0)，它是此条规则的原点。

替换标记

@R0=br，要使用的 rotations 字段值从 0 到 5 依次是 br、bl、l、tl、tr、r。

“set_no_flag=wall-@R0”变为“set_no_flag=wall-br”；“name={IMAGESTEM}@V-convex-@R0.png”变为“name={IMAGESTEM}@V-convex-br.png”。

生成规则四：@R0=r3，即@R0=bl 时

约束偏移

此条约束旋转角度是 3pi/6，它的约束编移是规则模板以 pos=1 这格子为旋转中心、顺时针旋转 pi/2 后得到约束偏移。具体到图 4-18，这条约束对应“bl”那条线。以下是规则中的映射字符串以及三描点对应的偏移。

```
            map="
.,
,  1
3,
,  2"
```

锚点	约束偏移	地图中坐标
1	(1, 0)	(1, 1)
2	(1, 1)	(1, 2)
3	(0, 1)	(0, 2)

三个锚点的约束都不是(0, 0)，它们都不是此条规则的原点。规则原点是在映射字符串顶行的句点处，对应地图中坐标是(0, 1)。

替换标记

@R0=bl，要使用的 rotations 字段值从 0 到 5 依次是 bl、l、tl、tr、r、br。

“set_no_flag=wall-@R0”变为“set_no_flag=wall-bl”；“name={IMAGESTEM}@V-convex-@R0.png”变为“name={IMAGESTEM}@V-convex-bl.png”。

生成规则五：@R0=r4，即@R0=l 时

约束偏移

此条约束旋转角度是 4pi/6，它的约束编移是规则模板以 pos=1 这格子为旋转中心、顺时针旋转 2pi/3 后得到约束偏移。具体到图 4-18，这条约束对应“1”那条线。以下是规则中的映射字符串以及三描点对应的偏移。

```
        map="
3,
,  1
2"
```

锚点	约束偏移	地图中坐标
1	(1, 0)	(1, 1)
2	(0, 1)	(0, 2)
3	(0, 0)(原点)	(0, 1)

pos=3 的约束偏移是(0, 0)，它是此条规则的原点。

替换标记

@R0=l，要使用的 rotations 字段值从 0 到 5 依次是 l、tl、tr、r、br、bl。

“set_no_flag=wall-@R0”变为“set_no_flag=wall-l”;“name={IMAGESTEM}@V-convex-@R0.png”变为“name={IMAGESTEM}@V-convex-l.png”。

生成规则六：@R0=r5，即@R0=tl 时

约束偏移

此条约束旋转角度是 5pi/6，它的约束编移是规则模板以 pos=1 这格子为旋转中心、顺时针旋转 5pi/6 后得到约束偏移。具体到图 4-18，这条约束对应“tl”那条线。以下是规则中的映射字符串以及三锚点对应的偏移。

```
        map="
.,
,  3
2,
,  1"
```

锚点	约束偏移	地图中坐标
1	(1, 1)	(1, 1)
2	(0, 1)	(0, 1)
3	(1, 0)	(1, 0)

三个锚点的约束都不是(0, 0)，它们都不是此条规则的原点。规则原点是在映射字符串顶行的句点处，对应地图中坐标是(0, 0)。

替换标记

@R0=tl，要使用的 rotations 字段值从 0 到 5 依次是 tl、tr、r、br、bl、l。

“set_no_flag=wall-@R0”变为“set_no_flag=wall-tl”；“name={IMAGESTEM}@V-convex-@R0.png”变为“name={IMAGESTEM}@V-convex-tl.png”。

旋转模板小结

- 如何确定旋转中心。一般是映射字符串的中心锚点。当垂直方向只有两个时，是下面那个。一旦通过垂直方向不能判定，水平方向尽量不要让出两个。很多模板中把旋转中心定义为

描点 1，那只是为易记，旋转中心和描点值无对应关系。以下是另一种常见的旋转模板映射字符串。锚点 1 是旋转中心。

```
map="
,  2
7,   3
,  1
6,   4
,  5"
```

- 旋转角度是 pi/6 整数倍。
- 旋转方向是顺时针。
- 旋转不会增加、减少一个四字块（锚点）。旋转后形成的规则不会改变旋转中心格子对应的栅格坐标，但会修改所在约束的偏移坐标。

城墙规则之图像素材

以单个城堡来说，它的六段城墙是中心对称；但一座城堡可能不会总显示六段城墙。图 4-18 中的(3, 1)处城堡，为表示和(3, 2)处是连通的，就不显示右下角、左下角这两段，相应的(3, 2)处就不显示左上角、右上角这两段。因而画城墙时需采用一段一段地画，即画六段城墙需六张图像，一张图像对应一段城墙。

确定出要六张图像，接下需确定如何把图像文件中的图像定位到用户最终看到的大地图中，这个定位是由存在于它们之间的纽带构造规则决定的。构造规则放置图像存在两种方式：原点对齐和中心点对齐。对此处城墙来说，六段城墙对应构造规则除了图像文件名不一样外其它可说没区别，以致几乎不可能以中心点对齐去放置，于是只能采用原点对齐。

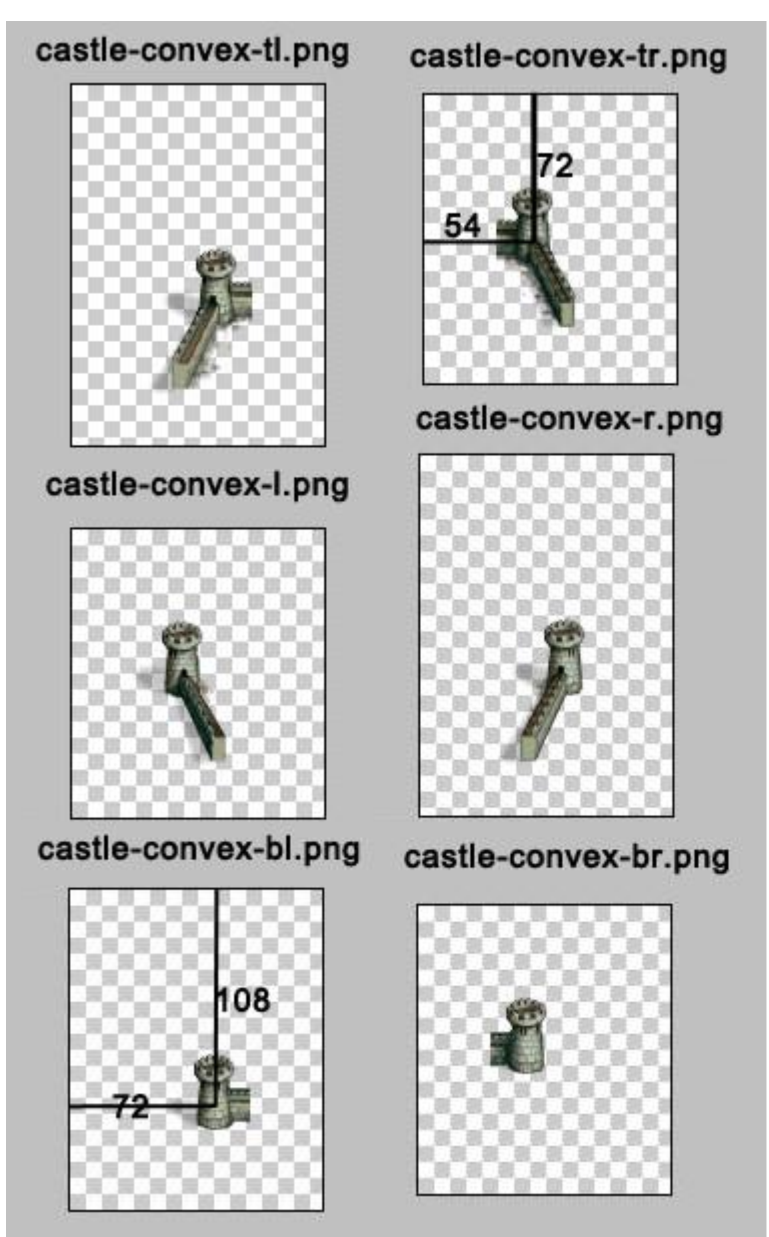


图 4-19 castle-convex

原点对齐要求被放置图像文件的(0, 0)像素对应构造规则原点的(0, 0)像素，其它像素以此对应关系进行平移。图 4-19 显示了组成城墙六张图像。让具体分析两张图像来加深理解城墙构造规则对墙城图像的绘制要求。

右上角那段城墙，对应图像是 castle-convex-tr.png。图像中塔楼和下侧围墙交叉处那像素在图像内偏移是(54, 72)；结合图 4-18，对栅格坐标(1, 1)处城堡，具体到右上角那段城墙，该城墙对应构造规则的原点是地图上(1, 0)坐标格子（参考"**城墙规则之旋转模板中的'@R0=r0，即@R0=tr 时'**"），即放置 castle-convex_tr.png 时，图像的(0, 0)会对应地图格子(1, 0)所在矩形的(0, 0)像素，从该图上可直观计算出塔楼和下侧围墙交叉处、在图像内偏移是(54, 72)。另外，castle-convex_tr.png 是全局图像，它会被拆解到三个约束中形成各约束的子图像，此处三条约束或多或少都会存在不透明的图像数据。

左下角那段城墙，对应图像是 castle-convex-bl.png。图像中塔楼中心点像素在图像内偏移是(72, 108)；结合图 4-18，对栅格坐标(1, 1)处城堡，具体到左下角那段城墙，该城墙对应构造规则的原点是地图上(0, 1)坐标格子（参考"**城墙规则之旋转模板中的'@R0=r3，即@R0=bl 时'**"），即放置 castle-convex_tr.png 时，图像(0, 0)会对应地图格子(0, 1)所在矩形的(0, 0)像素，从该图上可直观计算出塔楼中心点像素在图像内偏移是(72, 108)。另外，castle-convex_bl.png 是全局图像，

它会被拆解到三个约束中形成各约束的子图像，此处三条约束都或多或少会存在不透明的图像数据。

或 castle-convex-tr.png 中塔楼和下侧围墙交叉处在图像内偏移需是(54, 72)，或 castle-convex-bl.png 中塔楼中心点像素在图像内偏移需是(72, 108)，对整图像来说它们只是在对齐要求上两个明显位置而已，对于要绘制出“无缝衔接”城墙需要认真画出六张图像。

城墙规则之匹配条件

外凸（Convex）、内凹（Concave）

NEW::WALL_P 宏定义了两种模板，第一种是外凸，第二种是内凹，需要定义两种的原因是要“完整”显示出六段城墙。图 4-18 右半部分是没有内凹规则时形成的地图（要调试出此个地图可注释掉内凹模板，重新生成 tb.dat），为深入理解让分析下图 4-18 右侧的(3, 1)、(3, 2)两处城堡。对(3, 1)格子来说，根据外凸“br”规则，(3, 2)处是城堡地形，不满足地形匹配条件，因此不应用“br”规则，不会画 castle-convex-br.png；同样原因不应用“bl”规则，不会画 castle-convex-bl.png。换到(3, 2)格子，根据外凸“tr”规则，(3, 1)处是城堡地形，不满足地形匹配条件，因此不应用“tr”规则，不会画 castle-convex-tr.png；同样原因不应用“tl”规则，不会画 castle-convex-tl.png。由于不画这四个图像，(3, 1)、(3, 2)的衔接处城墙出现不“完整”。

要如何修补(3, 1)、(3, 2)，让看去它们是“完整”的城墙？——结合图 4-19 六段城墙素材，仔细看会发现，断掉城墙不能使用外凸规则不画的 castle-convex-br.png、castle-convex-bl.png、castle-convex-tr.png、castle-convex-tr.png 中任何一个，因为它们任何一个都会在水平方向向内画出一段城墙，反而适合画的应该是 castle-convex-r.png、castle-convex-l.png！要让“智能”补画 castle-convex-r.png、castle-convex-l.png，正是内凹规则要做的事。

根据内凹规则，针对(2, 2)格子，相邻(3, 1)、(3, 2)都是城堡地形，于是要画 castle-concave-r.png；同样的，针对(4, 2)格子，相邻(3, 2)、(3, 1)都是城堡地形，于是要画 castle-concave-l.png。

说下为什么叫“外凸”、“内凹”。外凸指画的城墙是由里向外凸出，图 4-18 中(3, 1)格子右上城墙来说，它是因为(3, 1)处城堡地形而导致画出来的，就好像(3, 1)向外凸出来的。内凹指画的城墙是由外向内凹进，图 4-18 中(3, 2)格子右侧城墙来说，它是因为(4, 2)处这个非城堡而导致画出来的，就好像是外部向(4, 2)内凹进来的。对于我们手画城墙来说，直观会想到外凸，但编写构造规则毕竟没手画这么“纯粹”，外凸画不“完整”只好靠内凹去弥补。

次约束中标记（set_no_flag）

不论是外凸还是内凹规则，相比于规则内其它约束，描点 1 处约束不仅充当旋转中心角色，而且直接决定了该规则放置哪段城墙，似乎显得较为重要，于是把它称为主约束，其它约束称为次约束。

标记是作为匹配条件而存在的。对于主约束来说，它放置了一段城墙，于是在约束对应格子置上放了该段城墙标记，这较好理解，那为什么次约束也要存在标记？

再次回顾图 4-18 中(3, 1)格子处放置右上城墙，要避免在(3, 1)处第二次放右上城墙，于是使用“set_no_flag=wall-tr”，以用户观点来说，这个“set_no_flag”语句表示(3, 1)格子处已放置过右上城墙，接下有希望在(3, 1)处放右上城墙的规则都不要让匹配了。我们知道，地图上格子是有相互影响的，在(3, 1)处放置右侧城墙，对和它相邻的(3, 0)格子来说是放了右下侧，对另一相邻的(4, 1)则等于放了左侧，虽然这两个格子就不是城堡地形，可内凹规则却是要以主约束不是城堡做为匹配条件！再回到“@R0=r0”时外凸规则，对 pos=2，set_no_flag 字段值是“wall-br”，pos=3，set_no_flag 字段值是“wall-l”，这两个正好满足了(3, 0)右下侧、(4, 1)右侧这两个标记设置。

次约束中标记体现了地图中一段城墙如何影响另外两个格子（次约束对应）的匹配判断。

10.3 生成地图

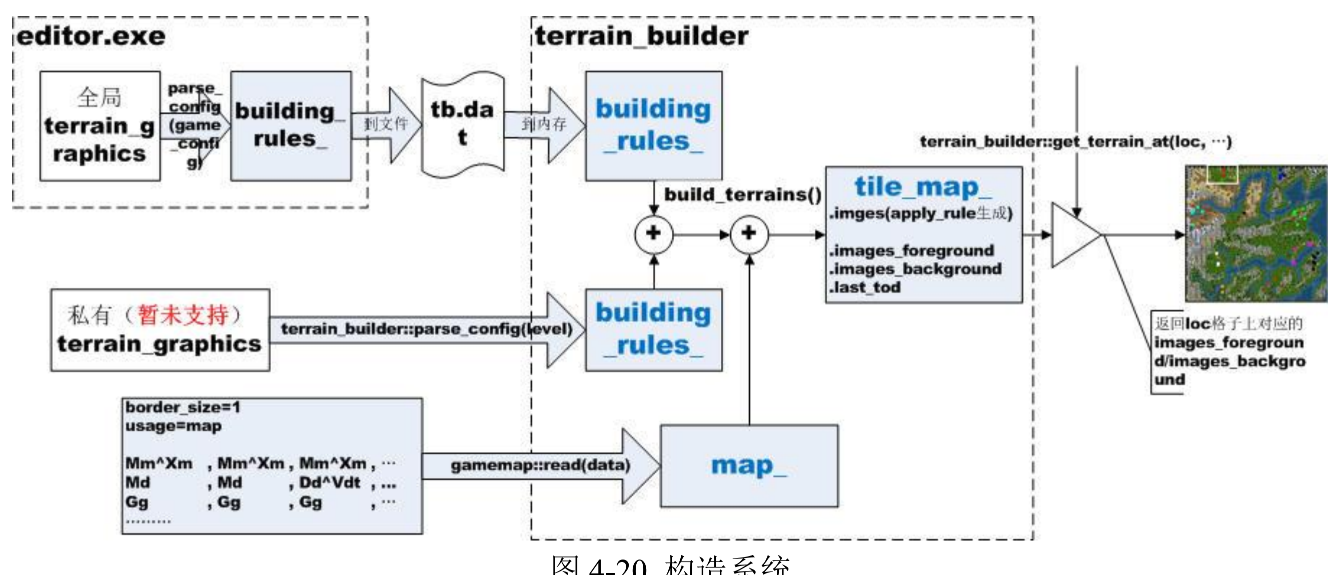


图 4-20 构造系统

生成地图中的地图是用户最终看到的以图像表示了的地貌地图，也叫运行时地图。在程序中，地图构造器（terrain_builder）执行整个生成过程，在执行具体生成（build_terrains()）时，它有两个输入：构造规则集（building_rules_）、字符串表示的地图（map_），生成结果存放在变量 tile_map_，tile_map_变量表示了运行时地图。

10.3.1 约束匹配

在具体叙述 build_terrains()前让说下约束匹配。约束匹配是指判断一条约束是否满足条件的过程。对构造规则内部组成，以组成单位分，它分成数条约束；以内容分，它分成判断条件和满足条件后执行的操作。对于判断条件、满足条件后执行的操作则可能被分散在各约束中，也可能直接位在规则根下，像全局图像操作。但不论全部位在约束也好，还是一部分在约束一部分在全局也罢，它们有个共同特点：条件或操作要发挥作用要么全部生效，要么啥也不做。换句话说，针对条件，能否满足此规则中条件是能否满足当中各约束中条件、全局条件的与操作。

约束中条件具体有哪些内容：地形和标志。地形是类似“type=Wo”，“type=(!,Gg)”的“type=<地形列表>”字符串，标志则是字段名是 has_flag、no_flag 字符串。在如何判断满足上，标志判断是个简单的字符串存在/不存在判断，而地形判断则要稍显复杂。要了解如何执行地形判断，让首先说下程序表示地形的数据结构：struct t_terrain。

C/C++中地形：struct t_terrain

- 地形标识：字符串表示的地形，经常使用场合是在 MOD 中表示地形（列表），像构造规则中的 type 值。它表示两层地形的语法是“基本层地标识^覆盖层地形”，像 Hh（丘陵），Hh^Fds（丘陵^夏季落叶林）。
- 地形值：整数值表示的地形，使用场合是 C/C++程序。类型就是 t_terrain，t_terrain 表示两层地形的语法就是该 struct 中两个字段：base（基本层层值）, overlay（覆盖层层值）。

t_terrain 以整数值方式表示了书写构造规则时的字符串地形标识。以下是地形标识和地形值之间关系。

1. 标识中的脱字符“^”用于分隔基本层和覆盖层，基本层对应 t_terrain 中 base，覆盖层对应 overlay。
2. 一个层标识最多四个 ASCII 字符，因而在表示层标识时层值用了个四字节的 uint32_t。
3. 层值从左到右翻译层标识，越靠左的字符越放在层值高位上。当层标识不满四字符时，不足

的部分以 0x00 去补。

地形标识	地形值
_off^_usr	(0x5f6f6666, 0x5f757372)
*^_fme	(0x2a000000, 0x5f666d65)

注：对于急于要知道生成地图的可略过接下部分，跳去看“判断地形匹配”。

```
t_terrain string_to_number_(std::string str, int& start_position, const t_layer filler)
```

\ 函数把地形标识转换为地形值。

- str：要转换的地形标识。
- start_position[OUT]：如果 str 中间含有空格，空格前字符串转换到的整数值。例如 str=2 Gg，返回的 start_position 值是 2；str=9 Gg，返回的值则是 9。为什么要存在 start_position 呢？——这是为让地形支持其它功能，像标识玩家位置。像以上的 2 Gg 就表示玩家 2 在此个格子，至于坐标则可由它在 map 文件中位置计算，str=9 Gg 则玩家 9 在此个格子。start_position 字面解释不是地形标识在 str 中开始位置，而是玩家初始所在位置。
- filler：地形标识中找不到覆盖分量时使用的覆盖层值。如果提供的 filler==WILDCARD，并且 str 中的基本层值是 NOT 或 STAR，则覆盖层值是 NO_LAYER。

执行逻辑

1. 去除掉 str 前缀、后缀的空格、TAB 符。
2. 如果中间有空格，去除空格及前面字符。只会查一次空格，这个查会计算出 start_position。
3. 如果在字符串存在脱字符，脱字符前认为是基本层，脱字符后是覆盖层；否则给的 str 都认为是基本层，filler 作为覆盖层。

判断地形匹配

要叙述如何判断地形匹配，让进入源码级调试。

1. 在 kingdom 工程打开 terrain_translation.cpp，在 terrain_matches 函数内设断点。
2. 运行时选“Debug”——“Starting Debug”。
3. 进入标题屏幕后“战役”，“群雄争霸”，一路“确认”，会触发断点。

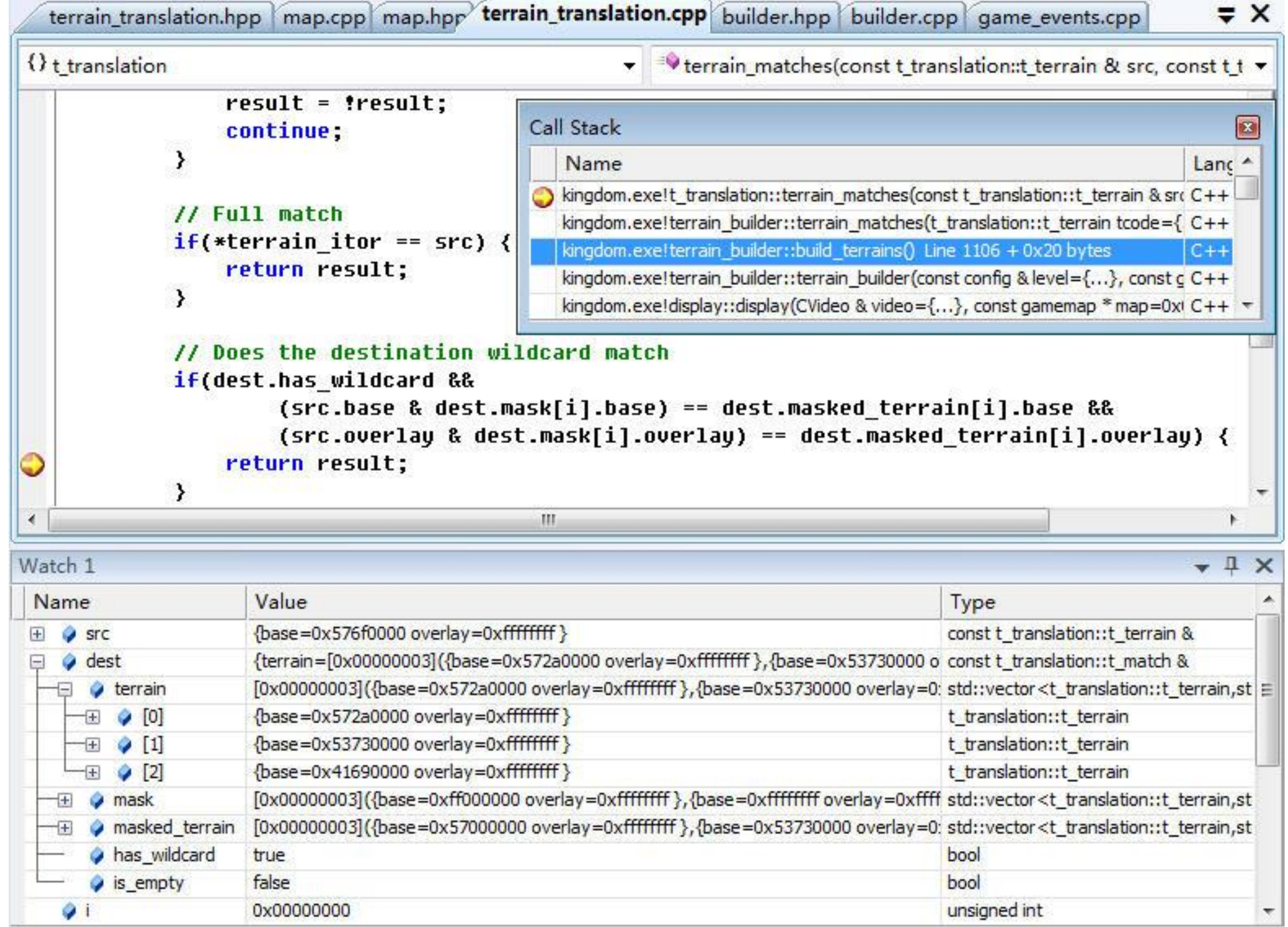


图 4-21 t_translation::terrain_matches

图 4-21 显示如何判断一次地形匹配。对判断地形匹配这个操作，用更容易理解的话来说是判断某一地形是否属于某个地形列表。例如要判断某一地形是否匹配“type=(!,Gg)”，等于是判断该地形是否出现在“(!,Gg)”这个地形列表中。具体到图 4-21，函数参数 src 指示“某一地形”，dest 指示“某个地形列表”，从 Watch 中则可猜出要判断是啥内容。

- src：t_terrain 类型。翻译成地形标识是“Wo”。
- dest：t_match 类型。翻译成地形标识列表是“W*^,Ss^,Ai^”。

dest 是个列表，它的长度是 3，表示列表中有三种/类地形，具体来说这个地形列表可匹配没有覆盖层，基本层是“Ss”、“Ai”和所有以“W”字符开始的地形。此次来判断是否匹配的地形是“Wo”，满足“W*”，因此在 i=0 时就满足条件，向上返回 true。

terrain_matches 是判断地形匹配函数，由于地形复杂性，像被拆为基本层、覆盖层，层内支持通配符“*”、非符“!”，地形列表可能较长，它使得判断地形匹配比判断标志要花去多得多的时间。

小结

- 约束匹配是指判断一条约束是否满足条件的过程。约束匹配包含地形匹配和标志匹配，判断地形要比判断标志花去多得多的时间。
- 标志匹配是个简单的字符串存在/不存在判断；地形匹配是判断某一地形是否属于某个地形列表。

注：对于急于要知道生成地图的可略过接下部分，跳去看“4.3.2 减少规则匹配判断次数”。

地形列表：t_translation::t_match

- terrain：t_terrain 数组。
- mask：t_terrain 数组。
- masked_terrain：t_terrain 数组。
- has_wildcard：地形中是否有通配符。只要一个地形分量中一个有通配符 has_wildcard 就是 true。覆盖层是默认值时并不表示 has_wildcard 是 true，默认值是 0xffffffff，而通配符是要求是值中必须有一个字节 0x2a（“*”）。由于 terrain 是数组，只要这数组中有一个 t_terrain 有通配符 has_wildcard 就是 true。

A：为什么 get_layer_mask_遇到参数 t_layer 中有*时，它返回的是 0x00 而不是 0xff?
Q：地形掩码值的确是 0xff，也就是说要是没有“*”，它返回的是 0xffffffff。但是遇到有*时，它在那里止住返回了 0x00。
*---：0x00000000
-*--：0xff000000
--*-：0xffff0000
---*：0xffffff00

值和 0x00 与时，它的值都是 0x00，既然两个都是 0x00，这意味着任何地形与它都是匹配的，也就是实现了通配的意义。

A：掩码值是 0xffffffff 和 terrain.overlry 默认值是 0xffffffff，这两者有关系吗？
Q：掩码值必须等于默认值。

对地形来说，地形有两分量，基本层分量默认值是 0，覆盖层分量默认值是 0xffffffff，程序内就把 0xffffffff 定义为一个叫 NO_LAYER 宏。NO_LAYER 字面解释就是没有为该地形相应层定义分量。由于定义地形标识时一定会定义基本层，以上所说的相应层一般就特指覆盖层。

基本层默认值是 0。

A：M*是否能匹配基层是 M*、而且存在覆盖层的地形？
Q：不能。M*形成的地形码中覆盖层是 0xffffffff，即 NO_LAYER，这个值只能匹配无地形。要写一个匹配基层是 M*、覆盖层则可能存在也可能不存在的可用“M*^*”。后者返回的掩码后覆盖层是 0x00000000，它与任何值相与都是 0x00，结果匹配。

10.3.2 减少规则匹配判断次数

知道什么是地形匹配后，让进入 build_terrains()。

1. 在 kingdom 工程打开 builder.cpp，在 build_terrains 函数内设断点。
2. 运行时选“Debug”——“Starting Debug”。
3. 进入标题屏幕后“战役”，“群雄争霸”，一路“确认”，会触发断点。

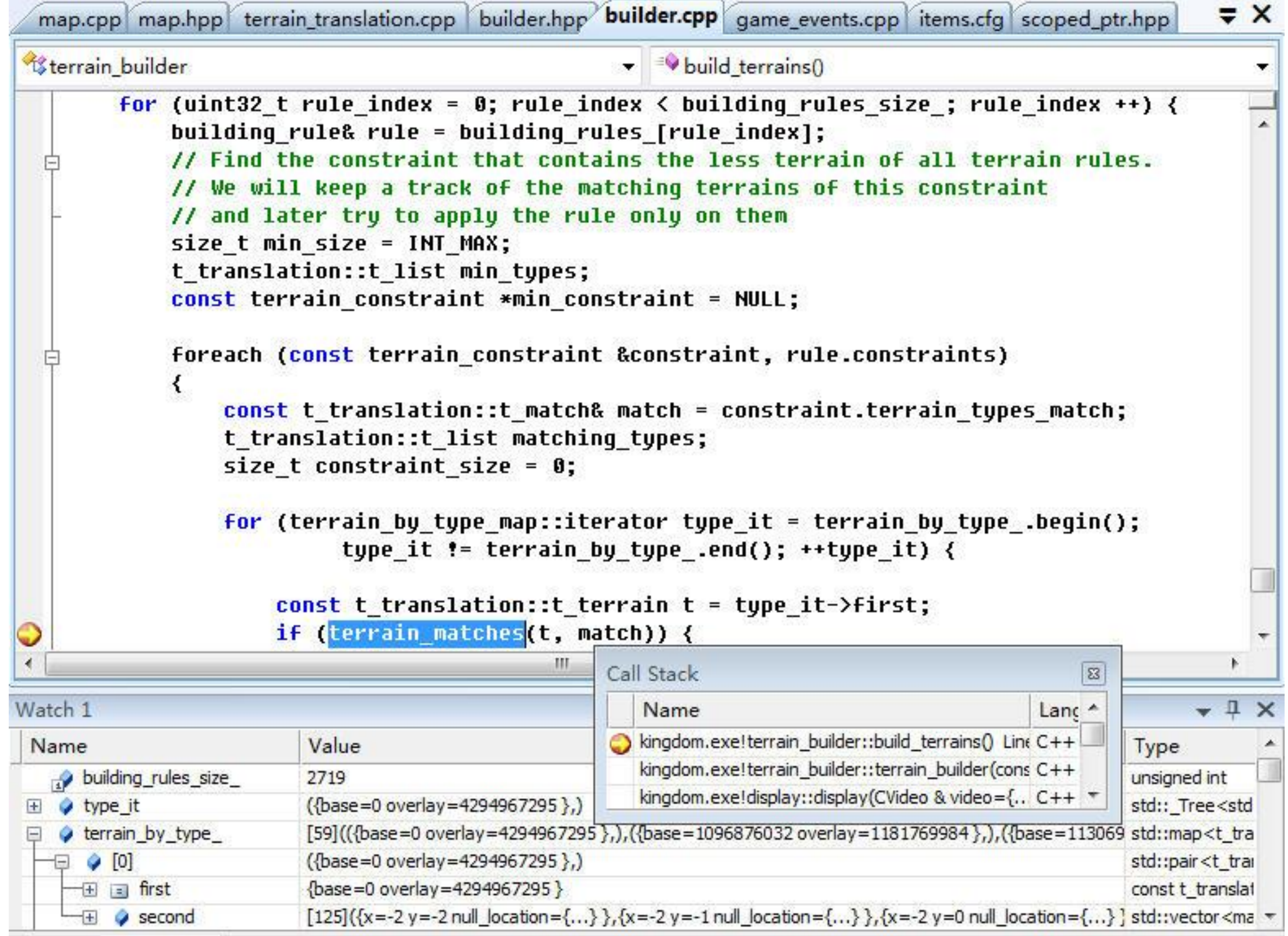


图 4-22 地图构造器（terrain_builder）如何生成运行时地图

图 4-22 显示地图构造器（terrain_builder）如何生成运行时地图。让暂时抛开代码，假设下换是你，你会如何设计程序：根据构造规则（building_rules_）和以 t_terrain 表示的地图（map_）生成运行时地图。一种较快可想到办法：分成两级循环，一级循环是逐条构造规则，二级循环是按一定次序地图上逐个格子（例如从左到右然后转到下一行继续从左到右），一个格子一个格子地执行判断规则匹配，匹配成功的则执行规则约定操作。如此一条条规则应用下去，一张地图就出来了。

强调下当中使用的匹配是规则匹配，规则匹配是指判断一条规则是否满足条件的过程。在约束匹配时已有说过，“能否满足此规则中条件是能否满足当中各约束中条件、全局条件的与操作”。举个例子，正在应用的是 NEW:FOREST 宏中的第一条规则，而轮到要判断的格子是图 4-2 中的(1, 1)，根据构造规则，(1, 1)有个相关格子是(1, 0)，虽然格子(1, 1)是满足条件了，但该规则要涉及到的相关格子(1, 0)的地形是“Gg”，不满足“type={ADJACENT}”这个地形条件，于是不使用该规则。对于它的下条构造规则，即 NEW:FOREST 宏中的第二条，它只有(1, 1)这么个约束，满足该约束就意味着满足该规则，由于满足了条件于是在(1, 1)上将该使用该规则，使用的具体操作是在(1, 1)格子画上 forest/deciduous-summer-sparse/2/3/4/5/6/7/8/9/10/11.png 中的一个，同时在该格子置上 forest 标志。

粗略估计下此个生成过程花在判断匹配上的计算量。匹配判断包括标志匹配和地形匹配，判断地形要比判断标志花去多得多的时间，只是粗略估计时可略去标志匹配。那花在地形匹配上时间是多少？以 3000 条构造规则（图 4-20 是 2719 条），一条构造规则平均 3 条约束，一个 150x100 地图计算，那它要进行 3000 x 3 x 150 x 100 次地形匹配判断，以每次地形匹配 1 微秒计算，当中就要花去 135 秒！——实际使用中每次地形匹配应该不会达到 1 微秒，但由于次数基数如此之大，这多少都会让相乘之后的总和时间变得较为恐怖，为此得有更好的生成逻辑来

减少花在判断上时间，而地形匹配（图 4-19 的 terrain_matches）往往已无法优化，这个减少就只能靠减少判断次数。

优化靠事物间的相关性。规则和规则之间，它们可说就没相关性，无法优化。对于同一规则下 loc1、loc2 两个格子之间，乍一看也没啥相关性，但如果换个角度，让改去看一条规则中各约束能匹配格子数，**不同约束时这个格子数往往是不同的，判断规则匹配时最多只须最小格子数次数，这个最小格子数往往要（远）小于地图格子数，从而就减少了判断次数。**举 NEW:FOREST 中第一个规则作为例子，规则有 2 条约束，约束 1 地形条件是“type={TERRAINLIST}”，假设满足的有 150 个格子，约束 2 地形条件是“type={ADJACENT}”，假设满足的有 160 个格子，至此对这条规则来说，生成过程中最多只须执行约束 1 中的规则判断 150 次。当然，约束 1 和 2 由于地形条件不同，约束 1 的匹配格子集可能不是约束 2 匹配子集，但对规则来说是要同时满足，数值小的那个就可做为最小判断数，至于约束 1 匹配时是否同时满足约束 2，那就须要执行进一步规则匹配才能知道。为叙述方便，在此把拥有最小匹配格子数的约束 1 称为该规则的关键约束。图 4-20 中 build_terrains 引入关键约束来生成地图，这个生成过程可归纳三个步骤。

步骤一：计算此地图中地形集

地形集，对应图 4-20 中 terrain_by_type_变量，它是根据 t_terrain 地图数据形成的一个 map 结构，map 的 first 是 t_terrain，map 的 second 是地图中地形 t_terrain 的坐标集 std::vecotr<map_location>。map 长度就是地图上地形种类数，图 4-20 指示指个地图有 59 种地形，具体到 t_translation::NONE_TERRAIN(base: 0x0, overlay: 0xffffffff)地形则有 125 个格子。构造规则集对所有地图是通用的，terrain_by_type_则是特定于地图的变量。

步骤二：针对一条规则，找到关键约束

逐条计算该规则中约束，看它能匹配多个少格子，最小格子数的那条就是关键约束。匹配只须地形匹配，但要注意地图中只要增加一种地形，便会极大增加此处匹配判断次数。以 3000 条构造规则、一条构造规则平均 3 条约束计算，只要地图中增加一种地形便会在此处增加 9000（3000x3）次判断（数值和该地形在地图中出现次数无关）。为减少接下步骤三时间，在寻找关键约束过程中应该形成关健约束时对应的地形列表 min_types，min_types 值类似{沙地、绿洲}，它和 terrain_by_type_共同决定出要进行规则匹配的格子；以及指向关键约束的指针 min_constraint，因为一旦有了 min_constraint，接下要执行的规则判断发现要判断的约束等于 min_constraint 就可立即认为通过，从而减少判断时间。

步骤三：对关键约束中所有格子进行规则匹配，通过的使用规则

一级循环上逐个类型集（min_types 中），二级循环上逐个格子（terrain_ty_type_中），执行当中所有需要格子上规则匹配，在这规则匹配过程中可能会出现规则不适用该格子（满足关键约束不代表能满足其它约束），但只要通过规则匹配了，那就可以执行在该格子上应用规则。为提高效率，既然已进能入此处就代表关键约束是肯定通过了，那在完全判断过程就没必要“认真”判断关键约束（参考 min_contraint）。

此种方法是通计算关键约束，通过关键约束来减少规则判断次数，从而实现优化地图生成过程。优化到底到达到什么程度，就看减少判断次数节省的时间和引入关键约束导致多花的时间，它们之间的差值。

10.3.3 运行时地图

build_terrains()在得出关键约束匹配的格子集后，对集合中每个格子判断规则匹配，一旦规则匹配通过后，就会在该格子上（包括规则会涉及到格子）使用规则定义的操作，像画图像、设置标记。这些设置了的图像、标记就会反映在构造器（terrain_builder）用于表示运行时地图的 tile_map_变量，这个设置是怎么个过程，以及图像、标记是如何存储的，让进入源码级调试。

1. 在 kingdom 工程打开 builder.cpp，在 apply_rule 函数内设断点。

2. 运行时选“Debug”——“Starting Debug”。
3. 进入标题屏幕后“战役”，“群雄争霸”，一路“确认”，会触发断点。

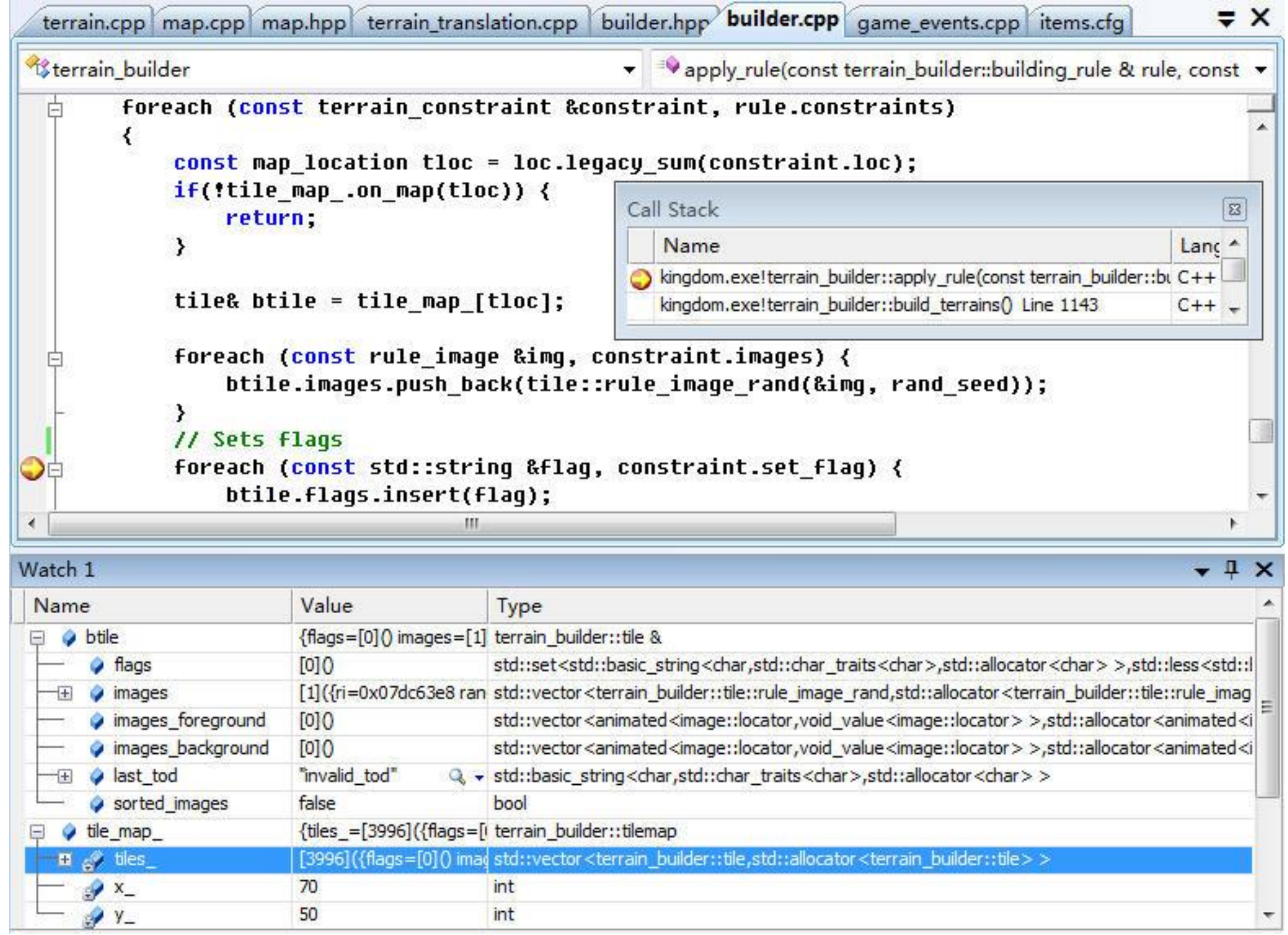


图 4-23 显示构造器如何把图像、标记设置到 tile_map_变量

图 4-23 显示构造器如何把图像、标记设置到 tile_map_变量。tile_map_可认为就是个 std::vector<tile>，当中一个单元 tile 对应一个地图上格子。在该格子使用规则操作，等同修改该格子对应的 tile 值（tile 是个结构，修改它的值等于修改内中字段）。另外，构造规则在由 WML 的[terrain_graphics]块转换为 C/C++代码 building_rule 结构时，已经把全局操作拆散到各约束中，因而到此时要应用的操作已全在约束。而对一条规则来说，不同约束肯定对应不同格子，由此让小结下图 4-23 中设置操作步骤。

- 调用 loc.legacy_sum，由(0, 0)处格子坐标和偏移值计算出该约束对应的格子坐标 tloc。
- btile = tile_map_[tloc]，取出该约束要操作到的 tile 变量。
- btile.images.push_back(...)执行设置图像操作。设置时用了 foreach，也就是说一个约束内可能存在多个[image]块（这些块可能是 WML 书写时就存在约束中，也可能是全局的被拆散到该约束中），而 images 中一单元对应一个[image]块。
- btile.flags.insert(flag)执行设置标记操作。后续的匹配判断可能会需要该标记。

注

- legacy_difference：偏移处格子坐标和偏移值计算出(0, 0)处的格子坐标；
- legacy_sum：(0, 0)处格子坐标和偏移值计算出偏移处的格子坐标；

让注意下图 4-23 中 tile_map_的 tile 数量。tile_map_的 x_、y_指示了不包括边界的地图尺寸，但地图是肯定要存在边界的，为支持边界 tile 数量其实是(x_ + 4) * (y_ + 4)，即在四周各多

两个格子，此个实例就是 3996（74x54）。由于存在这个固定编移，当上层用格子坐标来取 tile 时，要考虑到这个偏移。

```
terrain_builder::tile&          terrain_builder::tilemap::operator[](const map_location &loc)
{
   return tiles_[(loc.x + 2) + (loc.y + 2) * (x_ + 4)];
}
```

为什么两方向上都要“+2”，参考“10.3.3 运行时地图中**被隐藏的行、列**”。

tile 中的标记（flags）和图像（images），标记只是用于后续规则匹配判断，对运行时地图是没用的，真正有用的只有图像。images 类型是 std::vector<rule_image_rand>，那它是如何变成用户看到的运行时图像？接下让深入 images。

std::vector<rule_image_rand> images

WML中块 C/C++中类型/数组
[terrain_graphics] → building_rule/
[tile] → terrain_constraint/constraint_set
[image] → rule_image/rule_imagelist
[variant] → rule_image_variant
name = ... → animated<image::locator>
variantions = ...
[/variant]
[/image] *rule_image_rand*
[tile]
[/terrain_graphics]

图 4-24 WML 和 C 代码对应关系

从用户角度说，运行时图像就是张“完整”大图像；而从 C/C++代码说，要出来大图像，它是通过许多小尺寸图像拼凑而成，在拼凑过程中即使一个格子都可能要叠加若干个图像。“4.1.2 地图上地形图像形成过程”就有一个格子上叠加了 14 个图像。另一方面，程序为实现动态地形，像流动的深水、转动的风车，在地形上使用了动画（参考“第七章 动画”中“7.5 地形动画”），而图像，其实是 t 时刻在那些个动画的各自时间轴上采样到的那一帧！举个例子，“4.1.2”中一个格子叠加 14 个图像就表示了在那格子上存在 14 个动画，t 时刻在那些个动画的时间轴上采样到的正是“4.1.2”叙述的那 14 帧。

tile 是如何把需要的 14 帧返回给上层的？（要理解 get_terrain_at 如何融入大地图渲染系统参考“第七章 动画”中“7.5 地形动画”）

```
const   terrain_builder::imagelist   *terrain_builder::get_terrain_at(const map_location &loc, const std::string &tod, const TERRAIN_TYPE terrain_type)
{
   ......
   const   imagelist&   img_list   =   (terrain_type   ==   BACKGROUND)   ? tile_at.images_background : tile_at.images_foreground;
   ......
   return img_list;
}
```

terrain_builder::get_terrain_at 返回的是 terrain_builder::imagelist，它的更易懂定义是 std::vector<animated<image::locator>>，它指示了在 loc 这格子上存在的 14 个动画，再加上时刻 t，大地图渲染系统就可定位出 t 时刻对应的 14 帧。知道结果让来溯源，要返回的 img_list 来自 tile 中的 images_background（背景）或 images_foreground（前景），看下这两个字段是如何被赋值。

```
void terrain_builder::tile::rebuild_cache(const std::string& tod, logs* log)
{
    ......
    foreach(const rule_image_rand& ri, images) {
        bool is_background = ri->is_background();
        imagelist&   img_list   =   is_background   ?   images_background   :
images_foreground;
        foreach(const rule_image_variant& variant, ri->variants) {
            unsigned int rnd = ri.rand / 7919;
            const   animated<image::locator>&   anim   =   variant.images[rnd   %
variant.images.size()];
            ......
            img_list.push_back(anim);
            ......
        }
    }
}
```

外层 foreach 要迭代的 images 就是图 4-23 中形成的 images，也就是在这小节中要深入的 std::vector<rule_image_rand> images；而 variant.images 指的是某个[image]块内存在的动画集。images 是个 std::verctor，外层 foreach 作用是迭代此个 std::vector，内层 foreach 作用是迭代 rule_image_rand 内部的 variants 变量。以下是和此代码相关的几个结论（理解时结合图 4-24）。

- 一个约束（[tile]）内可能存在多个[image]块（这些块可能是 WML 书写时就存在约束中，也可能是全局的被拆散到该约束中），而 images 中一单元对应一个[image]块。（此结论注释图 4-23 已表述过，重抄是为了让小结更全面）。
- 一个[image]内动画或全属于背景，或全属于前景。
- 一个格子可同时有效数个[image]块，一个[image]块对应格子中一个要有效动画。有效动画指的是在渲染该格子时会让用户看到它当中图像的动画。
- 当前实现的 ri->variants 变量长度不是 0 就是 1。那 ri->variants 长度何时将大于 1？大于 1 时对应的 WML 是什么格式？

```
[image]
    name=forest/forest/pine-sparse-small@V.png
    variations=";2;3;4;5;6;7;8;9;11"
    random_start = yes
    [variant]
        name = ...
        variantions = ...
        random_start = ...
        tod = ...
    [/variant]
[/image]
```

以上[image]块形成的 ri->variants 长度将是 2，一个是[image]根下的 name，一个是[vairant]块中的 name。[image]往往都可形成根下的那个 variants，而不存在[variant]块，也就导致 ri->variants 长度是 1。既然之上有写实现的 ri->variants 变量长度不是 0 就是 1，也是说到现在已写的构造规则中都没有[variant]块。[image]中可以不止一个[variant]，每多一个[variant]就会让 ri->variants 长度加 1。为方便接下描述，把[image]根下的也称一个[variant]块。

- 即使一个[variant]中可能有一组动画，但一个[variant]块只会有效一个动画。如果[variant]内是一组动画，则随机选择一个。variant.images 这个 std::vector 变量长度指示了一组文件名中的文件名数，同时也是此个[variant]块中动画数。
- name、variantions 会形成一组文件名，一组文件名会形成一组动画，文件名如何形成动画参考“terrain_builder::load_images(building_rule &rule)”。

到现在再看下 rule_image_rand 这个结构。

```
struct rule_image_rand {
    rule_image_rand(const  rule_image*  r_i,  unsigned  int  rnd)  :  ri(r_i),
rand(rnd) {}
    const rule_image* operator->() const {return ri;}
    /** sort by layer first then by basey */
    bool operator<(const rule_image_rand& o) const {
        return  ri->layer  <  o.ri->layer  ||  (ri->layer  ==  o.ri->layer  &&
```

```
ri->basey < o.ri->basey);}
    const rule_image* ri;
    unsigned int rand;
};
```

rule_image_rand 作用是指示在渲染大地图时该格子上要叠加哪些图像，因而在图 4-24 构造规则拓扑结构中它对应的是 rule_image，即[image]块。**rule_image_rand 存的是 rule_image 指针**，扩展时（tile::rebuild_cache）须使用实际存储了[image]块数据的 building_rules_。

rule_image_rand 中 rand 意义是它可提供一个随机数，使得当[image]中存在多个动画时，可随机选择一个动画。

同一格子既然可叠加数个图像，不可避免要存在交叉覆盖问题，解决交叉覆盖一般原则：最底下图像最先画。rule_image_rand 要负责调整此格子所有要画图像次序，“operator<”就用于调整次序。rebuild_cache 得到格子上动画前会检量所有动画是否排序过，如果未排序会先调用 std::stable_sort(images.begin(), images.end())，一旦排序后就会把 sorted_images 置为 true。对于图像交叉覆盖问题，大地图渲染系统也实现了一个图像排序模块，但那个模块不会排序由 tile 得出的图像，tile 把内中图像加入到那个排序模块时对所有图像用了相同的层值、x、y。

```
void display::draw_hex(const map_location& loc)
{
    ......
    drawing_buffer_add(LAYER_TERRAIN_BG, loc, xpos, ypos,
        get_terrain_images(loc,tod.id, image_type, BACKGROUND));

    drawing_buffer_add(LAYER_TERRAIN_FG, loc, xpos, ypos,
        get_terrain_images(loc,tod.id,image_type, FOREGROUND));
    ......
}
```

小结 terrain_builder::tile 中各字段意义。

- flags：该格子上设置了的标记。后续的匹配判断可能会需要该标记。
- images：该格子上要画的图像，一个单元对应一个[image]块。
- images_foreground：基于 images 解析出的、要运行在该格子上的前景动画。
- images_background：基于 images 解析出的、要运行在该格子上的背景动画。
- last_tod：最后时段。
- sotred_images：指示是否已排序过 images。

注

- 为节省内存，在 images 被扩展后（tile::rebuild_cache）可清除 flags、images。
- images、flags 是一次性生成，一次性指的只要一个 build_terrains()函数。images_foreground、images_background 则是按需生成，只有该格子出现在当前窗口内后才会被生成。

被隐藏的行、列

```
[terrain_graphics]
    map="
   *
,
*,    *
,  1
*,    *
,  *"
    ......
[/terrain_graphics]
```

这里或许有人会存在疑问，这样一条规则，它怎么能使得 pos=1 对应上面的(-1, -1)格子处？(-1, -1)的左侧不是已经没有格子，而此规则是要求 pos=1 左侧必须有格子（使用了“*”）。

在程序内部，它的边界其实是两行、两列。地图中画的一行一列边界其实已是第二行、第二列，也就是说以上(-1, -1)左侧还有一列、上面还有一行。虽然在它们外面的格子地形码全是

NONE，但它们确是有效位置。

10.4 随机地图

10.4.1 生成算法主逻辑

随机地图是一个模块，它实现了根据传下的参数自动产生地图，出来的地图类似代码 4-1 的地形标识二维数组。评价随机地图系统优劣，除执行时间和耗费内存，还有产生的地图是否尽可能反映了现实中地貌，以及要产生应用要求的要素。地貌包括要遵循海拔变化，像对于存在山岭、丘陵、平原、浅水和深水地形的，依海拔就要做到山岭下去是丘陵，丘陵下去是平原，平原下去是浅水，浅水再往下是深水。除去海拔，地貌另一要求是气温，当气温是冬天时，大多的浅水就要变成冰面，丘陵要覆盖上积雪。至于应用要求的要素，像地图是要给两人对战游戏用时，就要能决定出两个城镇中心的最佳位置，城镇中心之间要能产生出道路，当这道路是要经过河流时，河流上要自动架桥。

在过程中，生成算法要根据尺寸产生一个更大地图，最后实际产生的地图其实是它的中间部分，为叙述方便让定义两个术语：真实地图、计算地图。

- 真实地图：实际要生成的地图。其尺寸是要求生成的尺寸，它内容来自于计算地图中间部分。
- 计算地图：算法在生成过程中实际操作于的地图。其尺寸是要求尺寸的九倍，垂直、水平各是 3 倍。

产生随机地图可分为两个过程：生成草稿和描绘细节。

步骤一：生成草稿

1. 在 kingdom 工程打开 mapgen.cpp，在 default_generate_map 函数内设断点。
2. 运行时选“Debug”——“Starting Debug”。
3. 进入标题屏幕后“随机地图”，地图列表中选“随机地图”，这时会触发断点。

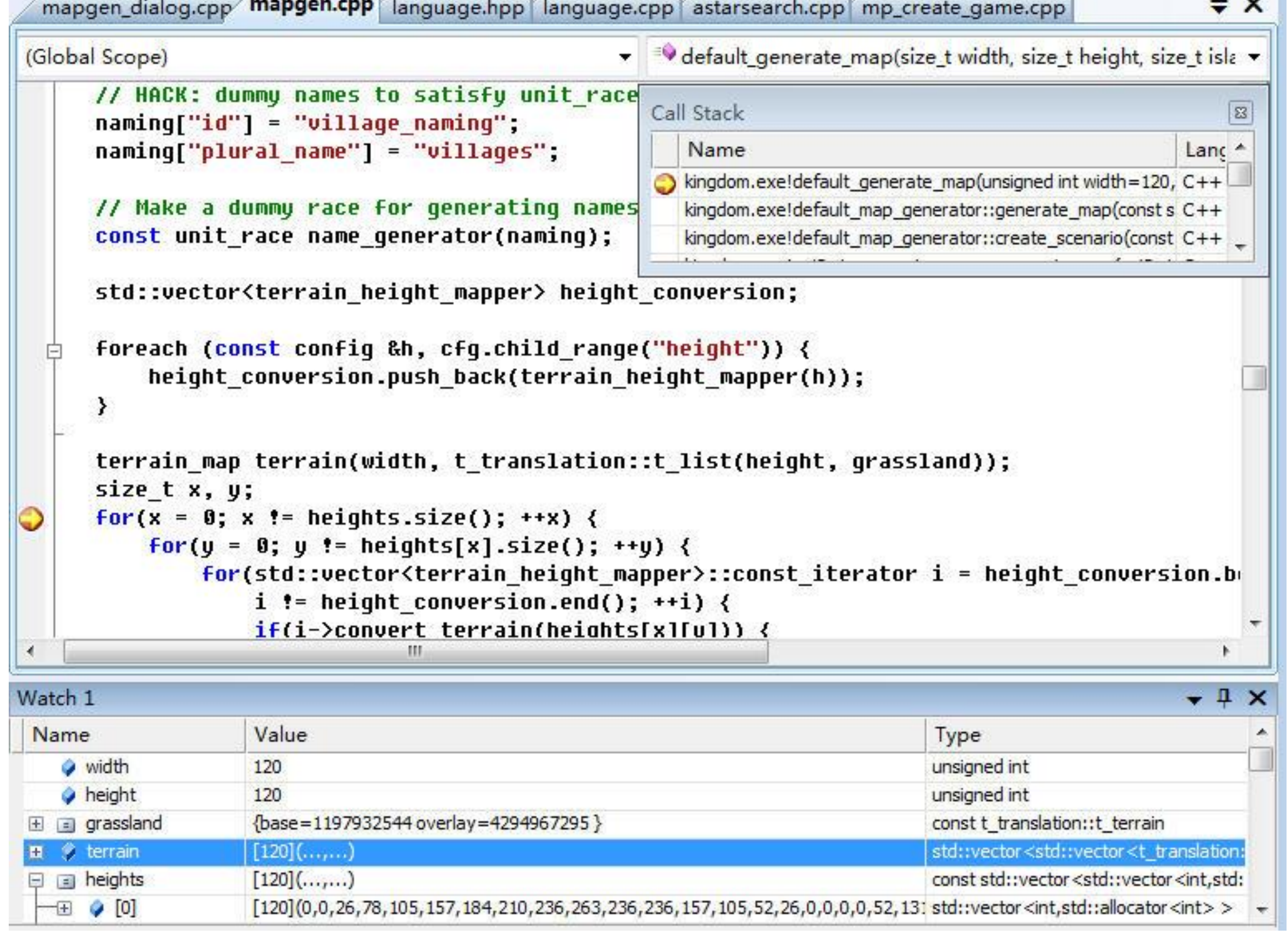


图 4-25 default_generate_map

图 4-25 中的 terrain 就用于存放地图草稿，它是尺寸 120（width）X 120（height）二维数组，数组中元素是用于表示地形的“struct t_terrain”。在图 4-23 断点处，terrain 已被初始化，所有单元被置为“grassland”地形，默认时“grassland”指的是平原（Gg）。

断点处时 terrain 当中地形全是平原（默认时），生成草稿过程是根据海拔分布图修改 terrain 中相应格子的地形。

图 4-25 中的 heights 表示海拔分布图，它也是二维数组，尺寸和 terrain 一样也是 width X height，不过数组中单元不是地形而是海拔值（int）。要理解如何生成 heights 参考“4.4.2 生成海拔/温度分布图”，这里可先不深入如何生成 heights，只要知道 heights 给地图中每个格子都给出了海拔值，这些海拔值在地图上分布是根据要求计算出的，像地图是要山地地图？岛屿地图？

有了海拔分布图 heights，接下是结合以 WML 格式给出的参数修改 terrain，形成草稿。“结合”具体是怎样一个过程？打开“随机地图”对应的 cfg 关卡文件：<data>/multiplayer/scenarios\0p_Random_Scenario.cfg，从中摘取两个[height]块。

```
[height]
    height=600
    terrain=Mm
[/height]
[height]
    height=500
    terrain=Hh
[/height]
```

[height]指示了多少海拔时放置什么地形。以上第一个[height]块表示当该格子海拔大于等于 600 时，放置山岭（Mm）；第二个[height]块表示当该格子海拔小于 600（条件来自上一[height]块）、大于等于 500 时，放置丘陵（Hh）。

图 4-25 中的 height_conversion 存储了 WML 中的[height]块信息。

“i->convert_terrain(heights[x][y])”用于判断坐标是(x,y)的格子、它的海拔是否符合 i 这个[height]块。

为让正确执行“结合”操作，WML 中书写[height]块时必须依着 height 值由大到小。

产生草稿时可同时实现地带。地带指的是具有某种性质一片地方，在这里一片地方具有“某种性质”指的是同一地形，像一片地方主要是平原时，就说该地形是平原地带，是沙漠则是沙漠地带，沼泽则是沼泽地带。如何实现地带，让看下两组[height]块。

```
[height]
   height=500
   terrain=Hh
[/height]
[height]
   height=100
   terrain=Gg
[/height]
第一组[height]块，平原带地。

[height]
   height=150
   terrain=Gs
[/height]
[height]
   height=120
   terrain=Dd
[/height]
第二组[height]块，沙漠带地。
```

对第一组[height]块，海拔大于等于 100、小于 500 时，地形是平原（Gg）；对第二组[height]块，海拔大于等于 120、且小于 150 时，地形是沙漠（Dd）。当生成的海拔分布图中[120, 150)内的海拔值有着较大概率出现时，那通过第一组[height]形成的地貌就是平原地带，第二组就沙漠地带。

小结草稿是怎样一个地图

- 地貌上符合海拔要求。
- 已表示出地带。
- 当格子的海拔不落在任何一个[height]块中时，它使用“grassland”地形（默认平原）。

步骤二：描绘细节

描绘细节指的是基于草稿，根据 WML 参数修改地图数据，即 terrain 中的单元值。

湖泊、河流

“max_lakes”指示地图中最多能存在的湖泊数。湖泊“中心”格子是随机选的，对格子海拔等没要求。“lake_size”和直径相关，但“lake_size”不是就等于直径。地图中各湖泊直径可能是不同的，但最大直径不会大于根号“lake_size”。湖泊涉及到格子会强行置为浅水地形。

河流最终要流入湖泊，所以河流终点肯定是湖泊，但湖泊可能不是河流终点。河流终点要求“中心”格子海拔大于“min_lake_height”，即只有海拔大于“min_lake_height”的格子才能做为河流终点。有了终点后就要计算出这条河流，基于已有的是终点，“计算”等于是向上游溯出河流。

计算河流大致流程。把终点格子作为当前格子，当前格子有 6 个毗邻格子，溯哪个方向是随机的。溯到格子海拔小于等于当前格子时，溯到格子被归入河流；大于时则要判断两格子海拔差，海拔差小于“river_frequency”归入河流，否则不归入（摸拟现实中河流不能出现大落差）。然后以被溯格子为当前格子，依着同样逻辑找到下一个被溯格子，直到某个当前格子的 6 个毗邻（准确说是 5 个）都不能归入河流时，该河流就找到源头了，而串联起过程中各个“当前格子”就是河流。注：过程中没使用 A*算法，对毗邻格子只要求能够满足落差，只要满足了就会

选它，就把它作为当前格子，因而就没必要使用全毗邻估值的 A*。

由于河流的源头一定是湖泊，只有找到源头了才会开始计算河流，所以地图中的河流数肯定小于或等于湖泊数。

气温

气温往往会影响水地形，所以要放在处理完湖泊、河流之后。

对浅水，夏天时显示蓝色，冬天时就要显示为冰面，也就是系统要能根据气温要求生成相应地图。处理气温，首先是要生成气温分布图（temperature_map），temperature_map 和海拔分布图（heights）一样尺寸、一样是 width X height 二维数组，不过数组中单元不是海拔值而是气温值（int）。气温分布图对草稿的描绘类似海拔分布图在“grassland”画出草稿，不同的，海拔是通过[height]块，气温则是[convert]块。

```
[convert]
    min_height=50
    max_temperature=20
    from=Ww, Wo
    to=Ai
[/convert]
摘自<data>/multiplayer/scenarios/Op_Random_Scenario.cfg

[convert]
    min_height=50
    max_temperature=150
    from=Ww, Wo, Ss
    to=Ai
[/convert]
摘自<data>/multiplayer/scenarios/Op_Random_Scenario_Winter.cfg
```

第一个[convert]指示海拔大于等于 50、气温小于等于 20 时，浅水、深水地形变成冰面。第二个[convert]指示海拔大于等于 50、气温小于等于 150 时，浅水、深水、沼泽地形变成冰面。由于第二个提高了要变成冰面的气温范围、而且把沼泽也归入要变成冰面的地形，这就会造成地图上更大面积冰面，也就是实现了低气温（冬天）地形。

城镇中心

城镇中心必须是在真实地图内。要放置在哪个坐标，采用的是对真实地图内所有格子计算分数，找出分数最好的格子，分数计算依据主要来自它、以及它周围格子对 “valid_terrian”中地形的匹配程度。WML 中的[castle]块用于指示如何设置城镇中心，像“valid_terrain”指示城镇中心必须放置在何样地形之上；“min_distance”指示两城镇中心间至少间隔多少距离；“castle_size”则是城镇范围，它不影响地形生成过程，但它告诉地图生成器删除以中心为圆心、邻近它的“castle_size”个格子上的所有标签。

道路

“roads”加上城镇中心数平方等于地图中道路数。城镇中心之间会有道路把它们连接起来。

要计算出一条道路，首先要计算出起点、终点。对于是连接城镇中心道路，它的起、终点就是两城镇中心，但不是这类道路时就要使用相对来说较为复杂的规则。

- 起点、终点必须是路径能通过的地形。[road_cost]中的“terrain”指示了道路能通过的地形。
- 在坐标选择上，起点、终点是随机的，但会选在边界。25%机率在计算地图的上边界，25%在真实地图的上边界，25%在计算地图的左边界，25%在真实地图的左边界。
- 起点、终点不能在同一边界。像不能都在计算地图的上边界。

找到起点和终点后，就要决定怎么修出道路，即确定中间是哪些格子。从起点开始，比较相邻 6 个格子，选择成本（“cost”）最小的，然后把此个成本最小格子做为当前格子，比较它的相邻 6 个格子（准确说是 5 个），选择成本最小的……这样不断下去，直到找到地图边界或是相邻

格子没了符合可以修道路的格子，而过程中踩过的格子就形成了路径。寻路过程采用 A*算法（深入 A*参考“**10.1 A*算法**”），既然是 A*肯定需要估值函数，具体到此处是不同格子地形可能不同，估值就按该地形来，也就是在该地形上需要的修路成本。WML 中的[road_cost]指示了道路通过各种地形时需要的成本。

```
[road_cost]
   terrain=Gg
   cost=10
   convert_to=Rr
[/road_cost]
[road_cost]
   terrain=Ww
   cost=50
   convert_to_bridge=Ww^Bw|, Ww^Bw/, Ww^Bw\
   convert_to=Ch
[/road_cost]
```

在草原地形（Gg）成本是 10，在浅水地形（Ww）成本是 50，也就是当一个格子周围既有草原又有浅水时，它会优先选草原。

如果一种地形在 WML 没有相应[road_cost]块，那该地形上不能修路。

当一个格子确定要通路径时，该格子需改为“道路”地形，“道路”加引号是因为改到的地形没必要一定是道路，是想说改变地形的原因是该格子上要修路。[road_const]中的 convert_to 就指示了即将是路径的格子是“terrain”指定的地形时它要转换到的“道路”地形。第一个[road_cost]表示要修路的格子是草原地形时，该格子就变成道路（Rr）地形；第一个[road_cost]表示要修路的格子是浅水（Ww）地形时，该格子就变成城堡（Ch）地形。

除了需要转换为道路地形，路径中格子可能还要转换为桥梁地形。要通过浅水、深水、沼泽时要把地形转为桥梁。[road_const]中的 convert_to_bridge 指示了当是“terrain”指定的地形时要转换到的桥梁地形。桥梁有三个方向，南北向：Ww^Bw|；东北、西南向：Ww^Bw/；东南、西北向:Ww^Bw\。书写 convert_to_brighe 值必须按这三次序。很显然，三个格子组成的方向不可能只有三种，当出现以上三种之外方向时，像出现拐弯，就使用“convert_to”指定的地形。

WML 参数小结

- map_width、map_height：要生成的地图尺寸。
- iterations、hill_size：设置海拔分布图。
- temperature_iterations、temperature_size：设置气温分布图。
- max_lakes：地图中最多能存在的湖泊数。
- lake_size：湖泊直径。
- min_lake_height：河流终点至少要达到的海拔。
- river_frequency：河流内的相邻两格子间最大允许的海拔差。
- roads：除去连着城中心道路外，地图中最多能存在的道路数。
- [height]：由它和海拔分布图生成草稿。
- [convert]：由它和气温分布图在地图上表现出气温。
- [road_cost]：在该地形上的修路成本，以及成为道路后要转化到的地形、桥梁。
- [castle]：放置城镇中心条件，像可放置在何地形，两中心间至少多少距离。

10.4.2 生成海拔/气温分布图（generate_height_map）

生成的地图要遵循海拔变化。举个例子，对于存在山岭、丘陵、平原、浅水和深水地形的地图，依海拔就要做到，在山岭下去是丘陵，丘陵下去是平原，平原下去是浅水，浅水再往下是深水。

对生成的地图来说，上层只是指示说这地图大概面貌，像山地大概占多少，但没指定哪坐标上出现山地。

generate_height_map 就用于解决以上两个问题，1）让生成的地形依从海拔变化；2）决定出

哪坐标上出现地形、深水。

generate_height_map 生成海拔分布图。之后程序就可根据分布图中各个坐标点上生成的海拔数据去放置地形，像 600 以上放山岭，400 以上放丘陵。

算法逻辑

generate_height_map 采用迭代画刷法。

1. 在区域内随机选择一点 p1，在这点上涂出一个画刷。画刷是个圆，p1 是圆心，半径 r 就是 hill_size，因而 hill_size 叫山地半径或山地面积。
2. 画刷中涉及到的各坐标点权值是不一样的。圆心 p1 取最大值，值是 hill_size，余下按扩散距离减少，针对图 4-26 中 p2 的权值是“r-R1”，把这个权值累加到相应点上的海拔值。（R1 是“p1”、“p2”组成的直角三角形斜边，知道“p1”、“p2”坐标就可用勾股定理计算出 R1。）

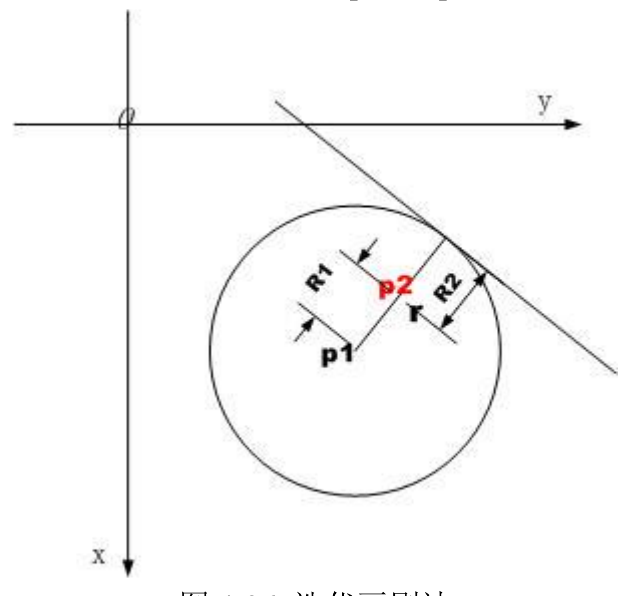


图 4-26 迭代画刷法

3. 不断重复 1、2 步骤，直到 iterators 次。
4. 规范化分布图中数据，让各点的海拔值落在[0, 1000]。规范化方法是计算出分布图中的最小海拔和最大海拔，让最小海拔对应规范化后 0 值，最大海拔对应规范化后 1000 值。

算法如何达到两个目的

1. 让生成的地形依从海拔变化。画刷中权值依距离在变化，其累加出的海拔也就大致依从了这个变化。
2. 决定出哪坐标上出现地形、深水等。p1 随择点是随机的，那些多次被选中或附近多次被选中就形成海拔高的山岭，很少选中的就形成了海拔低的深水。

选择“合适”p1 让产生岛屿地图

依以上算法，如果选择的 p1 都落在以地图中心为圆点、island_size 为半径圆内，那地图中以地图中心为圆点，以 island_size+hill_size 为半径圆内就出现海拔高的地形，而之外的就形成拔低的深水，从而产生岛屿地图。

第十一章 其它话题

本章目标

- 掌握 A*算法
- 音频重采样
- 图像重采样
- 处理 iOS 事件
- 嵌入自己的工程编译 Boost 库

11.1 A*算法

11.1.1 理论

以下内容摘自“算法驿站”（http://blog.pfan.cn/rickone）

不管哪种搜索，都统一用这样的形式表示：搜索的对象是一个图，它面向一个问题，不一定有明确的存储形式，但它里面的一个结点都有可能是一个解（可行解），搜索的目的有两个方面，或者求可行解，或者从可行解集中求最优解，我们用两张表来进行搜索，**一个叫 OPEN 表，表示那些已经展开但还没有访问的结点集，另一个叫 CLOSE 表，表示那些已经访问的结点集。**

一、蛮力搜索（DFS，BFS）

DFS 和 BFS 是最基本的搜索算法，用上面的形式表示它们是非常相似的。

BFS（Breadth-First-Search 宽度优先搜索）

首先将起始结点放入 OPEN 表，CLOSE 表置空，算法开始时：

1. 如果 OPEN 表不为空，从表中开始取一个结点 S，如果为空算法失败
2. S 是目标解吗？是，找到一个解（继续寻找，或终止算法）；不是到 3
3. 将 S 的所有后继结点展开，就是从 S 可以直接关联的结点（子结点），如果不在 CLOSE 表中，就将它们放入 OPEN 表**末尾**，并把 S 放入 CLOSE 表，重复算法到 1

DFS（Depth-First-Search 深度优先搜索）

首先将起始结点放入 OPEN 表，CLOSE 表置空，算法开始时：

1. 如果 OPEN 表不为空，从表中开始取一个结点 S，如果为空算法失败
2. S 是目标解吗？是，找到一个解（继续寻找，或终止算法）；不是到 3
3. 将 S 的所有后继结点展开，就是从 S 可以直接关联的结点（子结点），如果不在 CLOSE 表中，就将它们放入 OPEN 表**开始**，并把 S 放入 CLOSE 表，重复算法到 1

没看出有什么不同？擦干净了眼镜再看一遍吧。

知道 BFS 和 DFS 有什么不同吗？B 和 D 嘛。

仔细观察 OPEN 表中待访问的结点的组织形式，BFS 是从表头取结点，向表尾添加结点，也就是说 OPEN 表是一个队列，没错，BFS 首先就要让你想到‘队列’；而 DFS，它是从 OPEN 表头取结点，也向表头添加结点，也就是说 OPEN 表是一个栈！

DFS 用到了栈，所以有一个很好的实现方法，那就是递归，系统栈是计算机程序中极重要的部分之一，你不想递归？怕它用完了？放那儿也是浪费。而且用递归也有个好处就是，在系统栈中只需要存结点最大深度那么大的空间，也就是在展开一个结点的后续结点时可以不用一次全部展开，用一些环境变量记录当前的状态，在递归调用结束后继续展开。

利用系统栈实现的 DFS

```
函数 dfs(结点 s)
{
```

```
        s超过最大深度了吗？是：相应处理，返回；
        s是目标结点吗？是：相应处理；否则：
        {
              s放入CLOSE表；
              for(c=s.第一个子结点 ；c不为空 ；c=c.下一个子结点() )
                    if(c不在CLOSE表中)
                          dfs(c)；递归
        }
}
```

如果指定最大搜索深度为 n，那系统栈最多使用 n 个单位，它相当于有状态指示的 OPEN 表，状态就是 c，在栈里存了前面搜索时的中间变量 c，在后面的递归结束后，c 继续后移。在象棋等棋类程序中，就是用这样的 DFS 的基本模式搜索棋局局面树的，因为如果用 OPEN 表，有可能还没完成搜索 OPEN 表就暴满了，这是难于控制的情况。

们说 DFS 和 BFS 都是蛮力搜索，因为它们在搜索到一个结点时，在展开它的后续结点时，是对它们没有任何‘认识’的，它认为它的孩子们都是一样的‘优秀’，但事实并非如此，后续结点是有好有坏的。好，就是说它离目标结点‘近’，如果优先处理它，就会更快的找到目标结点，从而整体上提高搜索性能。

二、启发式搜索

为了改善上面的算法，我们需要对展开后续结点时对子结点有所了解，这里需要一个估值函数，估值函数就是评价函数，它用来评价子结点的好坏，因为准确评价是不可能的，所以称为估值。打个比方，估值函数就像一台显微镜，一双‘慧眼’，它能分辨出看上去一样的孩子们的手，哪个很脏，有细菌，哪个没有，很干净，然后对那些干净的孩子进行奖励。(我不是老师，不会打比方，呵）对，这里似乎需要‘排序’，排序是要有代价的，而花时间做这样的工作会不会对整体搜索效率有所帮助呢，这完全取决于估值函数。

排序，怎么排？翻翻《数据结构》排序算法一大堆，用哪一个？你会很喜欢高级的排序算法，qsort！不一定，要看要排多少结点，如果很少，简单排序法就很 OK 了。看具体情况了。

排序可能是对 OPEN 表整体进行排序，也可以是对后续展开的子结点排序，排序的目的就是要使程序有启发性，更快的搜出目标解。

如果估值函数只考虑结点的某种性能上的价值，而不考虑深度，比较有名的就是有序搜索（Ordered-Search)，它着重看好能否找出解，而不看解离起始结点的距离（深度）。如果估值函数考虑了深度，或者是带权距离（从起始结点到目标结点的距离加权和），那就是 A*，举个问题例子，八数码问题，如果不考虑深度，就是说不要求最少步数，移动一步就相当于向后多展开一层结点，深度多算一层，如果要求最少步数，那就需要用 A*。简单的来说 A*就是将估值函数分成两个部分，一个部分是路径价值，另一个部分是一般性启发价值，合在一起算估整个结点的价值，具体 A*以后再谈。

从 A*的角度看前面的搜索方法，如果路径价值为 0 就是有序搜索，如果路径价值就用所在结点到起始结点的距离（深度）表示，而启发值为 0，那就是 BFS 或者 DFS，它们两刚好是个反的，BFS 是从 OPEN 表中选一个深度最小的进行展开，而 DFS 是从 OPEN 表中选一个深度最大的进行展开，它们俩还真是亲兄弟啊！当然只有 BFS 算是特殊的 A*，所以 BFS 可以求要求路径最短的问题，只是没有任何启发性。

三、A*搜索

了解了基本搜索算法，下面就来看 A*。

A*是一种启发式搜索，一种有序搜索，它之所以特殊完全是在它的估值函数上，如果我要求的是从初始结点到目的结点的一个最短路径（或加权代价）的可行解，那对于一个还不是目标结点的结点，我对它的评价就要从两个方面评价：第一，离目标结点有多近，越近越好；第二，离起始结点有多远，越近越好。记号[a,b]是表示结点 a 到结点 b 的实际最短路径代价。设

起始结点为 S，当前结点为 n，目标结点为 G，于是 n 的实际代价应该是 $f^*(n)=g^*(n)+h^*(n)$，其中 $g^*(n)=[S,n]$,$h^*(n)=[n,G]$，对于是 $g^*(n)$是比较容易得到的，在搜索的过程中我们可以按搜索的顺序对它进行累积计算，当然按 BFS 和 DFS 的不同，我们对它的估价 $g(n)$可以满足 $g(n)>=g^*(n)$，大多可以是相等的。但是对于 $h^*(n)$我们却了解得非常少，目标结点正是要搜索的目的，我们是不知道在哪，就更不知道从 n 到目标结点的路径代价，但是或多或少我们还是可以估计的，记估价函数 $f(n)=g(n)+h(n)$。

我们说如果在一般的图搜索算法中应用了上面的估价函数对 OPEN 表进行排序的，就称 A 算法。在 A 算法之上，如果加上一个条件，对于所有的结点 x，都有 $h(x)<=h^*(x)$，那就称为 A*算法。如果取 $h(n)=0$ 同样是 A*算法，这样它就退化成了有序算法。

A*算法是否成功，也就是说是否在效率上胜过蛮力搜索算法，就在于 $h(n)$的选取，它不能大于实际的 $h^*(n)$，要保守一点，但越接近 $h^*(n)$给我们的启发性就越大，是一个难把握的东西。

A*算法流程

首先将起始结点 S 放入 OPEN 表，CLOSE 表置空，算法开始时：

1. 如果 OPEN 表不为空，从表头取一个结点 n，如果为空算法失败
2. n 是目标解吗？是，找到一个解（继续寻找，或终止算法）；不是到 3
3. 将 n 的所有后继结点展开，就是从 n 可以直接关联的结点（子结点），如果不在 CLOSE 表中，就将它们放入 OPEN 表，并把 S 放入 CLOSE 表，同时计算每一个后继结点的估价值 $f(n)$，将 OPEN 表按 $f(x)$排序，最小的放在表头，重复算法到 1

最短路径问题，Dijkstra 算法与 A*

A*是求这样一个和最短路径有关的问题，那单纯的最短路径问题当然可以用 A*来算，对于 $g(n)$就是[S,n]，在搜索过程中计算，而 $h(n)$我想不出很好的办法，对于一个抽象的图搜索，很难找到很好的 $h(n)$，因为 $h(n)$和具体的问题有关。只好是 $h(n)=0$，退为有序搜索，举一个例子：

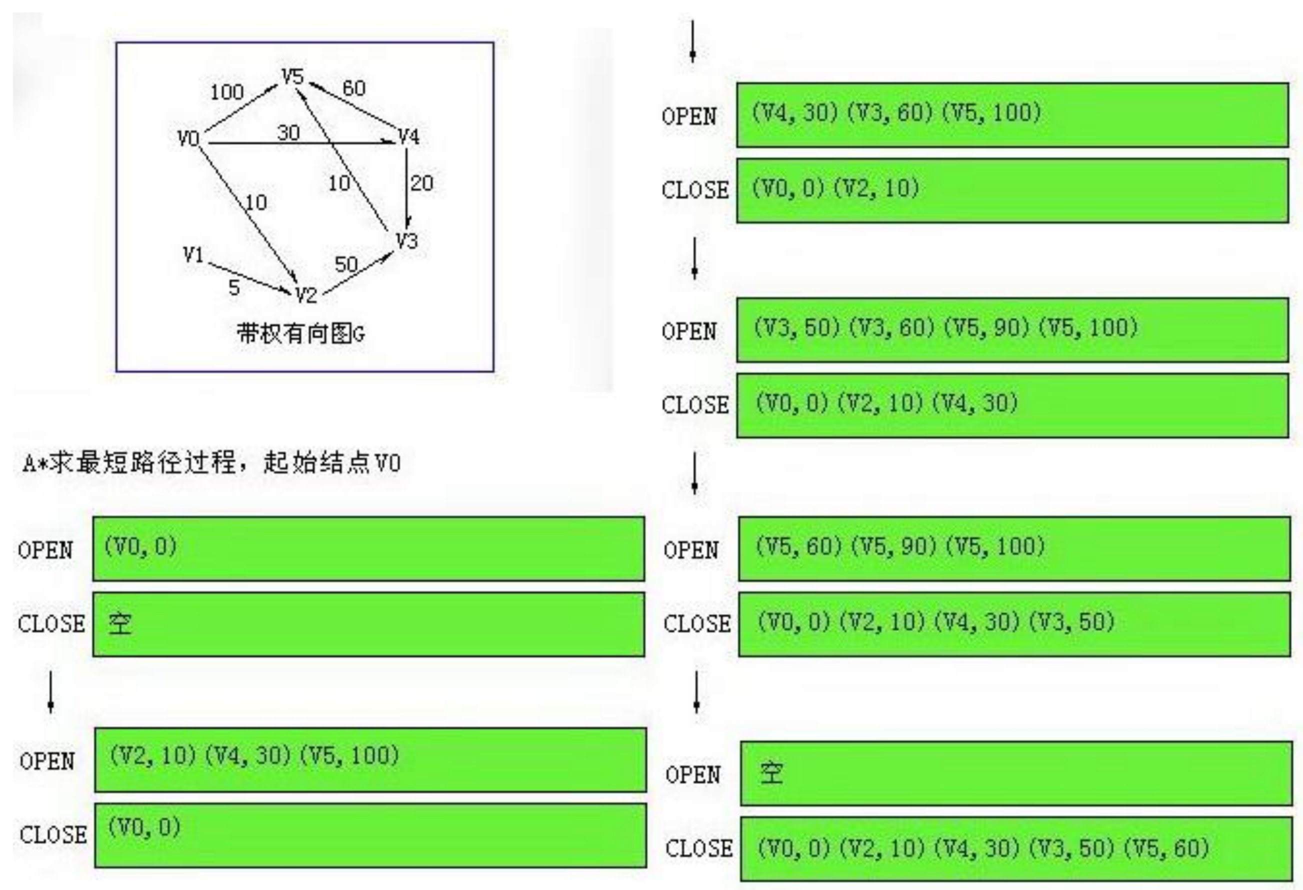


图 10-1 a-start

与结点写在一起的数值表示那个结点的价值 $f(n)$，当 OPEN 表为空时 CLOSE 表中将求得从

V0 到其它所有结点的最短路径。考虑到算法性能，外循环中每次从 OPEN 表取一个元素，共取了 n 次（共 n 个结点），每次展开一个结点的后续结点时，需 O(n)次，同时再对 OPEN 表做一次排序，OPEN 表大小是 O(n)量级的，若用快排就是 O(nlogn)，乘以外循环总的复杂度是 O(n^2logn)，如果每次不是对 OPEN 表进行排序，因为总是不断地有新的结点添加进来，所以不用进行排序，而是每次从 OPEN 表中求一个最小的，那只需要 O(n)的复杂度，所以总的复杂度为 O(n*n)，这相当于 Dijkstra 算法。在这个算法基础之上稍加改进就是 Dijkstra 算法。OPEN 表中常出现这样的表项：(Vk,fk1)(Vk,fk2)(Vk,fk3)，而从算法上看，只有 fk 最小的一个才有用，于是可以将它们合并，整个 OPEN 表表示当前的从 V0 到其它各点的最短路径，定长为 n，且初始时为 V0 可直接到达的权值（不能到达为 INFINITY），于是就成了 Dijkstra 算法。

11.1.2 算法框架

如何用 C/C++实现 A*，从 A*算法流程大概可把代码归为两类：算法框架、估值函数。

算法框架实现管理 open 表、close 表，排序 open 表中结点等算法的框架性工作。让以搜索最“佳”路径为目的，看如何编写 A*框架代码。

1、打开<src>/pathfind/astarsearch.cpp，在 a_start_search(...)内加入代码，并在 int ii = 0 处设断点。

```
if (dst == map_location(65, 48)) {
    map_location& loc = locs[i_];
    int ii = 0;
}
```

2、运行时选“Debug”——“Starting Without Debuging”。
3、标题屏幕选“战役”，一路“确认”，直到进入“群雄争霸”大地图。
4、“Debug”——“Attach to Process...”挂接程序。
5、鼠标击中“马超”部队，再把鼠标移动到右下角坐标（65, 48），界面上显示的是（66, 49），这时会触发断点。

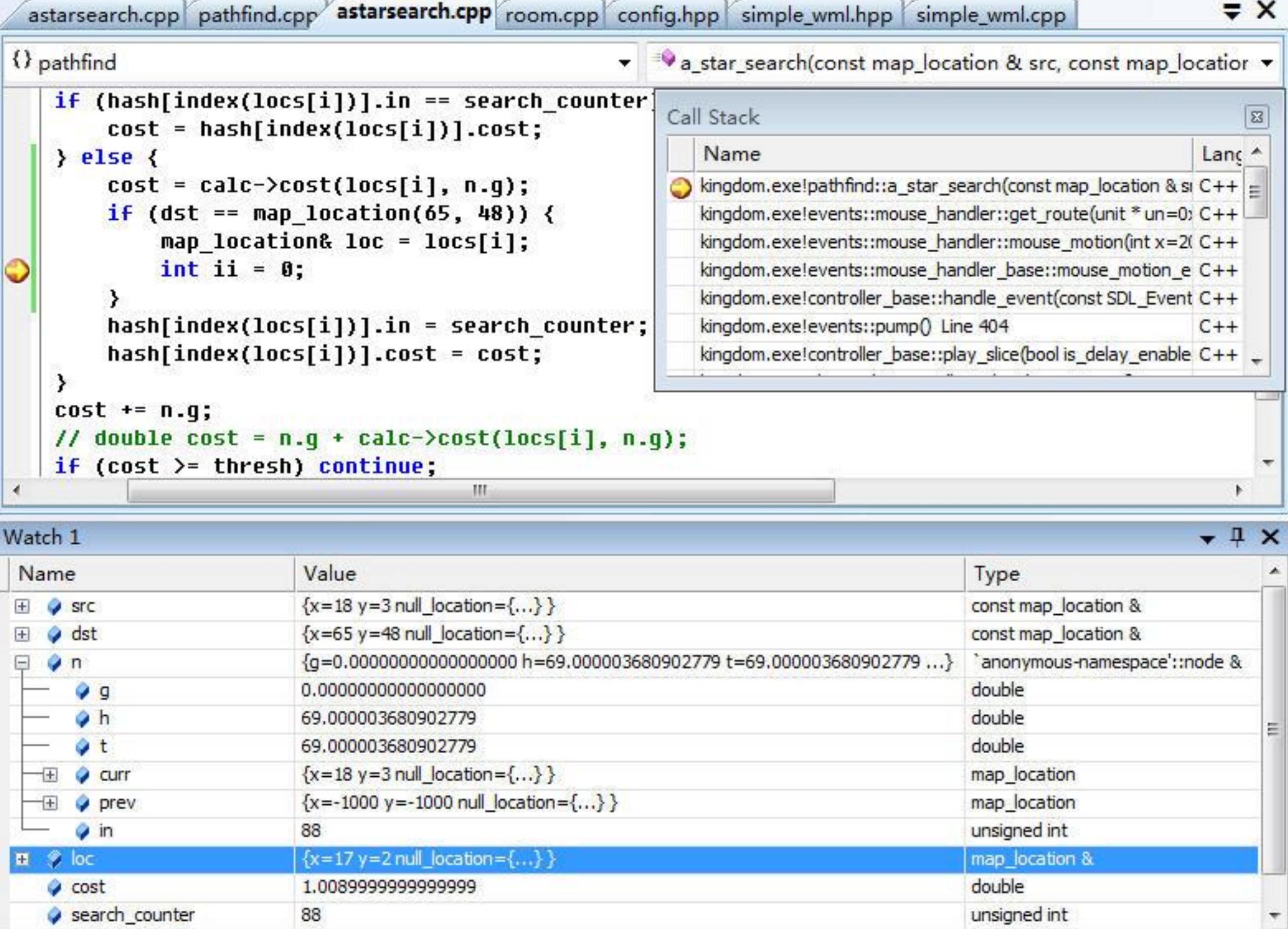


图 10-2 a_star_search

a_start_search 用于搜出一条从 src 到 dst 的最佳路径，图 10-2 显示展开当前节点 n 而执行的部分代码。展开当前节点 n 需作两个任务，1）判断当前节点 n 是否是目标解，如果是则退出搜索；2）不是目标解时，基于当前节点 n 根据规则计算出后继结点，并估算每个后继结点代价，筛选后放入 OPEN 表，然后重排 OPEN 表，重排时把最“佳”节点放在表头，它将作为下一次判断的当前结点。

Watch 中有个 n 变量，它的类型是 node，其中的 t、g、h 分别表示 f(n)=g(n)+h(n)中的 f、g、h 计算出的 double 值。注：t、g、h 默认值是 1e25，即 10 的 25 次方。

在此个实例中，源格子 src 是（18, 3），目标格子 dst 是（65, 48），a_start_search 要通过 A*算法搜出一条从（18, 3）到（65, 48）的最“佳”路径。让以这个实例“代入”A*算法“通用”公式。

- **OPEN 表：pq**
  pq 类型是 std::vector<int>，该 std::vector 中一个元素表示一个结点，元素具体值是（loc.y * h_ + loc.x），h_是地图高度，（loc.x, loc.y）则是某个 loc 坐标，因而 OPEN 表中结点是某一坐标。因为要排序，open 表不是一个简单 std::vector，它对内中元素时刻符合“堆排序”标准。
- **当前节点 n 是否是目标解**
  符合目标解可能是两种情况，1）n 后继结点中一个正好就是 dst；2）n 后继结点都不能放入 open 表，并且 n 是 open 表中唯一结点。对于第一种情况下目标解是成功找到了一条路径，对于第二种则是虽然符合了目标解，但出现的是此次无法找到路径。
- **基于当前节点 n 计算后继结点的规则**
  n 周围的 6 个格子是 n 的后继结点。
- **估算每个后继结点代价**
  结点代价 f(next)=g(next)+h(next)，估算就要分别计算出 g(next)、h(next)。对 g(next)，使用 cost=calc->cost(locs[i_], n.g)。locs[i_]指示某个要估值的后继结点，n.g 是 f(n)=g(n)+h(n)中的 g(n)，即从起始结点到当前结点的最短路径代价。cost(...)计算出的不是 h(n)，而是在 locs[i_]这格子上的代价！所以 g(next)=g(n)+cost，即从起始结点到此个后继结点最矩路径代价等于从起始结点到当前结点代价加上当前结点到此个后继结点代价。对 h(next)，使用 h(next)=heuristic(next, dst)，计算出从此个后继结点到终点路径代价，heuristic 计算时没考虑地图上部队分布、地形等情况，它就近似计算出个平面直角坐标系坐标 next 到坐标 dst 之间距离。
- **筛选后继结点并放入 OPEN 表**
  分析产生后继结点规则可看出，后继结点最多有 6 个，但 6 个有时不能全放入 OPEN 表，哪些是不能入的呢？——1）该后继结点已被访问过，并且它那时的 g(next)好于此刻 g(next)；2）该后继结点没被访问过，但计算出的 g(next)超过门限，像骑兵部队不能移动到山岭，在所有山岭格子就不能放入 OPEN 表。
- **重排 OPEN 表**
  此时 OPEN 表存的结点都已计算出代价 n.t（n.t=n.g+n.h），重排就是根据 n.t 值由优到劣排序（由小到大）。最优放在前头，它将作为下一次判断的当前结点。由于程序对 OPEN 表 qp 内中元素时刻符合“堆排序”标准，但加入/修改一个元素时只要调用 std::push_heap 对刚插入的（尾部）元素做堆排序，而不必调用 std::sort_heap 重排整个 OPEN 表。

根据以上分析，会发现没有提到 CLOSE 表。在描述 A*算法时，CLOSE 表功能是用于核对后继结点，看后继结点是否能放入 OPEN 表，但在此个例子中，能不能放入 OPEN 表判断条件是此次代价是否要比“上次”好，只要是好，即使是已经访问过的依旧可以继续存在 OPEN 表。对“上次”为什么加引号，是因为没访问过结点也归入“上次”结点，当没访问过时此次是一

定比“上次”好。

node 中 in 字段

说到 in 字段则必须说 search_counter 变量。search_counter 是个全局变量，但在一次 a_star_search 过程中它的值保持不变。图 10-2 显示此次 a_star_search 时 search_counter 值：88（如果没有回转，它表示这是第 44 次调用 a_star_search）。in 和 search_counter 关系是：在一次 a_star_search 过程中，访问过结点会被置为 in=search_counter+1。注：search_counter 最小值是 2（不可能是 0 和 1）。

in 变量用于判断某一格子在此次 a_star_search 时是否已被访问过。此次 a_star_search 没被访问过时，值是不确定的，但满足“(in - search_counter <= 1u)”不等式成立，这个式子中 in 可能比 search_counter 小，但因为是无符号数，<还是成立。

对 in 变量，要标志 node 是否访问过，用个 bool 变量不是更简单，干吗要 in 又要 search_counter，这不是增加复杂度吗？——这么做原因是 nodes 是个静态变量，nodes 中的元素数是地图格子数，用 in、search_counter 是为省去第“二”次及以后的初始化 nodes 时间。因为不初始化 nodes，第 N+1 次 a_star_search 用的是第 N 次操作后形成的 nodes，N+1 次的 in 值自然也要基于 N 次 in 值，而那个 in 值将“小于”此次 search_couter。

再者，要用 bool 变量标识的话，逃不掉各个 node 中开辟个状态变量方法，这个变量在搜索前全部置为 false，访问过后置为 true。用了以上 in+serach_counter 方法后，省略了这个全置 false 过程。

和 in 状态相关的另一个可能疑问，函数每次搜索前会调用 nodes.resize，nodes.resize 会不会把所有 in 复位为 0？——不会。以下是 resize 的函数实现。

```
void resize(size_type new_size) { resize(new_size, T()); }
void resize(size_type new_size, const T& x)
{
    if (new_size < size()) {
        erase(begin() + new_size, end()); // erase 区间范围以外的数据，确保区间以外
的数据无效
    } else {
        insert(end(), new_size - size(), x); // 填补区间范围内空缺的数据，确保区间内
的数据有效
    }
}
```

	in-search_counter<=1u(*1)	in=search_counter	in=search_counter+1
是否在 OPEN 表	不在/在	不在	在
in 对应的 g 是否有效	有效	有效	有效
是否分析过(*2)	已分析/未分析	已分析	可能未分析、可能已分析

注

1：此次未展开过该结点时，结点中的 in 比 search_counter 小，而且超过 1，但因为是无符号数减法，“<=1”不成立。

2：分析指的是否基于它展开过结点。

框架代码小结

首先将起始结点 src 放入 OPEN 表 pq，算法开始时：

1. 如果 pq 表不为空，从表头取一个结点 n，如果为空算法失败
2. n 是目标解吗？是，找到一个解，终止算法
3. 将 n 的所有后继结点展开，根据代价判断出比“上次”更好，就将它们放入 pq，并把 pq 按

代价排序，最小的放在表头，重复算法到 1

10.1.3 估值函数

估值函数用于计算代价。A*算法中代价会表现出多种，像 f(n)=g(n)+h(n)中就有指示从起始结点到目标结点代价的 f，从起始结点到当前结点代价的 g，以及从当前结点到目标结点代价的 h。上小节分析可看出，计算 g 用的估值函数是 g(next)=calc->cost(locs[i_], n.g)（cost 计算的虽然是在 locs[i_]上代价，但 g(n)是由当中一个个路过结点消耗的 cost(...)累加而成），计算 h 用的是 h(next)=heuristic(next, dst)，也就说同次算法中也可存在多种估值函数。对 cost(...)，heuristic(...)这两个估值函数，前者较为复杂也更为主要，让着重分析它。

1、打开<src>/pathfind/pattfind.cpp，在 shortest_path_calculator::cost 函数设断点。
2、运行时选“Debug”——“Starting Without Debuging”。
3、标题屏幕选“战役”，一路“确认”，直到进入“群雄争霸”大地图。
4、“Debug”——“Attach to Process...”挂接程序。
5、鼠标击中“马超”部队，再把鼠标移动到地图上其它部队附近，会触发断点。

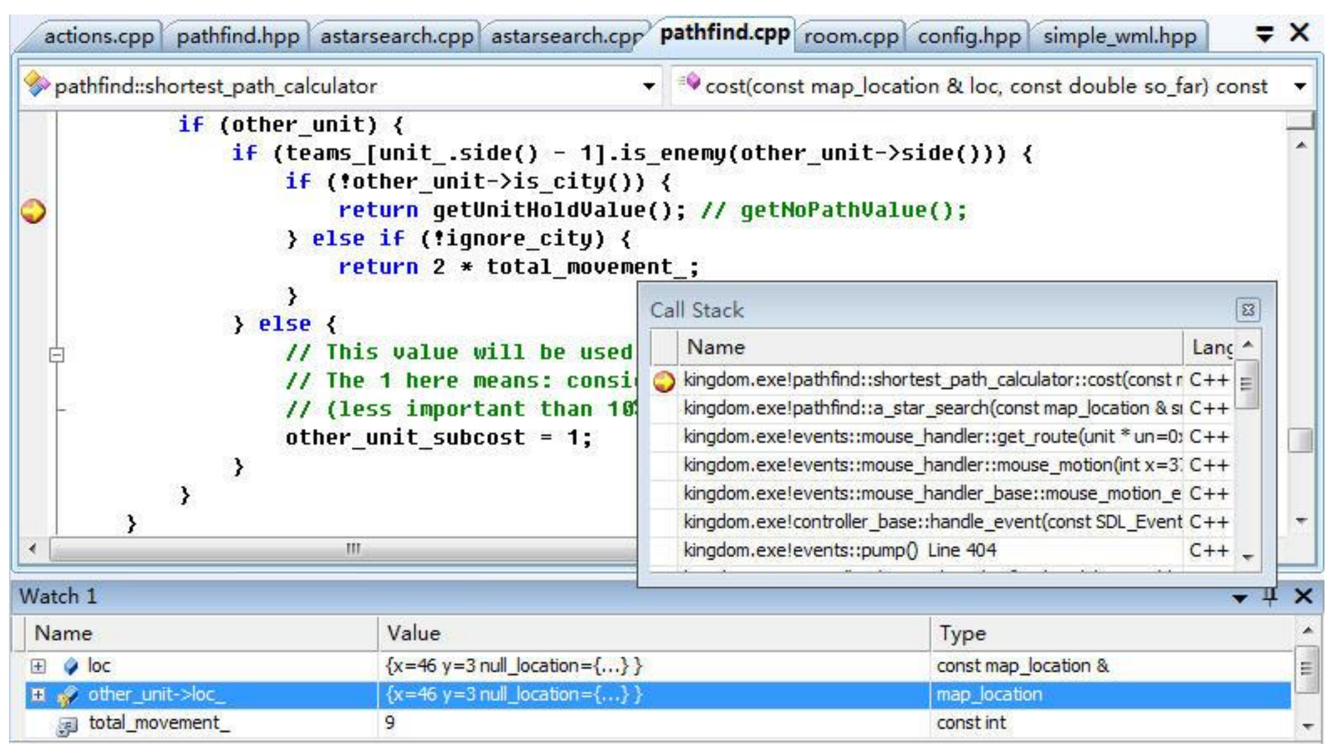


图 10-3 shortest_path_calculator::cost

图 10-3 显示要在一个站了敌方部队的格子 loc=(46, 3)上计算代价时代码。cost 一旦发现该格子上有敌方部队，算出的代价是 getUnitHoldValue()，即 424242.0。

自变量

估值函数既然是函数，就存在自变量，cost 要计算出部队在 loc 格子上付出的代价，以下是它的自变量。

1、loc 格子地形。像骑兵部队，它在平原消耗 1，在森林是 2，在山岭则不能通过。
2、loc 格子上是否有部队。如果有敌方部队那格子就不能通过。
3、loc 格子周围是否有部队。如果周围有敌方部队，loc 格子会被 zoc，消耗移动力将不是地形消耗值，是剩余移动力归零。
4、部队在 loc 格子上防御力。不同部队在不同地形有不同防御力，防御力高的代价要低于防御力高低的。

代价值

估值函数算出的代价是多少，把这问题称为代价值问题。要讨论代价值，则需要知道代价值有什么用。这个作用可分为 A*算法内在作用、非 A*的应用上作用。

- **A*算法内在作用**
  代价值用于排序 OPEN 表。排序 OPEN 表时用的代价值操作是比较，更直白地说是小于、等于、大于，这是种基于同一基数上的比较，所以只要符合同一基数，代价值具体多少是无所谓的。举个例子，1，2，3 和 5，10，15，前个代价值让假设基数是 1，后个基数是 5，虽然它们是不同值但排序产生的结果是一样的。
- **非 A*的应用上作用**
  以上说的 cost(...)计算出代价，它另一作用是设置路径门限。为什么要设置门限？起始、目标两个格子可能是随意指定的，有可能就找不到这两个格子之间路径，像中间隔着条骑兵无法通过的山岭，这时骑兵部队只能是是找不到路径，此次搜索失败。程序为让尽快返回失败，它需要设置门限，一旦累计确定的需要代价已超过这个门限，就不必再搜了，直接失败好了。设置门限时就需要知道 cost(...)具体是如何对代价计算值，像无法通过地形它的代价值是多少（程序不可能用个无限值，用的是 getNoPathValue，即 42424242.0）。

优劣评价标准

评价估值函数优劣，类似代价值也可从 A*和非 A*来衡量。

- **A*算法内标准**
  代价函数要尽能快地让算法找到目标解。以搜索路径来说，经过格子越少往往会做到尽可能快。
- **非 A*的的应用上标准**
  应用上评价就要结合程序要求搜索时达到的应用目标。以搜索路径来说，像搜索时要尽量避开站着敌方部队格子，尽可能路过防御力高的格子。

要满足以上两个“优”要求往往需要权衡，像由 A 到 B 选一条单格子小路是最快的，可要是那里站只敌方部队，以应用来说须要绕道，这叫以味着选它就符合前者，不选它就符合后者。实现估值函数时考虑到这些权衡，除去估值函数本身，这个权衡往往需要外界协助，像设置搜索门限。

```
double getUnitHoldValue() const { return (424242.0); }
......
stop_at = calc.getUnitHoldValue() + 10000.0;
```

以上设置一个门限，目标上解释是它要求选道时可以经过一个敌方部队站着的格子，但只允许一个，10000 是用于一路上非站有敌方部队的代价。

A*算法小结

- A*算法用途很多，《王国战争》中 A*算法不仅用于搜索最佳路径，也用于实现根据部队所在格子和可用移动力计算出它的可到达位置（见 find_routes）。
- 根据实际应用设计“好”的估值函数。
- 基于当前结点得到后继结点、计算后继结点价值、后继结点能否放入 OPEN 表，它们在各个应用下往往是不同的。

11.2 重采样算法

- 把一段采样率是 8K 的音频数据转变成 44.1K，这种转换称为插值（Interpolation）。
- 把一段采样率是 48K 的音频数据转变成 44.1K，这种转换称为抽样（Sampling）。
- 把一个分辨率是 48x48 的图像缩放成 72x72，这种转换称为拉伸（Upscaling）。

- 把一个分辨率是 72x72 的图像缩放成 48x48，这种转换称为缩小（Downscaling）。

以上的插值、抽样、拉伸、缩小可归纳为同一个问题：由 N 样本生成 M 样本。术语重采样（Resampling）来描述这个过程。根据重采样前后样本不同，重采样分为两类，当 N>M，即样本减少的重采样称为下采样（Downsampling），包括音频抽样、图像缩小；反之当 N<M，即样本增多的重采样称为上采样（Upsampling），包括音频插值、图像拉伸。

如何实现重采样？一般讲叙数字多媒体技术的书都会说到这问题，采样方法有简单、有复杂，有失真、有逼真，总的来说是遵循如果要让重采样后更逼真，就得用相对复杂算法，消耗更多 CPU。对这些算法，按产生的新样本是否最多只涉及到同一条直线上的两点线性权值分为线性重采样和非线性重采样。

相比非线性重采样，线性重采样算法较为简单，它利用在同一条直线上两点的线性权值来逼近所需要采样点的幅值，其计算量小，任意点的插值最多需要相邻两点幅度值，然后将其幅值进行线性加权，其中权值如何选择是这种重采样算法的关键所在。算法计算某个输出样本时最多涉及两个输入样本，“窗”长度仅是 2，几乎不存在延时，又因为计算量小，相比非线性重采样它较少花费时间开销。

有一种重采样它不产生新样本。像要把 8K 的音频数据转变成 44.1K 时，它就是用原来 8K 中的数据重复填充以补足 44.1K，要把 48K 的音频数据转变成 44.1K 时，则只是从 48K 中的数据抽取以减少到 44.1K，它不执行什么加权，这么多方法中它的计算量可说最小，但失真最厉害，这种方法叫无增重采样。

11.2.1 音频

1、进入<data>/sounds，把 button.wav 改名为 button_.wav。复制 join.wav，在同目录下“粘贴”，新生成文件名改为 button.wav。
2、在 kingdom 工程中打开 SDL 工程中的 SDL_audiotypecvt.c，在 SDL_Downsample_S16LSB_2c 内设断点。
3、运行时选“Debug”——“Starting Debug”。
4、进入标题屏幕后随便按哪个键，这时会触发断点。

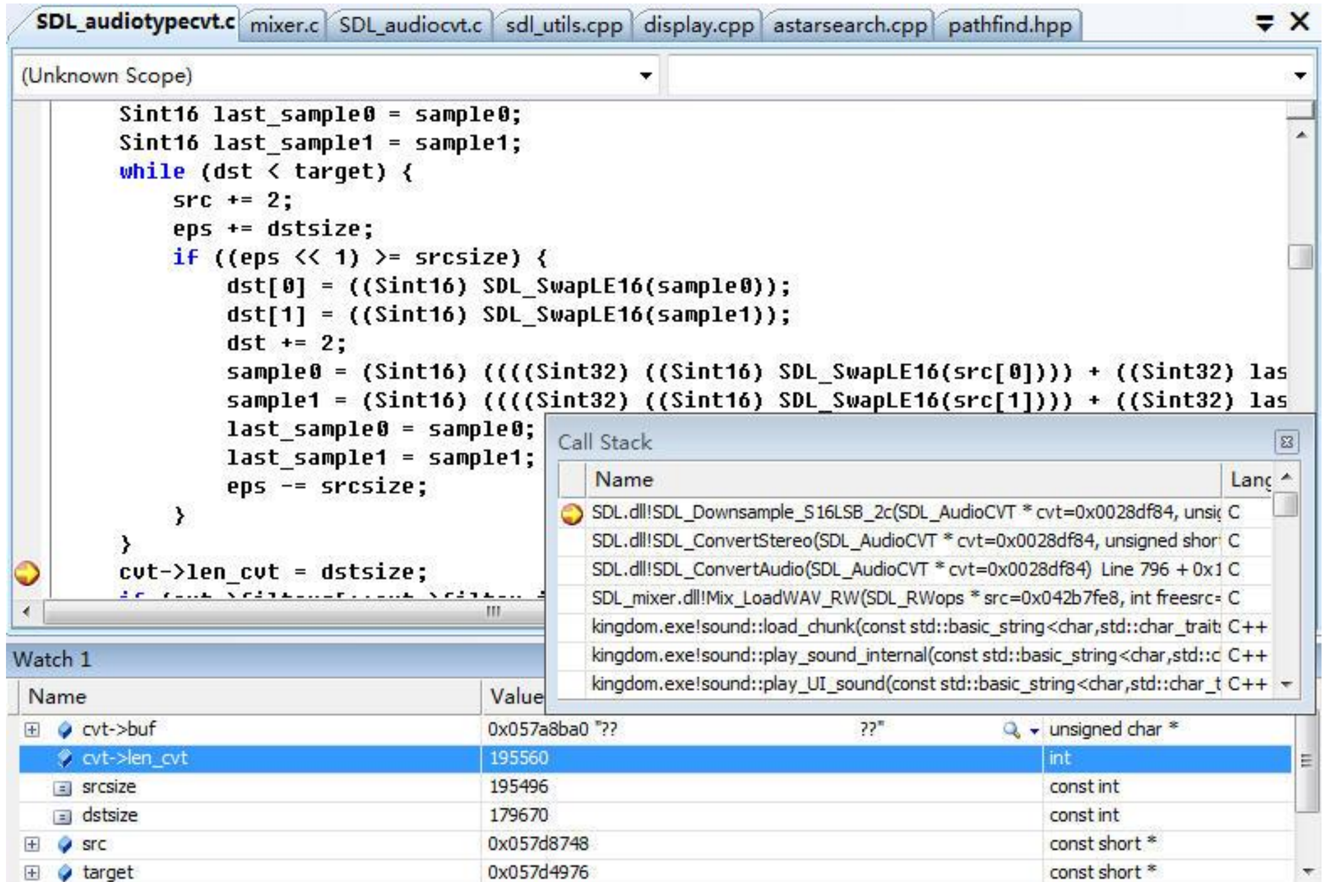


图 10-4 SDL_Downsample_S16LSB_2c

SDL_Downsample_S16LSB_2c 针对是双声道、样本长度 16 位的声音数据，因而对于连续四个字节来说，当中存着两个样本，一个左声道、一个右声道。第一个是左声道还是右声道，这往往是不确定的，只要记住如果第一个是左声道，那么此串数据中所有奇数样本都需要属于左声道。由于左、右声道样本总是成对出现，以下叙述时把一对左、右声道样本称为样本对。

图 10-4 显示 SDL 把采样率 48K 的音频数据转为 44.1K，当中 srcsize、dstsize 是指示字节数而不是样本数。代码执行的是下采样，对下采样可归纳为两个问题：如何计算输出样本幅值；如何控制输出样本数量。

如何计算输出样本幅值

SDL_Downsample_S16LSB_2c 属于线性重采样，用“相邻”两样本的线性权值来计算输出样本幅值。幅值计算分两种情况。

第一对输出样本

```
Sample#0_L = src[0]
Sample#0_R = src[1]
```

其它输出样本对

```
Sample#N_L = (src[(N - 1) * 2] + src[N * 2]) / 2
Sample#N_R = (src[(N - 1) * 2 + 1] + src[N * 2 + 1]) / 2
```

计算使用的权值是 0.5，简单说第 N 个输出数据就是输入中“相邻”的第 N-1 个和第 N 个样本对的平均值。至于“相邻”为什么要加引号，这是由于要下采样，它必须扔弃一些样本对，一旦出现扔弃，紧接它之后那个计算出的输出样本对用的其实是两个不相邻的输入样本对。

如何控制输出样本数量

从 48K 到 44.1K，执行这个下采样一个目标是要让形成的 44.1K 尽可能逼真重现 48K 声音。由于输出样本数小于输入，必须执行扔样本，扔时需尽可能做到在整个过程均匀地扔。那如何做到均匀，能整除时，像 48K 到 24K，可使用见两对扔一对，那不能整除时，像 48K 到 44.1K，

SDL_Downsample_S16LSB_2c 是使用减步长办法。

步长=输入字节数-输出字节数

此例中“步长 = srcsize - dstsize”，即“195496 - 179670 = 15826”。

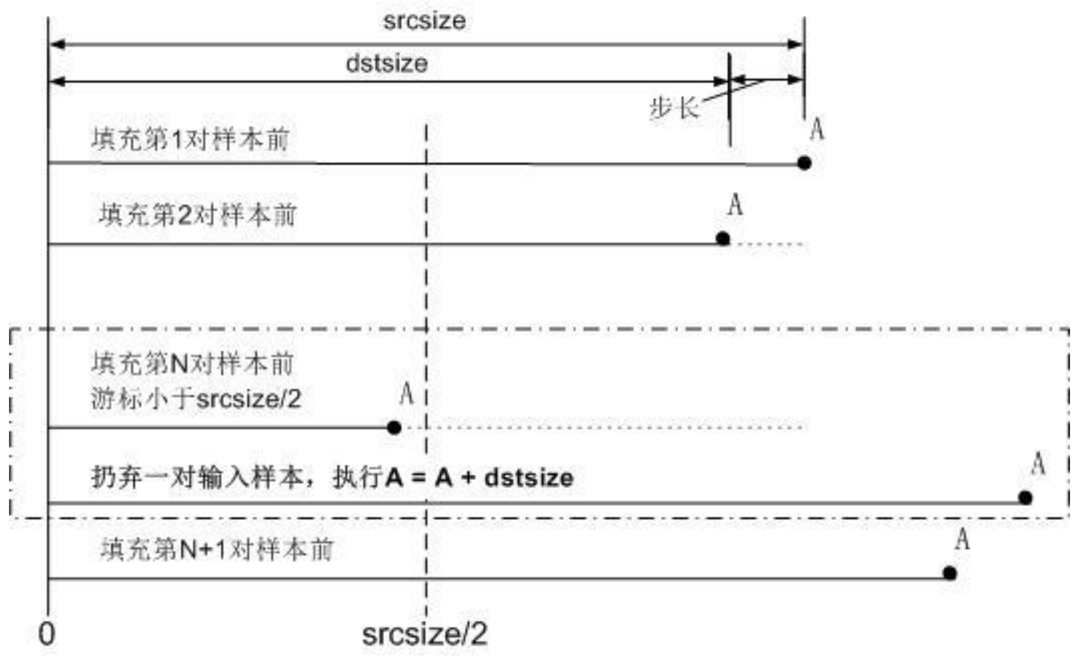


图 10-5 SDL_Downsample_S16LSB_2c-graph

大概策略

1. 每填充完一对样本后，更新游标（图 10-4 中的 A，图 10=3 中的 eps 变量）到新位置，eps-=步长。
2. 一旦发现游标小于 srcsize/2，认为此次要扔样本，同时更新游标到新位置，eps+=dstsize。

此种扔样本策略下，隔填充几次扔一次不是一个固定值，但平均来说是个固定值，并且输入样本长度和输出样本长度决定了这个固定值。图 10-4 中，固定值=输入样本数/减少样本数=srcsize/(srcsize-dstsize)=12.35，也就是说大概是隔 12 次填充扔一对样本。

注意图 10-4 中 srcsize 是如何取值，“srcsize = cvt->len_cvt - 64”，理论上说 srcsize 应该等于 cvt->len_cvt，那减去 64 对算法有何影响？根据图 10-5，srcsize 只在判断何时扔样本期间发生作用，64 这个减量越大，srcsize 越小，从而步长越小，srcsize/2 也越小，但对判断是否要扔来说步长是个累积值，由步长变小产生的影响要大于 srcsize/2 变小产生的影响，于是需要隔更多填充次数才扔一次。而填充次数是固定值，也就意味着过程中“本来”应该被扔的没被扔，结果倒致更多的后面声音被消失。

11.2.2 图像

相对来说，图像重采样要比声音复杂，因为图像是二维的而声音只有一维。

1、打开 sdl_utils.cpp，在 scale_surface 内设断点。
2、运行时选“Debug”——“Starting Debug”。会触发断点。

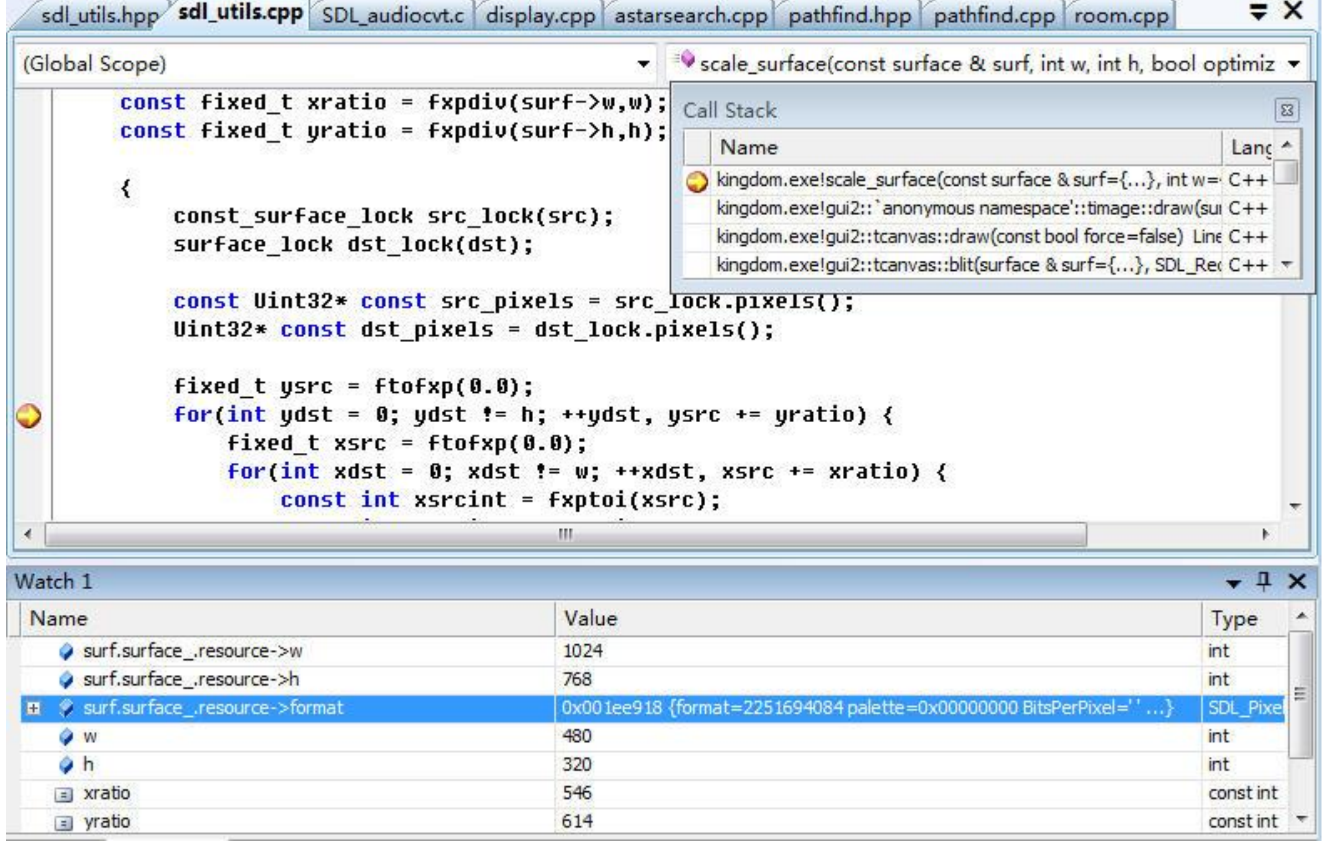


图 10-6 scale_surface

图 10-6 显示 scale_surface 如何把分辨率是 1024x768 图像缩小成 480x320。代码执行的是下采样，和音频下采样一样，对图像下采样也可归纳为两个问题：如何计算输出样本幅值；如何控制输出样本数量。

如何计算输出样本幅值

10.2.1 中计算音频样本幅值使用相邻样本平均值，即涉及到的是相邻的两个样本、使用的权值都是 0.5，这样两像素均值计算法可用以下公式表示。

```
s=A1*s1 + A2*s2
```

1. s：输出样本幅值
2. A1、s1：第一样本权值、幅值
3. A2、s2：第二样本权值、幅值

音频是一维数据，一旦变成二维图像，基于以上这种思路，很快可想到一种计算输出图像像素幅值方案，使用四个相邻（00, 01, 10, 11）像素，取它们的平均值。

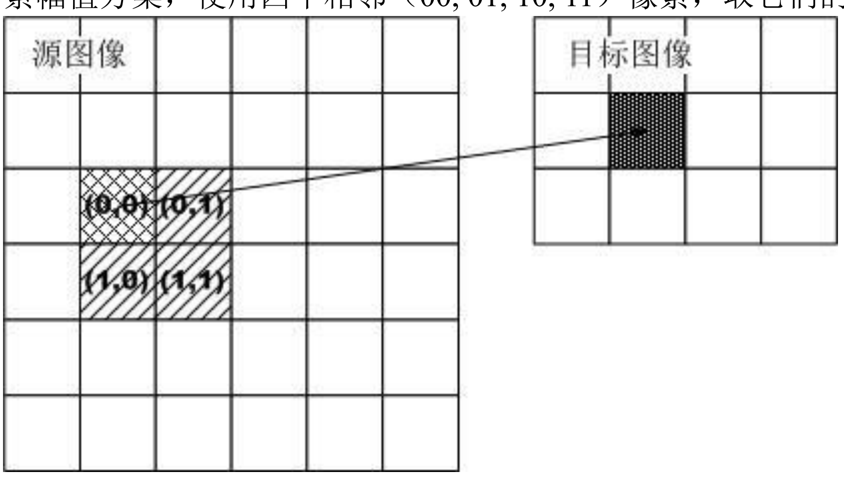


图 10-7 scale_surface graph

```
s=A1*s1 + A2*s2 + A3*s3 + A4*s4
```

- s：输出像素幅值。像素幅值分四个分量：A、R、G、B。
- A1、s1：第一像素权值、幅值
- A2、s2：第二像素权值、幅值
- A3、s3：第三像素权值、幅值
- A4、s4：第四像素权值、幅值

按以上公式，接下就是要找出这个四像素以及对应权值。

公式之四个源像素

```
const int dx = (xsrcint + 1 < src->w) ? 1 : 0;
const int dy = (ysrcint + 1 < src->h) ? src->w : 0;
......
const Uint32* const src_word = src_pixels + ysrcint*src->w + xsrcint;
......
pix[0] = *src_word;            // northwest
pix[1] = *(src_word + dx);      // northeast
pix[2] = *(src_word + dy);      // southwest
pix[3] = *(src_word + dx + dy);  // southeast
```

结合图 10-7，不难理解以上代码。

公式之四个权值

根据图 10-7，对权值应该怎么取会有个“直观”方法，越接近“真实”点的像素权值越大。“真实”点的意思是根据缩放比例计算出的目标像素在源图像上对应的点。举个例子，要把一个 24x24 图像缩小成 12x12，当要计算目标图像中的(2, 3)这个像素时，它换算到源图像中对应的位置是(4, 6)，(4, 6)就是那个“真实”点。对于取权值来说，(4, 6)应该是 100%，而其它三像素(5, 6)、(4, 7)、(5, 7)权值都该是 0。24x24 到 12x12 因为可以整除，它说来较直观，让举个不能整数例子，图 10-7 的 6x6 缩小到 4x3，当要计算目标图像中的(1, 1)这个像素时，它换算到源图像中对应的“真实”点是(1.5, 2)，那对于取权值来说，(1, 2)、(2, 2)应该是 0.5，而其它两像素(1, 3)、(2, 3)应该是 0。

由以上分析可得到个算法，**使用缩放比例值的小数部分来决定权值。**

要支持小数则意味着程序必须用 float/double，相比于整数运算，浮点计算量倍增，那能不能不使用精度类型也实现类似小数功能？——可以把比例值放大一定倍数，像 256，这样一来部分本应属于小数的数值就会上升到整数。

```
const fixed_t e = 0x000000FF & xsrc;
const fixed_t s = 0x000000FF & ysrc;
const fixed_t n = 0xFF - s;
const fixed_t w = 0xFF - e;
......
bilin[0] = n*w;
bilin[1] = n*e;
bilin[2] = s*w;
bilin[3] = s*e;
```

分析下以上代码。放大倍数是 256，即 0xff，语句“e = 0x000000FF & xsrc”功能是计算出 xsrc 这 x 坐标中本属于小数、可因为放大 256 而升到整数部分。“w = 0xff - e”功能是计算出 e 的“互补”值，0xff 是放大倍数，如果把 e 认为是小数，这语句等同“w = 1 - e”。由于水平上只涉及两个像素去加权，一个加权是 N%，另一个自然是（100-N）%。同样分析也适用于 s、w。

e、s、n、w 只是一个方向上权值，对四个像素中一个来说，它在水平、垂直上都会发生作用，这时就要把它们各自一维权值相乘得到二维权值，像(0, 0)，它就是 n*w。而得出的二维权值就是参与最终计算时四像素对应权值，也就是说 bilin 存储的就是对应四个像素权值。

以上分析或许有点抽像，让具体代入个例子。图 10-7 的 6x6 缩小到 4x3，当要计算目标图像中的(1, 1)这个像素时，它换算到源图像中对应的“真实”点是(1.5, 2)，乘上放大因子后是(384, 512)。

```
const fixed_t e = 0x000000FF & 0x180 = 128;
const fixed_t s = 0x000000FF & 0x200 = 0;
const fixed_t n = 0xFF - 0 = 256;
const fixed_t w = 0xFF - 128 = 128;
bilin[0] = n*w = 256 x 128 = 32768;
bilin[1] = n*e = 256 x 128 = 32768;
bilin[2] = s*w = 0;
bilin[3] = s*e = 0;
```

bilin 存的是放大 65536(256x256)倍后的权值（相乘是因为二维），最后加权计算时需要除去放大倍数回归到小数，通过验证可得出这个计算出的权值就是之前“直观”分析出的值。

```
bilin[0] = 32768 / 65536 = 0.5;
bilin[1] = 32768 / 65536 = 0.5;
bilin[2] = s*w = 0;
bilin[3] = s*e = 0;
```

如何控制输出样本数量

回看图 10-6，scale_surface 根据要输出分辨率，由上到下、由左到右使用两个嵌套的 for 循环实现逐个计算目标像素，在计算每个目标像素时会通过比例值（xratio、yratio）“自动”找出源图像中和它对应四像素。

两个 for 循环“合”起来的次数保证了输出样本数量。

11.3 处理 iOS 事件

在 Apple 官方出的《iPhone 应用程序编程指南》中有个章节专门讲述事件处理，那里把事件分为触摸事件和运动事件。触摸事件基于多点触摸模型，用户不是通过鼠标和键盘，而是通过触发设备的屏幕来操作对象、输入数据、以及指示自已的意图；运动事件则源自设备加速计，当用户以特定方式移动设备，比如摇摆设备时，iPhone/iTouch/iPad 会产生运动事件。这两类事件可归为输入事件，它们类似于 PC 上的鼠标、键盘输入设备产生事件，而在 PC，还有一类事件，像系统重启，程序主窗口被最小化，它们被称为系统事件，相应地在 iOS 上也存在系统事件，像按一下“Home”按键要返回桌面，电话到来要中断当前程序，用户选择忽略新电话重新恢复程序。由于这些事件往往导致正在运行程序被中断，在有地方称处理这些事件为响应中断。

在这里把系统事件和输入事件合称为事件，要合为一种的主要原因是 C/C++程序上以着类似的方法处理它们，而处理代码主要在 SDL 中，为区别于另一类事件，往往称它们为 SDL 事件。什么是另一类事件呢？对一个历史模拟游戏，程序中事件可分为两种：SDL 事件和游戏事件。SDL 事件指由 SDL 系统产生的事件，就是以上说的输入事件、系统事件。游戏事件指玩游戏时产生的事件，像攻战了某一城市，某一部队被消灭。本节叙述“处理 iOS 事件”中的事件指的是 SDL 事件。

11.3.1 SDL 处理 iOS 事件通用逻辑

为更清晰分清此个处理逻辑，让进入 iOS 模拟器进行调试。

1、打开 SDL 工程的 SDL_uikitview.m，在 touchesBegan::withEvent(...)内设断点。

2、运行时选“Run”——“Debug Breakpoints On”。

3、进入标题屏幕后单击鼠标（类似真实设备上触碰事件），这时会触发断点。

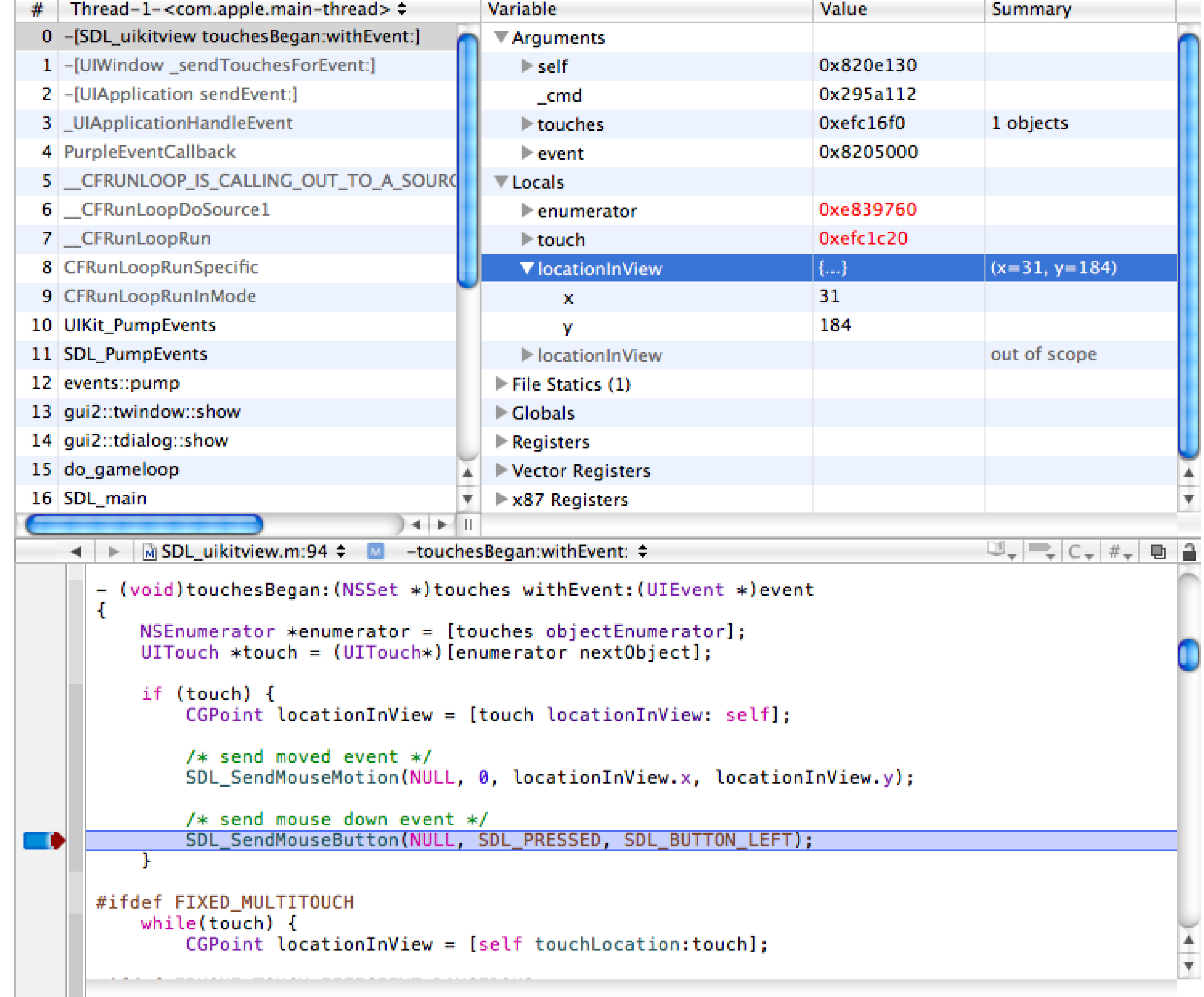


图 10-8 touchesBegan::withEvent

图 10-8 显示在 iOS 平台下 SDL 如何收集触摸事件。在此让回顾下“第二章 SDL”中的“2.5.1 处理事件”，以下是那里对 SDL_PumpEvents 描述。

1. 在主线程（Main Thread），应用程序调用自个实现函数 events::pump，后者调用 SDL 提供的 API：SDL_PumpEvents。
2. SDL_PumpEvents 是个跨平台收集事件函数，它要实现收集需调用各操作系统下“真正”的收集函数，Windows 系统就是 WIN_PumpEvents。

以上两点是针对 Windows 平台下的 SDL_PumpEvents，在 iOS 平台下要对第二条做出修改：SDL_PumpEvents 是个跨平台收集事件函数，它要实现收集需调用各操作系统下“真正”的收集函数，iOS 系统就是 UIKit_PumpEvents。

图 10-8 中，touchesBegan 解析出坐标后，依次调用 SDL_SendMouseMotion 把鼠标移动（SDL_MOUSEMOTION）放入事件队列，SDL_SendMouseButton 则是把鼠标按下（SDL_MOUSEBUTTONDOWN）放入队列（结合“第二章 SDL”中的“2.5 处理事件）。以上就是 SDL 收集 iOS 事件的主要逻辑，对于处理事件另个任务：检索，iOS 平台和 Windows 平台几乎一模一样。检索基于事件队列（SDL_EventQ），这队列是 SDL 基于各平台抽象出的一个数据结构，基于该数据结构的检索代码也是通用于各平台。

11.3.2 setjmp、longjmp

在这里让具体看下 UIKit_PumpEvents，以下是该函数代码。

```
void UIKit_PumpEvents(_THIS)
{
    if (setjmp(*jump_env()) == 0) {
        /* if we're setting the jump, rather than jumping back */
            SInt32 result;
        do {
            result = CFRunLoopRunInMode(kCFRunLoopDefaultMode, 0, TRUE);
        } while (result == kCFRunLoopRunHandledSource);
    }
}
```

CFRunLoopRunInMode 是 iOS 提供的一个 API，用它来监控默认模式（kCFRunLoopDefaultMode）下的输入源，对触摸事件，它最终会调用 touchesBegian::withEvent，SDL 重载后者，在自代码中实现把触摸事件放入事件队列（SDL_EventQ），从而实现了 UIKit_PumpEvents 的收集事件功能。也就是说，要是单以实现收集事件到事件队列这功能来说，UIKit_Events 只要有那个调用 CFRunLoopRunInMode 的 while 循环就可以了，没必要存在 setjmp(*jump_env())。

UIKit_PumpEvents 为什么要存在 setjmp(*jump_env())？——iOS 没给程序以太多时间来响应中断（处理系统事件），对于系统事件，iOS 要求应用程序"绝对"服从，即只要按下"Home"按键，不管应用程序当前是什么状态，必须中断自已以能让系统显示桌面。响应中断须"绝对"服从，这更多是体现在用户操作观点，以程序代码观点来说，在发生系统事件时，iOS 使用了输入事件不同环境，**这个环境造成了程序不能以正常的栈递归方式回到上层。**

什么是"程序不能以正常的栈递归方式回到上层"？让看下图 10-8，按照 C/C++函数调用规则，一旦 touchesBegan::withEvent 执行完后，控制权就返回到它的直接上层 UIWindow _sendTouchesForEvent，后者一旦执行完，按制权则返回到它的直接上层 UIApplication sendEvent，如此一层层向上，控制权就可到达显示的最顶层函数 SDL_main，这种方式称之为正常的栈递归方式回到上层。有了这个概念，"程序不能以正常的栈递归方式回到上层"指的就是这种递归方式被破坏了，底层函数已不能通过层层向上方式把控制权交给上层。这种破坏具体表现出什么，让进入调试。

1、打开 SDL 工程的 SDL_uikiappdelegate.m，在 application::applicationWillResignActive(...)内设断点。

2、运行时选"Run"——"Debug Breakpoints On"。

3、进入标题屏幕按下"Home"按键（类似真实设备上按下"Home"按键），这时会触发断点。

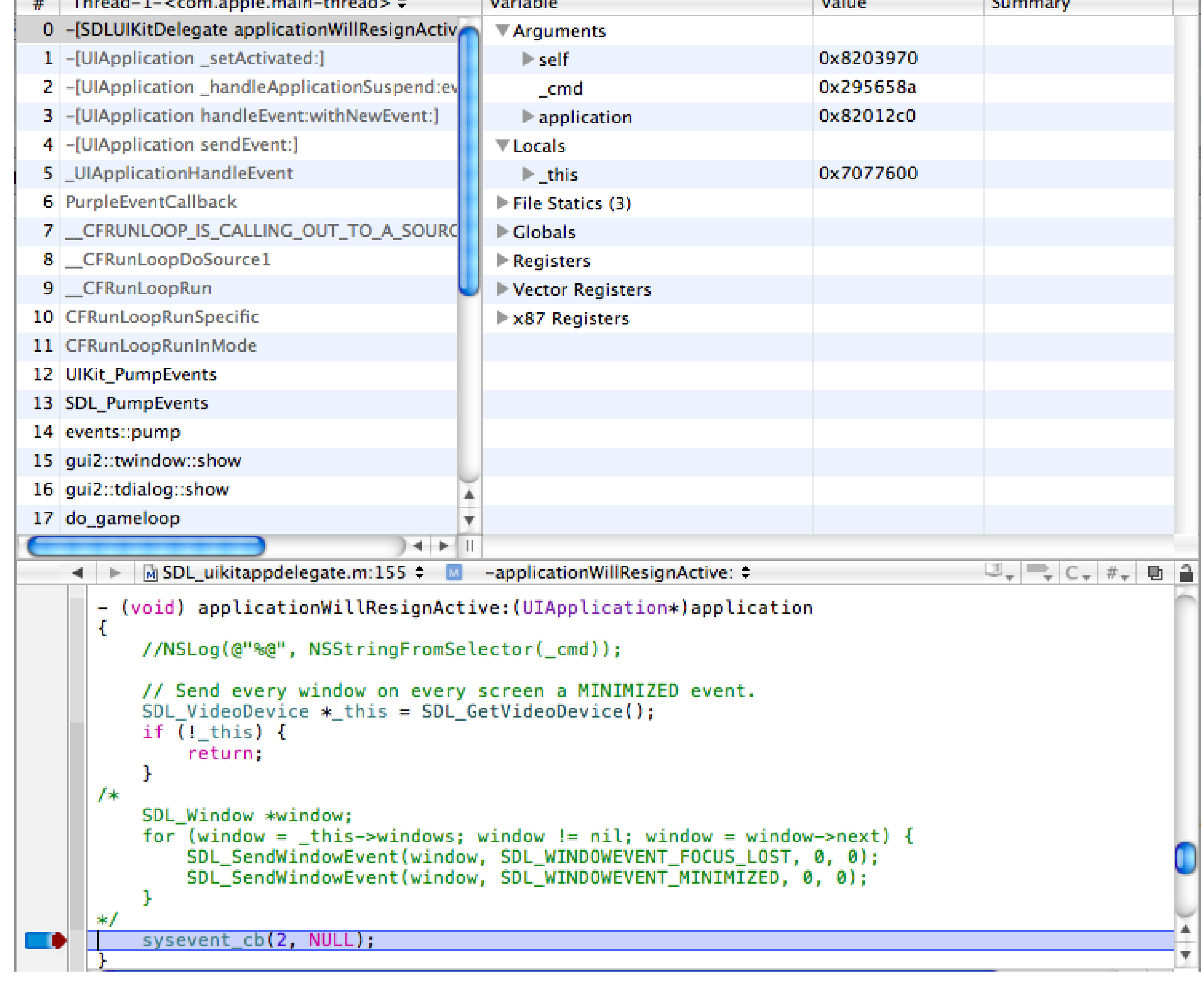


图 10-9 applicationWillResignActive

注 1：application::applicationWillResignActive 是可以以正常的栈递归方式回到上层，不能的是 applicationWillTerminate，但用 xcode 很难捕抓到后者的函数栈。这里使用 applicationWillResignActive 是因为它的函数栈和 applicationWillTerminate 较相像，而且它们都有一个特点，一旦发生后 C++异常处理机制将不能正常工作。

图 10-9 显示 SDL 如何处理按下"Home"按钮要求回到桌面这个系统事件。结合图 10-8 对比两个图的函数栈，会看到处理这两个事件时函数栈无非就是最顶上一、两个函数不同而已，但这只是表示在显示上，实际 iOS 给他们生成了两个环境：图 10-8，程序可以以正常的栈递归方式回到上层；图 10-9（严格说是 applicationWillTerminate 时函数栈，见注 1），程序不可以以正常的栈递归方式回到上层。我不清楚图 10-9 中是哪个函数破坏了这个递归规则，但这似乎不重要，重要的是这个结论。

对于 iOS 区别对待输入事件、系统事件，在 iOS 编程指南中已或多或少能看到些端倪。像应用将被终止时，系统会调用应用代理的 applicationWillTerminate 方法以使应用能做退出前的任何需要的回收处理。为此，应用可重载 applicationWillTerminate 来保存用户数据或应用状态信息，以供应用随后重新启动恢复状态时使用。该方法最长运行时限为 5 秒，过期应用即被 kill 掉并且移除内存。

iOS 要求的处理系统事件机制使得"程序不能以正常的栈递归方式回到上层"，也就是说依着正常方式从 applicationWillTerminate 是没法回到 main 函数了，可要是不回到 main 函数，

C/C++没法正常退出程序，那该怎么办呢？这会让想到采用 C/C++异常处理机制。

C/C++异常处理机制有两种：try、throw、catch 的 C++异常处理机制，setjmp、longjmp 的 C 异常处理机制。要采用哪一种机制，对于 C/C++混合编程来说自然是第一种，因为在捕获到异常时 setjmp、longjmp 无法做到自动析构类（这些类定义在函数栈，按正常规则一旦函数返回该类会被自动析构）（实测下来不是所有 C 编译器都如此，但至少有编译是如此，像 Apple GCC）。可让人不幸的，iOS 构建的处理系统事件机制除了使“程序不能以正常的栈递归方式回到上层”，还使 try、throw、catch 这种 C++异常处理机制没法正常工作（其它情况下能够正常工作）！还好，setjmp、longjmp 能正常工作，也就是说要让返回到 main 只能靠 setjmp、longjmp。

setjmp、longjmp 无法做到自动析构类，这就让代码受到很多限制。如果不想受太多限制，那可以采取种偷懒办法，一旦发生系统事件，不管三七二十一直接用 longjmp 跳到 main，进而退出程序。一旦退出程序，系统会清光该程序所占内存，当然，这种办法是以损失好的用户体验以及不符合 iOS 对应用程序行为要求为代价。

小结

- iOS 构建的处理系统事件机制除了使“程序不能以正常的栈递归方式回到上层”（只是 applicationWillTerminate），还使 C++异常处理机制没法正常工作（applicationWillResignActive/applicationWillTerminate）。
- 使用 setjmp、longjmp 异常处理机制可以实现把控制权从中断现场返回到 main。UIKit_PumpEvents 中的 setjmp(*jump_env())就是 SDL 为让应用程序能回到 main 而引入的。

11.4 使用 Boost

11.4.1 嵌入自已的工程编译

嵌入自已的工程编译 Boost 指的是直接把 Boost 相关 cpp 添加到自已工程中，作为自已工程内的 cpp 进行编译。

为什么要嵌入自已的工程编译？

- **同一平台上，简化 Boost 编译过程**
  要使用 Boost，必须要能把相关 Boost 库代码链入自个程序，这个链入有两种方式：以库的方式，以源代码方式。以库方式链入时，编译 Boost 有两种方法，完全编译和部分编译。完全编译指的是编译整个 Boost、形成所有子库，这个编译往往需要一段很长过程；部分编译指的是只编译相关子库，但不论是完全编译还是部分编译，它们都使用 Boost 提供的 bjam，最后都形成一个个子库，以库的方式链入最后工程。Boost 虽然可以编译单个子库，但链入工程编译/链接选项多个时它必须为每种设置都提供一个库，像静态单线程、动态单线程，静态多线程、动态多线程，为此就要提供四个库。而以嵌入工程方式时以着同一条“Build”命令就可自动自成。
- **不同平台上，方便统一管理在用的 Boost**
  以库方式链入时，就先需要编译出相关 Boost 子库，而不同平台是采用不同方法编译 Boost 子库。想象下这种情况，你哪天想加个子库，要是采用库链入工程方式，意味着每个平台都要来一次编译此个子库，这种分散编译不易于在自个工程中管理 Boost。

如何嵌入自已的工程编译？

1. 对于使用 GCC 场合（包括 Mac OS X、iOS、Android），只需把 Boost 源码加入工程，不必对编译、链接选项做额外设置。
2. 对于使用 Microsoft C 场合（Visual Studio），使用嵌入工程编译方时需定义宏 BOOST_ALL_NO_LIB 或者 BOOST_XXX_NO_LIB（XXX 是某个库的名称），以指示 Boost

库不要使用自动链接功能。如果工程内有多个项目，只要是调用到 Boost 库的最好都定义上该宏，即包括只调用 hpp 的。

3. 对需要组件，直接把相关 cpp 拉入工程。Boost 的 cpp 位在<boost_x_xx>/libs 目录下，在 Boost 内它们被以组件进行分类，像 iostreams 组件的 cpp 是位在<boost_x_xx>/libs/iostreams/src。
4. Boost 中的 iostreams 组件若要支持更多压缩方式需额外库，像 gzip 须 zlib 库，bzip2 须 bzip2 库，这些库源码可从它们官方网站下载。**注**：zlib128 在 XCode 5.1 会报错，须修改 zconfig.h，参考：XCode 5.1：Implicit declaration of function 'lseek' is invalid in C99