# 欢迎使用 KiCad 开发人员文档

此页面仅供开发人员使用，如果您只是想使用 KiCad，这不是您想要的。

# 编译

## 从源代码编译 KiCad

如果您是用户而不是开发人员，请考虑使用其中一个预建的软件包 可在 KiCad 网站 找到 KiCad 的 下载 页面。从源代码编译 KiCad 不适合无经验的人，除非您有合理的软件开发经验，否则不建议这样做。本文档介绍了如何在支持的平台上从源代码 编译 KiCad。 它的目的不是作为安装或编译的指南 [库依赖项](#library_dependencies)。 编译库依赖项时， 请参考您的平台文档以安装软件包或源代码。 目前支持的平台是 Windows 版本 7-10，几乎任何版本的 Linux 和 MacOS 10.9-10.13。 您可能能够在其他平台上编译 KiCad，但不支持它。 在 Windows 和 Linux 上 GNU GCC 是唯一受支持的编译器而在 MacOS 上 Clang 是唯一受支持的编译器。

## 页面

- 入门
  *开发工具 在开始编译 KiCad 之前，除了编译器之外，还需要一些工具。 这些工具中有些是从源代码编译所必需的，有些是可选的。 CMake 编译配置工具 CMake 是 KiCad 使用的编译配置和 Makefile 生成工具。这是必需的。 Git 版本控制系统 官方的源代码仓库托管在 GitLab 上，需要 git 获取最新的源代码。 如果你更喜欢使用 GitHub，有一个官方 KiCad 仓库的只读镜像。 以前的官方托管地点 Launchpad 仍作为镜像活跃。 更改应作为 合并请求 通过 GitLab 提交。 开发团队不会审查在 GitHub 或 Launchpad 上提交的更改，因为这些平台只是镜像。 Doxygen 代码文档生成器 KiCad 源代码使用 Doxygen 解析 KiCad 源代码文件，并将依赖关系树与源文档一起编译为 HTML。 只有在您要编译 KiCad 文档时才需要 Doxygen。 SWIG 简化的包装器和界面生成器 SWIG 是用于生成 KiCad 的 Python 脚本语言扩展。 如果您不打算编译 KiCad 脚本扩展，则不需要 SWIG。 库依赖项 本节包括编译 KiCad 所需的库依赖项列表。 它不包括库的任何依赖项。 有关任何其他依赖项，请参阅库文档。 其中一些库是可选的，具体取决于您的编译配置。 这不是关于如何使用系统包管理工具安装库依赖项或如何从源代码编译库的指南。 要执行这些任务，请参阅相应的文档。*
- Linux
  *使用 gcc 的 Linux 使用说明*
- macOS
  *macOS 使用 cmake 和 clang 的说明*
- Windows (MSYS2)
  *使用 MSYS2 编译 KiCad 的指南*
- Windows (Visual Studio)
  *使用 Microsoft Visual Studio 和 vcpkg 编译 KiCad 的指南*
- 编译选项
  *通过 CMake 配置编译选项摘要*

# 入门

## 开发工具

在开始编译 KiCad 之前，除了编译器之外，还需要一些工具。 这些工具中有些是从源代码编译所必需的，有些是可选的。

### CMake 编译配置工具

CMake 是 KiCad 使用的编译配置和 Makefile 生成工具。这是必需的。

### Git 版本控制系统

官方的源代码仓库托管在 GitLab 上，需要 git 获取最新的源代码。 如果你更喜欢使用 GitHub，有一个官方 KiCad 仓库的只读镜像。 以前的官方托管地点 Launchpad 仍作为镜像活跃。 更改应作为合并请求 通过 GitLab 提交。 开发团队不会审查在 GitHub 或 Launchpad 上提交的更改，因为这些平台只是镜像。

### Doxygen 代码文档生成器

KiCad 源代码使用 Doxygen 解析 KiCad 源代码文件，并将依赖关系树与源文档一起编译为 HTML。只有在您要编译 KiCad 文档时才需要 Doxygen。

### SWIG 简化的包装器和界面生成器

SWIG 是用于生成 KiCad 的 Python 脚本语言扩展。 如果您不打算编译 KiCad 脚本扩展，则不需要 SWIG。

## 库依赖项

本节包括编译 KiCad 所需的库依赖项列表。 它不包括库的任何依赖项。 有关任何其他依赖项，请参阅库文档。 其中一些库是可选的，具体取决于您的编译配置。 这不是关于如何使用系统包管理工具安装库依赖项或如何从源代码编译库的指南。 要执行这些任务，请参阅相应的文档。

# wxWidgets 跨平台 GUI 库

wxWidgets 是 KiCad 使用的图形用户界面 (GUI) 库。 当前的最低版本是 3.0.0。 但是，应该尽可能使用 3.0.2，因为以前版本中的一些已知错误可能会在某些平台上导致问题。 请注意，在从源代码编译 wxWidget 之前，还必须应用一些特定于平台的补丁。 这些补丁程序可以在 KiCad 源代码的 patches 文件夹 中找到。 这些补丁由应该应用它们的 wxWidgets 版本和平台名称命名。wxWidget 必须使用 --with-opengl 选项编译。 如果您的系统上安装了 wxWidgets 的打包版本，请验证它是否使用此选项编译。

# Boost C++ 库

仅当您打算使用系统安装的 Boost 版本而不是默认的内部编译版本编译 KiCad 时，才需要 Boost C++ 库。 如果使用系统安装的 Boost 版本，则需要 1.56 或更高版本。请注意，编译有效的 Boost 库需要一些特定于平台的补丁。 这些补丁程序可以在 KiCad 源代码的 patches 文件夹 中找到。 这些补丁程序按应应用它们的平台名称命名。

# GLEW OpenGL Extension Wrangler 库

OpenGL Extension Wrangler 是 KiCad 图形抽象库 [GAL] 使用的 OpenGLhelper 库， 在编译 KiCad 时总是需要的。

# ZLib 库

KiCad 使用 ZLib 开发库来处理压缩的 3D 模型 (.stpz 和 .wrz 文件)， 并且在编译 KiCad 时始终需要该开发库。

# GLM OpenGL Mathematics 库

OpenGL Mathematics 库 是由 KiCad 图形抽象库 [GAL] 使用的 OpenGLHelper 库， 并且总是编译 KiCad 所必需的。

# GLUT OpenGL 实用程序工具箱库

OpenGL Utility Toolkit 是 KiCad 图形抽象库 [GAL] 使用的 OpenGL 辅助程序库，在编译 KiCad 时始终需要它。

# Cairo 2D 图形库

Cairo 不可用时，OpenGL2D 图形库用作备用渲染画布，并且始终需要它来编译 KiCad。

# Python 编程语言

Python 编程语言用于为 KiCad 提供脚本支持。 除非禁用 [KiCad 脚本](#kicad_script) 编译配置选项，否则需要安装它。

# wxPython 库

wxPython 库用于为 pcbnew 提供脚本控制台。 除非禁用 [wxPython脚本](#wxpython_script) 编译配置选项，否则需要安装它。 在支持 wxPython 的情况下编译 KiCad 时，请确保 wxWidgets 库的版本与系统上安装的 wxPython 版本相同。 已知不匹配的版本会导致运行时问题。

# Curl 多协议文件传输库

Curl 多协议文件传输库 用于为 [GitHub][] 插件提供安全的互联网文件传输访问。 除非禁用 GitHub Plug 编译选项，否则需要安装此库。

# OpenCascade 库

OpenCascade 社区版 用于提供对加载和保存 3D 模型文件格式(如 STEP)的支持。 除非禁用 OCE 编译选项，否则需要安装此库。

Open CASCSADE 技术 (OCC) 也应该作为 OCE 的替代方案。 可以在编译时指定库级联库的选择。参见[STEP/IGES 支持](#OCE_OPT) 部分。 使用选项 BUILD_MODULE_DRAW=OFF 编译 OCC 时，使编译更加容易

## Ngspice 库

Ngspice 库 用于在原理图编辑器中提供 SPICE 模拟支持。 确保使用的 ngspice 库版本是使用 --with-ngshare 选项编译的。除非禁用 Spice 生成选项，否则需要安装此库。

# KiCad 生成配置选项

KiCad 有许多编译选项，可以配置这些选项来编译不同的选项，具体取决于给定平台上对每个选项的支持情况。 本节记录这些选项及其默认值。

# 脚本支持

KICAD_SCRIPTING 选项用于将 Python 脚本支持编译到 Pcbnew 中。 默认情况下，此选项处于启用状态，并且在禁用时将禁用所有其他 KICAD_SCRIPTING_* 选项。

# Python 3 脚本支持

KICAD_SCRIPTING_PYTHON3 选项用于启用使用 Python 3 而不是 Python 2 进行脚本支持。 默认情况下，此选项处于禁用状态，仅当启用 [KICAD_SCRIPTING](#scripting_opt) 时才相关。

# 脚本模块支持

KICAD_SCRIPTING_MODULES 选项用于支持编译和安装 KiCad 提供的 Python 模块。 默认情况下，此选项处于启用状态，但如果禁用 [KICAD_SCRIPTING](#scripting_opt)，则此选项将被禁用。

# wxPython 脚本支持

KICAD_SCRIPTING_WXPYTHON 选项用于将 wxPython 接口编译到 Pcbnew 中，包括 wxPython 控制台。 默认情况下，此选项处于启用状态，但如果禁用 [KICAD_SCRIPTING](#scripting_opt)，则此选项将被禁用。

# wxPython Phoenix 脚本支持

KICAD_SCRIPTING_WXPYTHON_PHOENIX 选项用于使用新的 Phoenix 绑定(而不是旧的绑定)编译 wxPython 接口。 默认情况下该选项处于禁用状态，启用该选项需要启用 [KICAD_SCRIPTING](#scripting_opt)。

# Python 脚本操作菜单支持

KICAD_SCRIPTING_ACTION_MENU 选项允许将 Python 脚本直接添加到 Pcbnew 菜单。 默认情况下，此选项处于启用状态，但如果禁用 [KICAD_SCRIPTING](#scripting_opt)，则此选项将被禁用。请注意，此选项是高度实验性的，如果 Python 脚本在 Pcbnew 中创建无效的对象状态，可能会导致 Pcbnew 崩溃。

# 集成 Spice 仿真器

KICAD_SPICE 选项用于控制是否为 EesChema 编译 Spice 仿真器接口。启用此选项时，它要求 [ngspice][] 作为共享库可用。 默认情况下，此选项处于启用状态。

# 对 3D 查看器的 STEP/IGES 支持

KICAD_USE_OCE 用于 3D 查看器插件以支持 STEP 和 IGES 3D 模型。 此选项启用与 OpenCascade Community Edition(OCE) 相关的编译工具和插件。 启用时，它要求 [liboce][] 可用，并通过 OCE_DIR标志传递已安装的 OCE 库的位置。 默认情况下，此选项处于启用状态。

或者，可以使用 KICAD_USE_OCC 代替 OCE。不应同时启用这两个选项。

# Wayland EGL 支持

KICAD_USE_EGL 选项将 OpenGL 后端从使用 X11 绑定切换到 Wayland EGL 绑定。 只有在运行 wxWidgets 3.1.5+ 和 wxGLCanvas 的 EGL 后端时，该选项才与 Linux 相关(这是默认选项，但可以在 wxWidgets 编译中禁用)。

默认情况下，设置 KICAD_USE_EGL 将使用静态链接到 KiCad 的 Glew 库的树内版本(使用在 EGL 画布上运行所需的附加标志进行编译)。 如果 Glew 的系统版本支持 EGL (必须使用 GLEW_EGL 标志进行编译)，则可以通过将 KICAD_USE_Bundled_Glew 设置为 OFF 来使用它。

# Windows HiDPI 支持

KICAD_WIN32_DPI_AWARE 选项使 KiCad 的 Windows 清单文件使用支持 DPI 的版本， 该版本告诉 Windows KiCad 希望每个监视器 V2 识别 DPI (需要 Windows 10 版本 1607 和更高版本)。

# 开发分析工具

KiCad 可以编译为支持多个功能，以帮助捕获和调试运行时内存问题

## Valgrind 支持

KICAD_USE_VALGRIND 选项用于在工具框架中启用 Valgrind 的堆栈注释功能。 这为 Valgrind 提供了跟踪工具框架中的内存分配和访问的能力，并减少了报告的误报数量。 默认情况下，此选项处于禁用状态。

## C++ 标准库调试

KiCad 提供了两个选项来启用 GCC C++ 标准库中包含的调试断言：KICAD_STDLIB_DEBUG 和 KICAD_STDLIB_LIGHT_DEBUG。 默认情况下，这两个选项都处于禁用状态，并且一次只应打开一个选项，且 KICAD_STDLIB_DEBUG 优先。

KICAD_STDLIB_LIGHT_DEBUG 选项通过将 `_GLIBCXX_ASSERTIONS` 传递到 CXXFLAGS 来启用轻量级标准库断言。 这允许对字符串、数组和向量进行边界检查，以及对智能指针进行空指针检查。

KICAD_STDLIB_DEBUG 选项通过将 `_GLIBCXX_DEBUG` 传递到 CXXFLAGS 来启用全套标准库断言。 这启用了对标准库的完全调试支持。

### Address Sanitizer 支持

KICAD_SANITIZE 选项启用地址清理程序支持，以跟踪内存分配和访问以确定问题。 默认情况下，此选项处于禁用状态。Address Saniizer 包含多个运行时选项，用于调整其行为， 在其 文档 中有更详细的描述。

并非所有编译系统都支持此选项，并且已知在使用 mingw 时会出现问题。

## 演示和示例

KiCad 源代码包括一些演示和示例来展示该程序。您可以使用 KICAD_INSTALL_DEMOS 选项选择是否安装它们。 您还可以使用 KICAD_DEMOS 变量选择它们的安装位置。在 Linux 上，演示程序安装在。 默认情况下为 $prefix/share/kicad/demos。

## 质量保证 (QA) 单元测试

KICAD_BUILD_QA_TESTS 选项允许编译用于质量保证的单元测试二进制文件，作为默认编译的一部分。 默认情况下，此选项处于启用状态。

如果禁用此选项，仍可以通过手动指定目标来编译 QA 二进制文件。 以 `make` 为例：

- 编译所有 QA 二进制文件: `make qa_all`
- 编译特定的测试: `make qa_pcbnew`
- 编译所有单元测试: `make qa_all_tests`
- 生成所有测试工具二进制文件: `make qa_all_tools`

有关测试 KiCad 的更多信息，请参阅 [本页](testing.md)。

## KiCad 编译版本

当 git 可用时，KiCad 版本字符串由 `git Describe–dirty` 的输出定义， 或者由 CMakeModules/KiCadVersion.cmake 中定义的版本字符串定义， 并在前者后面附加 KICAD_VERSION_EXTRA 的值。如果未定义 KICAD_VERSION_EXTRA 变量， 则不会将其附加到版本字符串。如果定义了 KICAD_VERSION_EXTRA 变量， 则会将其与前导 '-' 一起追加到完整版本字符串，如下所示：

```
(KICAD_VERSION[-KICAD_VERSION_EXTRA])
```

生成脚本自动从 [git][] 仓库中创建版本字符串信息。 有关资料如下：

```
(5.0.0-rc2-dev-100-g5a33f0960)
 |
 如果 git 可用，则输出 `git describe --dirty`。
```

# KiCad 配置目录

KiCad 默认配置目录为 `kicad`。在 Linux 上位于 `~/.config/kicad` ， 在 MSW 上位于 `C：\Documents and Settings\用户名\Application Data\kicad` ， 在 MacOS 上位于 `~/Library/Preferences/kicad` 。 如果安装包愿意，它可以指定一个替代配置名称，而不是 `kicad`。 这对于对配置参数进行版本化并且允许同时使用例如 `kicad5` 和 `kicad6` 而不丢失配置数据可能是有用的。

这是通过在编译时指定 KICAD_CONFIG_DIR 字符串来设置的。

# 获取 KiCad 源代码

有几种方法可以获得 KiCad 源代码。如果您想编译稳定版本，可以从 [GitLab][] 仓库下载源压缩包。使用 tar 或其他压缩程序解压系统上的源代码。如果您使用的是 tar，请使用以下命令：

```
tar -xaf kicad_src_archive.tar.xz
```

如果您直接参与 GitLab 上的 KiCad 项目，则可以使用以下命令在您的计算机上创建本地副本：

```
git clone https://gitlab.com/kicad/code/kicad.git
```

以下是源链接列表：

稳定的版本压缩包： https://kicad.org/download/source/

开发分支： https://gitlab.com/kicad/code/kicad/tree/master

GitHub 镜像: https://github.com/KiCad/kicad-source-mirror

# 已知问题

有些已知问题会影响所有平台。本节提供了在任何平台上编译 KiCad 时当前已知问题的列表。

## Boost C++ 库问题

从 GNU GCC 版本 5 开始，使用下载、修补和编译 Boost 1.54 的默认配置将导致 KiCad构 建失败。因此，必须用更新版本的 Boost 来编译 KiCad。如果您系统安装了 Boost 1.56 或更高版本，则您的工作非常简单。 如果您的系统未安装 Boost 1.56 或更高版本，则必须从源代码下载和 编译 Boost。如果您使用 MinGW 在 Windows 上编译 Boost，则必须应用 KiCad 源 patches 文件夹中的 Boost 补丁。

# LINUX

# 在 Linux 上编译 KiCad

要在 Linux 上执行完整编译，请运行以下命令：

```
cd <你的 kicad 源镜像文件夹>
mkdir -p build/release
mkdir build/debug                # 对于调试版本是可选的。
cd build/release
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo \
        ../../
make
sudo make install
```

如果 CMake 配置失败，请确定缺少的依赖项并将其安装在您的系统上。 默认情况下，CMake 会将 Linux 上的安装路径设置为 `/usr/local` 。 使用 `CMAKE_INSTALL_PREFIX` 选项指定不同的安装路径。

我们建议对个人版本使用 `RelWithDebInfo` 编译类型，因为这将包括调试符号， 以便在遇到崩溃时提供更有用的堆栈跟踪。

对于调试版本，请将 `RelWithDebInfo` 替换为 `Debug`。

# 小贴士和技巧

## Ninja

KiCad 使用 Ninja 编译系统代替 `make` 编译速度更快。要使用 Ninja ，可以在 CMake 命令行中指定 Ninja 输出：

```
cmake -G Ninja -DCMAKE_BUILD_TYPE=RelWithDebInfo ../../
ninja
sudo ninja install
```

MACOS

# 在 macOS 上编译 KiCad

从 V5 开始，macOS 的编译和打包可以使用 kicad-mac-builder， 来完成，它可以为 macOS 下载、修复、编译和打包。它用于创建正式版本和夜间版本，并将编译环境设置的复杂性降低为一两个命令。 kicad-mac-builder 的用法在其网站上有详细介绍。

如果您希望在不使用 kicad-mac-builder 的情况下进行编译，请使用以下内容及其源代码作为参考。在 macOS 上编译需要编译依赖库，这些依赖库需要打补丁才能正常工作。

在以下命令集中，将 macOS 版本号(即 10.11)替换为所需的最低版本。为您正在运行的同一版本编译可能是最简单的。

KiCad 目前不能与可由 MacPorts 或 Homebrew 等包管理器下载或安装的 wxWidget 的现成版本一起使用。 为了避免处理补丁，GitHub 上维护了 KiCad 的 wxWidget 分支。 所有需要的补丁和其他一些修复/改进都包含在 `kicad/macos-wx-3.0` 分支中。

要执行 wxWidgets 编译，请执行以下命令：

```
cd <您的 wxWidgets 编译文件夹>
git clone -b kicad/macos-wx-3.0
https://gitlab.com/kicad/code/wxWidgets.git
mkdir wx-build
cd wx-build
../wxWidgets/configure \
    --prefix=`pwd`/../wx-bin \
    --with-opengl \
    --enable-aui \
    --enable-html \
    --enable-stl \
    --enable-richtext \
    --with-libjpeg=builtin \
    --with-libpng=builtin \
    --with-regex=builtin \
    --with-libtiff=builtin \
    --with-zlib=builtin \
    --with-expat=builtin \
    --without-liblzma \
    --with-macosx-version-min=10.11 \
    --enable-universal-binary=i386,x86_64 \
    CC=clang \
    CXX=clang++
make
make install
```

如果一切正常，您可以在 `<您的 wxWidgets 编译文件夹>/wx-bin` 中找到 wxWidgets 二进制文件。 现在，使用以下命令编译一个不带 Python 脚本的基本 KiCad：

```
cd <你的 kicad 源文件镜像>
mkdir -p build/release
mkdir build/debug                # 对于调试版本是可选的。
cd build/release
cmake -DCMAKE_C_COMPILER=clang \
        -DCMAKE_CXX_COMPILER=clang++ \
        -DCMAKE_OSX_DEPLOYMENT_TARGET=10.11 \
        -DwxWidgets_CONFIG_EXECUTABLE=<your wxWidgets build folder>/wx-
bin/bin/wx-config \
        -DKICAD_SCRIPTING=OFF \
        -DKICAD_SCRIPTING_MODULES=OFF \
        -DKICAD_SCRIPTING_WXPYTHON=OFF \
        -DCMAKE_INSTALL_PREFIX=../bin \
        -DCMAKE_BUILD_TYPE=Release \
        ../../
make
make install
```

如果 CMake 配置失败，请确定缺少的依赖项并将其安装在系统上，或者禁用相应的 KiCad 功能。 如果一切正常，您将在 `build/bin` 文件夹中获得自包含的应用程序包。

使用 Python 脚本编译 KiCad 更加复杂，这里不再详细介绍。 您将不得不针对 KiCad 分支的 wxWidgets 源代码编译 wxPython 可能与 wxPython 包捆绑在一起的现有 wxWidget 将无法工作。 请参阅 wxPython 文档。或者 [macOS 打包编译脚本][] (`Compile_wx.sh`) 了解具体操作方法。 然后，使用 CMake 配置，如下所示将其指向您自己的 wxWidgets/wxPython：

```
cmake -DCMAKE_C_COMPILER=clang \
        -DCMAKE_CXX_COMPILER=clang++ \
        -DCMAKE_OSX_DEPLOYMENT_TARGET=10.9 \
        -DwxWidgets_CONFIG_EXECUTABLE=<your wxWidgets build folder>/wx-
bin/bin/wx-config \
        -DPYTHON_EXECUTABLE=<path-to-python-exe>/python \
        -DPYTHON_SITE_PACKAGE_PATH=<your wxWidgets build folder>/wx-
bin/lib/python2.7/site-packages \
        -DCMAKE_INSTALL_PREFIX=../bin \
        -DCMAKE_BUILD_TYPE=Release \
        ../../
```

# WINDOWS (MSYS2)

# 使用 MSYS2 编译

## 设置

MSYS2 项目为编译 KiCad 所需的所有依赖项提供了包。 要安装 MSYS2，请执行以下操作编译环境，下载并运行 msys2 主页提供的 **MSYS2 64 位安装程序**。 安装完成后，运行 MSYS2 安装路径中的 `msys2_shell.cmd` 文件，并运行命令 `pacman-Syu` ， 更新到最新的软件包版本。如果更新了 `msys2-runtime` 包，请关闭 shell 并运行 `msys2_shell.cmd` 。

## 编译

以下命令假定您是为 64 位 Windows 编译的，并且您的主目录中名为 `kicad-source` 的文件夹中已经有 KiCad 源代码。 如果您需要编译 32 位版本，请参见下面的更改。从 MSYS2 安装路径运行 `mingw64.exe` 。 在命令提示符下运行以下命令：

```
pacman -S base-devel \
            git \
            mingw-w64-x86_64-cmake \
            mingw-w64-x86_64-doxygen \
            mingw-w64-x86_64-gcc \
            mingw-w64-x86_64-python2 \
            mingw-w64-x86_64-pkg-config \
            mingw-w64-x86_64-swig \
            mingw-w64-x86_64-boost \
            mingw-w64-x86_64-cairo \
            mingw-w64-x86_64-glew \
            mingw-w64-x86_64-curl \
            mingw-w64-x86_64-wxPython \
            mingw-w64-x86_64-wxWidgets \
            mingw-w64-x86_64-toolchain \
            mingw-w64-x86_64-glm \
            mingw-w64-x86_64-oce \
            mingw-w64-x86_64-ngspice \
            mingw-w64-x86_64-zlib
cd kicad-source
mkdir -p build/release
mkdir build/debug               # 对于调试版本是可选的。
cd build/release
cmake -DCMAKE_BUILD_TYPE=Release \
        -G "MSYS Makefiles" \
        -DCMAKE_PREFIX_PATH=/mingw64 \
        -DCMAKE_INSTALL_PREFIX=/mingw64 \
        -DDEFAULT_INSTALL_PATH=/mingw64 \
        ../../
make -j N install   # 其中 N 是系统可以处理的并发线程数
```

对于 32 位版本，运行 `mingw32.exe` ，将包名中的 `x86_64` 改为 `i686`，并将 cmake 配置中的路径从 `/mingw64` 改为 `/mingw32` 。

对于调试版本，请在 `build/debug` 文件夹中使用 `-DCMAKE_BUILD_TYPE=Debug` 运行 cmake 命令。

# 带 Clion 的 MSYS2

与 MSYS2 结合使用的 KiCad 可以配置为与 Clion 一起使用，以提供良好的 IDE 体验。

## toolchain 设置

首先，您必须将 MSYS2 注册为 toolchain，即编译器。

打开设置窗口 `Files > preferences` 。

导航至 Build、Execution、Development，然后导航至 Toolchains 页面。

添加新的 toolchain，并按如下方式进行配置

- Name: `MSYS2-MinGW64`
- Environment Path: `<您的 msys2 安装文件夹>\mingw64\`
- CMake: `<您的 msys2 安装文件夹>\mingw64\bin\cmake.exe`

所有其他字段将自动填充。

## 工程设置

File > Open 并 select 包含 kicad 源的文件夹。 Clion 可能会尝试启动 CMake 生成，但会失败，这是可以接受的。

再次打开设置窗口。 导航到 Build、Execution、Development，然后导航到 CMake 页面。 这些设置将保存到工程中。

您希望这样创建调试配置

- Name: `Debug-MSYS2`
- Build-Type: `Debug`
- Toolchain: `MSYS2-MinGW64`
- CMake 选项:

```
-G "MinGW Makefiles"
-DCMAKE_PREFIX_PATH=/mingw64
-DCMAKE_INSTALL_PREFIX=/mingw64
-DDEFAULT_INSTALL_PATH=/mingw64
```

- 编译目录： `build/debug-msys2`

您现在可以触发 Clion 中的 "Reload CMake Cache" 选项来生成 cmake 工程，您应该删除 "junk" 编译文件夹(通常名为 cmake-build-debug-xxxx)， 它可能是在上面更改之前在源代码中创建的。我们更改了 build 文件夹，因为我们有一个用于 `/build` 的 gitignore

警告：收到有关 Boost 编译的警告消息是正常的。

# 已知的 MSYS2 编译问题

有一些特定于 MSYS2 的已知问题。本节提供了使用 MSYS2 编译 KiCad 时当前已知问题的列表。

## 使用 Boost 1.70 编译

使用 Boost 版本 1.70 编译 KiCad 时出现问题，原因是 CMake 在配置期间没有定义正确的链接库。可以使用 Boost 1.70，但需要在 CMake 配置过程中添加 `-DBoost_NO_BOOST_CMAKE=ON` ，以确保链接库正确生成。

## 从源代码编译 OCE

KiCad 默认需要 OCE，从 2018 年 3 月开始，`pacman` 安装的版本会导致 x86_64 系统内部版本错误。 为了解决这个问题，您可以在这些系统上从源代码编译 OCE。在 Windows 上编译 OCE 需要将源代码放在非常短的目录路径中， 否则会遇到 Windows 上最大路径长度导致的错误。在下面的示例中，`MINGW-Packages` 仓库克隆到 `/c/mwp` ， 相当于 Windows 路径术语中的 `C:\mwp` 。如果您不想将 `git clone` 命令放在 C 盘的根目录下， 您可能希望更改该命令的目的地，但如果您遇到有关丢失文件的奇怪编译错误，可能是因为您的路径太长。

```
git clone https://github.com/Alexpux/MINGW-packages /c/mwp
cd /c/mwp/mingw-w64-oce
makepkg-mingw -is
```

# WINDOWS (VISUAL STUDIO)

# 使用 Visual Studio 编译 (2019 年)

## 环境设置

### Visual Studio

您必须先安装 Visual Studio，并安装 **桌面开发和 C++** 功能集。 此外，您还需要确保安装了可选组件 C++CMake Tools for Windows。

### vcpkg (需要使用 kicad fork)

KiCad 维护一个 vcpkg 分支，为我们提供 wxPython 端口。 它还允许我们在 vcpkg 更新相当频繁时锁定依赖项。

**如果您是 vcpkg 新手** 您必须在您的系统上选择一个位置来放置它。 然后运行以下三个命令

```
git clone https://gitlab.com/kicad/packaging/vcpkg
.\vcpkg\bootstrap-vcpkg.bat
.\vcpkg\vcpkg integrate install
```

这将为您提供一个 vcpkg 安装，可以在接下来的步骤中使用

## KiCad 特定设置

即使在 64 位机器上，vcpkg 也默认为 x86-windows， 为便于使用，建议您设置一个名为 **VCPKG_DEFAULT_TRIPET**、 值为 **x64-WINDOWS** 的 **USER** 或 **SYSTEM** 环境变量

KiCad 目前仍支持 32 位版本，但将来可能不会支持，因此 64 位版本是首选。

## 1. 安装 vcpkg 包

vcpkg 需要以下软件包

```
.\vcpkg install boost
.\vcpkg install cairo
.\vcpkg install curl
.\vcpkg install glew
.\vcpkg install gettext
.\vcpkg install glm
.\vcpkg install icu
.\vcpkg install libxslt
.\vcpkg install ngspice
.\vcpkg install opencascade
.\vcpkg install opengl
.\vcpkg install openssl
.\vcpkg install python3
.\vcpkg install wxpython
.\vcpkg install wxwidgets
.\vcpkg install zlib
```

如果您没有设置 **VCPKG_DEFAULT_TRIPET** 环境变量，则必须在每个包名称的末尾添加：x64-windows，例如 `Boost：x64-windows` 。

如果在 wxpython 上 vcpkg 安装失败，您没有从 `kicad` 分支 https://gitlab.com/kicad/packaging/vcpkg 克隆并检出 vcpkg

## 2. CMakeSettings.json

编译根目录中包含一个 `CMakeSettings.json.sample` ，将该文件复制并重命名为 `CMakeSettings.json` 编辑 `CMakeSettings.json` 向上更新 VcPkgDir 环境变量以匹配 vcpkg 克隆的位置。

```
{ "VcPkgDir": "D:/vcpkg/" },
```

## 3. 在 Visual Studio 中 "打开文件夹"

启动 Visual Studio (仅在完成上述步骤之后)。

初始向导启动后，选择 **打开本地文件夹** 这是让 Visual Studio 直接处理 **CMake** 工程的正确方式。

# Visual Studio 扩展

## 尾随空格删除

强烈建议用户安装 尾随空格观察器 它不仅会在您键入时突出显示尾随空格，而且在保存文件时默认情况下会自动将其删除。

# 编译选项

这些是在配置过程中传递给 cmake 的选项

## 所有平台

| 选项 | 描述 | 默认 |
| --- | --- | --- |
| KICAD_SCRIPTING | 在 KiCad 二进制文件中编译 Python 脚本支持。 | ON |
| KICAD_SCRIPTING_MODULES | 编译 pcbnew Python 模块的本机部分：_pcbnew.{pyd，so}，以便操作系统命令行使用 Python。<br>如果使用 WXPYTHON，则 python 版本必须与用于编译 wxPytho n的版本相同 | ON |
| KICAD_SCRIPTING_PYTHON3 | 为 Python 3 而不是 2 编译。 | OFF |
| KICAD_SCRIPTING_WXPYTHON | 在 Python 和 py.shell 中编译用于编译 WX 接口的 wxPython 实现。 | ON |
| KICAD_SCRIPTING_WXPYTHON_PHOENIX | 使用新的 wxPython 绑定。 | OFF |
| KICAD_SCRIPTING_ACTION_MENU | 使用注册的 python 插件编译工具菜单：操作插件。 | ON |

| 选项 | 描述 | 默认 |
| --- | --- | --- |
| KICAD_USE_OCE | 编译与 OpenCascade Community Edition 相关的工具和插件。<br>需要支持导入/导出 STEP | ON |
| KICAD_USE_OCC | 编译与 OpenCascade 技术相关的工具和插件。<br>覆盖 KICAD_USE_OCE | OFF |
| KICAD_INSTALL_DEMOS | 安装 KiCad 演示和示例。 | ON |
| KICAD_BUILD_QA_TESTS | 编译软件质量保证单元测试。 | ON |
| KICAD_SPICE | 使用内部 Spice 仿真器编译 KiCad。 | ON |
| KICAD_BUILD_I18N | 编译翻译语言库 | OFF |
| KICAD_I18N_UNIX_STRICT_PATH | 将语言库安装到标准 UNIX 安装路径 ${CMAKE_INSTALL_PREFIX}/share/locale | OFF |
| BUILD_SMALL_DEBUG_FILES | 在调试版本中：创建较小的二进制文件。<br>在 Windows 上，链接选项 -g3 创建的二进制文件 **非常大**<br>(pcbnew 超过 1 GB，完整 kicad 工具箱超过 3 GB)<br>此选项使用链接选项 -g1 创建二进制文件，+<br>但是 debug 中的信息较少 (但是提供了文件名和行号) | OFF |

| 选项 | 描述 | 默认 |
| --- | --- | --- |
| MAINTAIN_PNGS | 允许从相应的 .svg 文件生成/重建菜单中使用的位图图标。<br>如果您是PNG维护者，并且在 bitmap_png/CMakeLists.txt 文件中提供了所需的工具，则设置为 true | OFF |

# 并非所有平台都支持

| 选项 | 描述 | 默认 |
| --- | --- | --- |
| KICAD_SANITIZE | 使用 sanitizer 选项编译KiCad。<br>警告：与 gold 链接器不兼容。 | OFF |
| KICAD_STDLIB_DEBUG | 在启用 libstdc++ 调试标志的情况下编译 KiCad。 | OFF |
| KICAD_STDLIB_LIGHT_DEBUG | 使用 libstdc++ 编译 KiCad，并启用 -Wp,-D_GLIBCXX_ASSERTIONS 标志。<br>不像 KICAD_STDLIB_DEBUG 那样具有侵入性 | OFF |
| KICAD_BUILD_PARALLEL_CL_MP | 使用 /MP 编译器选项并行编译(出于安全原因，默认设置为关闭)。 | OFF |
| KICAD_USE_VALGRIND | 在启用 valgrind 堆栈跟踪的情况下生成 KiCad。 | OFF |

# 注意

启用选项 `KICAD_SCRIPTING` 或 `KICAD_SCRIPTING` 或 `KICAD_SCRIPTING_MODULES` 时：

调用 cmake 时可以定义 `PYTHON_EXECUTABLE` （使用 `-DPYTHON_EXECUTABLE=<python 路径>/python.exe` 或 python2）用户未定义时，Windows 下默认为 python.exe，其他系统默认为 python2。 python 二进制文件应在 exec 路径中。

# 注意 1

KICAD_SCRIPTING 控制整个 Python 脚本系统。 如果该选项处于关闭状态，则不允许编写其他脚本

因此，如果 `KICAD_SCRIPTING` 为 OFF，则这些其他选项将被强制关闭：
`KICAD_SCRIPTING_MODULES, KICAD_SCRIPTING_ACTION_MENU,KICAD_SCRIPTING_PYTHON3 KICAD_`

# 注意 2

`KICAD_SCRIPTING_WXPYTHON_PHOENIX` 需要启用 `KICAD_SCRIPTING_WXPYTHON` 标志。 因此 wxWidgets 库的版本设置正确

# 注意 3

这些符号始终是已定义的，不是 cmake 调用的选项：

**COMPILING_DLL**

这是一个指向 import_export.h 的信号，当它出现时， 会切换对该文件中的 #defines 的解释。 它的目的不应该超出这一点。

**USE_KIWAY_DLLS**

作为用户配置变量来自 CMake，可在 CMake 用户界面中设置。 它决定 KiCad 是否使用 *.kiface 程序模块编译。

**BUILD_KIWAY_DLL**

来自 CMake，但在第二层，而不是顶级。第二层指的是 pcbnew/CMakeLists.txt，而不是 /CMakeLists.txt。 它不是用户配置变量。相反，第二层 CMakeLists.txt 文件查看顶级 `USE_KIWAY_DLLS` ， 并决定如何编译第二层控制下的目标文件。如果它决定与 `USE_KIWAY_DLLS` 步调一致， 则此本地 CMakeLists.txt 文件可能会将编译器命令行上定义的 `BUILD_KIWAY_DLL` (单数) 传递给其控制下的相关编译步骤集。

# 注意 4

在 Linux 系统上使用 `KICAD_BUILD_I18N` 编译时，gettext 需要规则文件 `shared-mime-info.its` 和 `metainfo.its` / `appdata.its` 来翻译 Linux 元数据文件。

# 规则和准则

- 代码编程规范
  *任何贡献到 KiCad 代码仓库的代码都必须遵守的代码编程规范。*
- 图标设计指南
  *应用于图标更改或添加的图标样式指南*
- 提交规范
  *关于如何格式化 git 提交信息的指引*
- UI 规范
  *有关如何实现用户界面元素的规则*

# 代码编程规范

## 1. 简介

本文档旨在为 KiCad 开发人员提供有关如何在 KiCad 中设置源代码样式和格式的参考指南。 它不是一个全面的编程指南，因为它没有讨论很多事情，如软件工程策略、源目录、现有类或如何国际化文本。 我们的目标是使所有的 KiCad 源代码都符合本指南。

## 1.1 为什么代码规范很重要

您可能会想，使用本文档中定义的样式并不能使您成为一名优秀的程序员，这样您就是正确的。 任何给定的编码样式都不能取代经验。 然而，任何有经验的程序员都会知道，比查看不是您喜欢的编码样式的代码更糟糕的是， 查看不是您首选的编码样式的二十种不同的编码样式。 一致性使 a) 问题更容易发现，b) 长时间查看代码更容易容忍。

## 1.2 强制执行

如果你不遵守 KiCad 代码规范，我们并不会闯入你家并用你的键盘暴揍你（虽然有一部分人认为应该这么做）。 但是，确实存在一些你应该遵守本文档的充分的理由。 假设你正在贡献一个补丁，如果你的补丁是按照规范格式的，那么它将会被主要开发者 (primary developers) 更加认真地对待。 忙碌的开发者没有时间帮你重新格式化你的代码。 如果你有意成为一个对开发分支有提交权限的正式 KiCad 开发者， 那么你就不应该经常因为不遵守代码编程规范而受到首席开发者 (lead developers) 的批评。 遵守代码编程规范是一个好的编程礼仪，它体现出对已有的开发者作出的贡献的尊重。 其他 KiCad 开发者也会感谢你对此作出的努力。

**警告**

**未经项目负责人同意，请勿修改本文档。 对此文档的所有更改都需要审批。**

## 1.3 工具

这里有一些能够帮助你快速格式化代码的工具。

clang-format 是个不仅能帮你 在你自己的编辑器里自动代码编程格式化的工具，它还能在你 git 提交代码时检查代码格式（使用 "Git Hook"）。 必须安装这个程序才能使用 "Git Hooks" 。

代码规范配置文件是 `_clang-format` ， 当设置了 `--style=file` 这个参数时， 这个文件应该会自动被 `clang-format` 自动读取。

你可以这样打开 Git 钩子(hooks)（这个操作每个 git 仓库只需要做一次）：

```
git config core.hooksPath .githooks
```

设置 `git clang-format` 工具使用我们提供的 `_clang-format` 文件:

git config clangFormat.style file

然后通过设置 `kicad.check-format` git 配置项为 "true" 来为 KiCad 仓库打开代码格式检查：

```
git config kicad.check-format true
```

如果没有配置这一配置项， 代码格式检查就不会在提交（commit）的时候运行， 但是你仍然可以手工检查在暂存去的文件（参见后文）。

如果钩子 （hook）是打开的， 当你提交改动时，将提示你所改动的代码是否违反了代码规范。 你可以相应地通过手工，或者使用下面的工具，来消除错误。

如果你确定你的代码格式是正确的，但还是出现格式错误警告，你可以这样来强制提交：

```
git commit --no-verify
```

# 1.3.1 纠正格式错误

有一个 git 别名(alias) 文件提供了显示和纠正格式错误的命令。 通过下面的命令可以把它加入到你的仓库配置中：

```
git config --add include.path $(pwd)/helpers/git/format_alias
```

之后，你就可以使用下面的别名：

- `git check-format` ：显示存在的格式警告 （但不做更改）
- `git fix-format` ：纠正格式 （之后你需要做一次 `git add` ）

这些别名使用脚本 `tools/check-coding.sh` ，这个脚本确保只检查那些应该遵循规范的文件。这个脚本还有一个作用：

- 修正 （或者查看） 上一次提交修改的文件中的格式错误。这在交互式变基 (interactive-rebasing) 中有用：
- `check_coding.sh --amend [--diff]`

# 2. 命名规范

在探讨神秘的缩进和格式问题之前，先要强调命名规范。 这一节，我们将定义命名的风格， 而不会定义代码应该用什么样的名字。请在参考文件一节获取一些优秀的编码指引。 当定义多单词名字时，使用下面的约定来改善可读性：

- 对于全部大写或者全部小写的变量名，使用下划线分隔单词使之易于识读。
- 对于大小写混合的变量名，使用驼峰命名法 (camel case) 。

不要混合驼峰命名法和下划线。

**例如**

```
CamelCaseName           // 如果使用驼峰式就不要用下划线
all_lower_case_name
ALL_UPPER_CASE_NAME
```

## 2.1 类， 类型定义， 命名空间和宏名

类 (class), 类型定义 (typedef), 枚举 (enum), 命名空间 (name space) 和宏 (macro) 应该由大写字母组成。

**例如**

```
class SIMPLE
#define LONG_MACRO_WITH_UNDERSCORES
typedef boost::ptr_vector<PIN>      PIN_LIST;
enum KICAD_T {...};
```

# 2.2 局部变量， 私有变量和自动变量

自动变量，静态局部变量和私有变量的变量名第一个字符必须用小写字母。 以此来表示这些变量对它所被定义的函数，文件或者类的外部并不可见。 局部可见性由于开始的小写字母来表示，因为小写字母通常被认为是没有大写字母那么高调。

**例如**

```
int i;
double aPrivateVariable;
static char* static_variable = NULL;
```

类和结构体的私有成员必须以 `m_` 开头，并且 `m_` 后面的第一个字母也必须是小写。

```
private:
    int m_privateData;
```

# 2.3 公开变量和全局变量

公开变量和全局变量的变量名第一个字母必须是大写字母。 以此来表示这个变量对它所被定义的类或者文件的外部可见。 （一个特例是，有时也会用以 `g_` 开头来表示全局变量）

**例如**

```
char* GlobalVariable;
```

通常情况下， 类，不应该包含公开成员变量，而应该使用 getter/setter 方法来访问私有变量。 一个特例是，仅仅用来表示数据结构，和只包含少量的帮手方法的类。 比如， `SEG` 类经常被当作数据结构来表示两点之间的直线段。 这个类有公开成员变量 `A` 和 `B` 用于保存它的端点。 这是可以接受的，因为 `SEG` 类符合只是用来表示数据结构的定义， 因为修改 `A` 或 `B` 的值时不需要其他连带动作。

# 2.4 局部函数，私有函数和静态函数

局部，私有和静态函数的第一个字母必须是小写字母。 以此来表示这个函数对它所被定义的类或者文件的外部不可见。

**例如**

```
bool isModified();
static int buildList( int* list );
```

# 2.5 函数参数

函数参数名称用 'a' 开头。 'a' 代表参数 (argument), 并且能够帮助生成智能和简约的 Doxygen 注释。

**例如**

```
/**
 * 将 aFoo 复制到此实例中。
 */
void SetFoo( int aFoo );
```

我们注意到，读者在读到这里的时候，可以默读成 “a Foo”。

# 2.6 指针

指针变量名并不需要内嵌 'p' 字母来表示。 指针代表一个变量从属于一个类型，而不是目标。

**例如**

```
MODULE*   module;
```

这个变量的意义是代表一个 MODULE， `p_module` 这样的写法只会增加辨识的难度。

## 2.7 访问成员变量和成员函数

我们不在类的内部使用 `this->` 来访问成员变量或者成员函数。 我们让 C++ 帮我们做到这一点。

## 2.8 'auto' 的使用

我们不使用 `auto` 来减少重复。 但我们可以用它来增加可读性。 也就是说仅仅在 std::lib 变得过度啰嗦，或者，不使用 `auto` 就会导致不可避免的折行情况下使用 `auto`。

# 3. 注释

在 KiCad 中，注释被分为两类：内联代码注释和 Doxygen 注释。 不跟随在语句之后的内联注释必须有和代码一致的缩进，除此之外没有其他格式要求。 跟随在语句之后的内联注释除非绝对必要，否则不应该超过 99 列。 避免在可视列宽为 100 的编辑器上发生自动换行。 内联注释可以既可以使用 C 也可以使用 C 的注释风格， 但是如果是单行或少量几行的注释，建议使用 C 风格。

避免出现显而易见的描述注释。 添加一个注释来说明这是一个 dtor, ctor, function, iterator 之类的，只是在注释中添加了无用的废话。只要有点经验的开发者就能看出来。 避免添加注释来描述代码做了什么。 代码做了什么应该很明显能被看出来， 不然，就说明代码需要被重写到能明显能看出来。

## 3.1 注释上面的空行

如果注释是一行的开始，那么这个注释上面应该有一行或多行空行。建议一行。

## 3.2 Doxygen

Doxygen 是本项目使用的一个 C++ 源代码文档工具. 安装 Doxygen 之后编译名叫 **doxygen-docs** 的目标， 就会从源代码生成描述性的 *.html 文件。

```
cd <kicad_build_base>
make doxygen-docs
```

生成的 *.html 文件将会放到下面的目录中 <kicad_project_base>/Documentation/doxygen/html/.

Doxygen 注释被用于从源代码编译出开发者文档。 这些注释通常只会放在头文件 (.h) 里而不放在代码文件 (.cpp) 里。 这样就避免需要同步两处注释的麻烦。 如果类，函数或者枚举等仅仅定义在一个代码文件中而不出现在头文件里， 这种情况下 Doxygen 注释就应该出现在代码文件里。 再说一遍，Doxygen 注释应该避免同时出现在头文件和代码文件中。

KiCad 使用 JAVADOC 注释风格，定义在 Doxygen 手册 的 doccode 一节。 别忘了使用特殊 Doxygen 标签: bug, todo, deprecated, 等。 这样其他开发者就能快速地找到你代码里的有用信息。 一个很好的习惯是，在提交你的补丁前，应当编译 doxygen-docs 目标生成 Doxygen *.html 文件， 并在浏览器里检查你的 Doxygen 注释的质量。

不要定义类和函数名称，参数和返回数据类型。这些多余的信息其他开发者可以轻易地看出来， 并在生成的文档中重复出现。

尽量使用 Doxygen's markdown 语法而不是 HTML 标签。 Markdown 注释比 HTML 更加易读。

当在 Doxygen 注释中引用其他代码的时候，使用 Doxygen 链接。 这可以让其他开发者更容易找到信息。

使用 Doxygen 分组命令 将相关的注释部分组合成一个组。 这样这些信息就不会散落在不同的文档页中，查找相关的信息会变得更方便。

### 3.2.1 函数注释

函数注释应该在头文件里，除非这个函数是这个源码文件私有的（即静态函数）。 格式化的函数注释有两个目的：在源代码文件里描述函数的声明和使 Doxygen 的文档输出有一致的开始句子。 格式要求是， 单一行描述性的句子，接一个空行，再接一个可选的详细描述。 下面是这个格式的例子。

**例如**

```
/**
 *
 *
 * 格式化文本并将文本写入输出流。
 *
 * @param aMestLevel 是输出前面的空格倍数。
 * @param aFmt 是 printf() 样式的格式字符串。
 * @param ... 是将在格式字符串控制下混合到
 *  输出中的参数的变量列表。
 * @return 输出的字符数。
 * @throw IO_ERROR, 如果输出有问题。
 */
int PRINTF_FUNC Print( int aNestLevel, const char* aFmt, ... );
```

单行的描述处于注释的第二行。 如果 @return 关键字出现的话， 返回值应该说明在一个连字符 (-) 之后。 @param 关键字之后跟函数的参数， 再之后的的文字应该和前面的名字组成一个正确的英语语句，包括标点。

### 3.2.2 类的注释

类的注释描述类的目的和用法来完成一个类的声明。 它的格式类似于函数格式。 Doxygen 可以使用 html \<p\> (段落标记) 让输出中新起一个段落。 所以，如果注释的文本太长的话，可以根据需要把它们分成多段。

**例如**

```
/**
 * An interface (abstract) class used to output UTF8 text in a
 * convenient way.
 *
 * The primary interface is "printf() like" but with support for
 * indentation control. The destination of the 8 bit wide text is
 * up to the implementer.
 * <p>
 * The implementer only has to implement the write() function, but
 * can also optionally re-implement GetQuoteChar().
 * <p>
 * If you want to output a wxString, then use CONV_TO_UTF8() on it
 * before passing it as an argument to Print().
 * <p>
 * Since this is an abstract interface, only classes derived from
 * this one may actually be used.
 */
class OUTPUTFORMATTER
{
```

# 4. 格式化

这一节定义了 KiCad 源代码的格式风格。

## 4.1 缩进

KiCad 源代码的一级缩进使用 4 个空格，请不要使用制表符 (tab)。

### 4.1.1 define 指令

`#define` 语句之后只应该有一个空格.

### 4.1.2 列对齐

请尽可能将个相似的行进行列对齐，例如，#define 语句，可以看成是由四列组成的，`#define` ，符号, 值和注释。请看下面的例子中，这四列是如何对齐的。

**例如** ~~~~{.cpp} #define LN_RED 12 // my favorite #define LN_GREEN 13 // eco friendly ~~~~

还有一个常见的案例是自动变量的申明。尽量将他们按照类型和变量名进行列对齐。

## 4.2 空行

### 4.2.1 函数申明

类文件中的函数声明，如果有 Javadoc 注释，那么函数申明之前应该有一个空行。

### 4.2.2 函数定义

*.cpp 文件中的函数定义通常不需要有注释，因为注释都在头文件里。 在 *.cpp 文件中的函数定义，在函数定义前最好先空 2 个空行。

### 4.2.3 控制语句

控制语句的开语句前，关闭语句或者右花括号后都应该有一个空行， 这样就能很容易地识别出控制块的开头和结尾。 这里所说的控制块包括 `if`, `for`, `while`, `do` 和 `switch` 。

## 4.3 行宽

最大的列宽是 99 列。特例是引文的长字符串，比如一些国际化文本需要满足下面描述的 MSVC++ 的要求。

## 4.4 字符串

KiCad 项目组不再支持用微软 Visual C++ 的编译。 当你需要将长字符串分割成多个段字符串时，请使用 C99 标准的方法来改善可读性。 下面描述的以前接受的分割国际化文本的方法现在已经不再接受了。

**例如**

```
    // This works with C99 compliant compilers is the **only** accepted
method:
    // 这种方式可以运行在符合 C99 标准的编译器上， 这是唯一接受的方法：
    wxChar* foo = _( “this is a long string broken ”
                     “into pieces for readability.” );

    // This works with MSVC, breaks POEdit, and is **not** acceptable:
    // 这种方式可以运行在 MSVC 上, POEdit 不支持, 而且现在已经不再接受了：
    wxChar* foo = _( “this is a long string broken ”
                    L“into pieces for readability” );

    // This works with MSVC, is ugly, and is **not** accepted:
    // 这种方式可以运行在 MSVC 上, 很丑, 而且现在已经不再接受了：
    wxChar* foo = _( “this is a long string \
    broken into pieces for readability” );
```

另外一种可接受的方案是将字符文本放在一行，即使它超过了 99 列的行宽限制。 但是， 更建议尽量将字符串分割不超过 99 列以防止自动换行。

# 4.5 行尾空白字符

很多编程编辑器可以很方便的帮你缩进代码。但是其中一些在处理得不太好， 会在行尾留下空白字符。幸运的是，大多数编辑器都包含删除尾部空白字符的宏， 或者至少有一个设置使得尾部的空行可以被看见以便可以手工删除它。 尾部空白字符会导致一些文本分析工具失效，还会导致版本管理系统中产生一些不必要的差异 (diffs)。 所以，请删除行尾空白字符。

# 4.6 一行多语句

我们建议一条语句占一行。对于没有关键字的语句尤其要独占一行。

```
    x=1; y=2; z=3; // 坏的，应该在独占一行。
```

# 4.7 大括号

大括号应该在关键字后面一行，而且和关键字缩进等级一致。 如果只有一条语句， 就不需要使用大括号。在 if..else if..else 这种语句中， 将他们缩进到同一个等级。

```
void function()
{
    if( foo )
    {
        statement1;
        statement2;
    }
    else if( bar )
    {
        statement3;
        statement4;
    }
    else
        statement5;
}
```

## 4.8 括号

括号应该立即跟在函数名和关键字后面。函数中开括号后面，闭括号前， 逗号和下一个参数之间应该留有一个空格。如果函数没有参数，那就不需要留空格。

```
void Function( int aArg1, int aArg2 )
{
    while( busy )
    {
        if( a || b || c )
            doSomething();
        else
            doSomethingElse();
    }
}
```

## 4.9 Switch 格式

case 语句应该和 switch 在同一个缩进级。

```
switch( foo )
{
case 1:
    doOne();
    break;
case 2:
    doTwo();
    // 失败了。
default:
    doDefault();
}
```

如果能提高可读性，最好将所有的 case 放在一行里。在查值或者映射函数中经常用这种方式。 在这种情况下，如果你使用 clang-format ，就需要拒绝它的的建议，使用手动对齐来增加可读性：

```
switch( m_orientation )
{
case PIN_RIGHT: m_orientation = PIN_UP;    break;
case PIN_UP:    m_orientation = PIN_LEFT;  break;
case PIN_LEFT:  m_orientation = PIN_DOWN;  break;
case PIN_DOWN:  m_orientation = PIN_RIGHT; break;
}
```

# 4.10 Lamdas

大括号和语句体应该像缩进一个方法那样缩进，大括号与上面的语句对齐。

```
auto belowCondition = []( const SELECTION& aSel )
                      {
                          return g_CurrentSheet->Last() !=
g_RootSheet;
                      };
```

or:

```
auto belowCondition =
    []( const SELECTION& aSel )
    {
        return g_CurrentSheet->Last() != g_RootSheet;
    };
```

## 4.11 类定义布局

成员变量应该在类定义的最底端。类成员的可见性排序应该按照公开 (public)， 保护 (protected) 和私有 (private) 的顺序进行。不要重新定义同一个可见性。 下面是一个类定义的例子：

```
class FOO
{
public:
    FOO();
    void FooPublicMethod();

protected:
    void fooProtectedMethod();

private:
    void fooPrivateMethod();

    // Private not redefined here unless no private methods.
    // 这里不需要重新定义私有(private) 除非上面没有私有方法
    int m_privateMemberVariable;
};
```

## 4.12 Getters 和 Setters

为了提高可读性，没有其他附带作用的读写私有成员的类方法可以触犯一些通常的格式规则。 这种方法的所有成分都在一行里，方法之间没有空行：

```
public:
    wxString GetFoo() const { return m_foo; }
    void SetFoo( const wxString& aFoo ) { m_foo = aFoo; }
```

# 5. 许可声明

可以将 copyright.h 中的内容拷贝到你的新代码文件中，修改 \<author\> 字段。 KiCad 依靠版权法版权保护能力，也就是说源代码文件必须有版权，不能释放到公共域。 每一个源代码文件都有一到多个属主 (owner)。

# 6. 调试输出 (debugging output)

调试输出是验证代码的一个常用方法。但是他不应该在调试编译 (debug build) 中一直打开。 不然会让其他开发者很难看到他们自己调试输出，还会严重影响调试编译的性能。 当你需要使用调试输出时，使用 wxLogDebug， 不要使用 `printf` 或者 C++ 输出流。 如果你不小心把调试输出遗留在代码中， 他会在发布编译 (release builds) 中被扩展成空。在推送到 KiCad 仓库前， 所有的调试输出都应该被删除，不要简单地将调试输出注释起来。这样会使代码库积累下繁琐的内容。 如果你需要为以后的测试留下调试输出，那就应该用追踪输出 (tracing output), 参见 6.1 小节。

## 6.1 用追踪输出代替调试输出

有时候需要用调试数据来确保代码是否按照期望的那样工作了。这种情况下就需要用 wxLogTrace 。这中方法可以允许调试输出被 `WXTRACE` 环境变量控制。当使用 wxLogTrace 时， 追踪环境变量字符串应当添加到 `trace_helper.{h/cpp}` 源代码文件中或者是本地使用 Doxygen 注释 `\ingroup trace_env_vars` 。

# 7. 头文件

项目的 *.h 源文件应该：

- 包括一个许可申明
- 包括一个嵌套包含 (nested include) `#ifndef`
- 完全独立，不依赖于其他没有包含在里面的头文件。

许可申明已经在之前详述了.

## 7.1 嵌套包含 (nested include) #ifndef

每一个头文件都应该包含一个 `#ifndef` 。 当这个头文件被多次输送给编译器时，它通常用来防止编译器报错。 在许可申明之后，文件的头部，应该有这样几行 (特定文件名标记不一定是 `RICHIO_H_` ):

```
#ifndef RICHIO_H_
#define RICHIO_H_
```

在头文件的最后，用这样的一行：

```
#endif // RICHIO_H_
```

`#ifndef` 包括了从许可申明之后开始直到文件的最后的内容。重要的是， 它应该也要包括 `#include` 语句，这样如果 `#ifndef` 是假时，编译器可以跳过这些文件， 就能节约很多编译时间。

# 7.2 没有未满足的依赖项的头文件

任何一个头文件都应该包含它所依赖的所有头文件。 （注意：KiCad 现在还没有完全做到这一点，但是这是项目的目标）

对项目中的任何一个头文件进行编译，如果正确地对编译器设置了包含文件路径， 那么这个头文件应该能够没有错误地正常编译。

**例如**

```
$ cd /svn/kicad/testing.checkout/include
$ g++ wx-config --cxxflags -I . xnode.h -o /tmp/junk
```

这样的头文件结构使得在客户 *.cpp 文件包含一个项目头文件时， 不需要在这个头文件之前包含其他的项目头文件。 （客户 *.cpp 文件是指准备使用，而不是实现这个头文件暴露的公开 API 的文件）。

不应该要求客户代码来粘接头文件所想要暴露的东西。 这个头文件应该被看作是使用这个头文件所暴露的 API 的 **凭证** 。

这里不是要指出暴露多少，而是说头文件想要暴露的东西， 应该只需要包含着一个头文件就能完全使用，而不需要再包含其他的头文件。

对于类的头文件和一个对它的实现 *.cpp 文件的情况，最好按照现在的做法， 隐藏尽可能多的私有实现， 不需要暴露为公开 API 的头文件，应该在 *.cpp 文件中包含。 总而言之，这一节主要要强调的是客户代码， 只需要包含一个头文件就可以使用这个头文件所暴露的所有 公共 API 了。

# 8. 当有疑问的时候...

当编辑一个已经存在的代码文件时，如果有多个可以选择的代码格式选项或者没有已经定义好的格式， 那么应该遵循这个文件中已经存在的格式。

# 9. 在看到这篇文档前我已经写了 n 行代码了

没关系。 我们都会犯错。幸运的是，KiCad 提供了一个代码美化工具 uncrustify 的配置文件。 uncrustify 不会纠正你的几个具体问题，但是它在美化代码方面做得很好。 但是 uncrustify 在有些地方的缩进选择并不太理想， 并且在 wxT(“”) 和 _(“”) 字符串申明宏作为其他函数的参数时比较纠结。 所以在 uncrustify 代码文件之后， 请检查是否有错误提示，并手工纠正错误。 可以在这里找到关于 uncrustify [website][uncrustify] 的更多信息。

[uncrustify]: http://uncrustify.sourceforge.net/

# 10. 给我个例子

用一个例子就能更加一阵见血地说明问题。 下面的 richio.h 代码文件就是直接从 KiCad 的源代码里拷贝出来的。

```
    /*
     * This program source code file is part of KICAD, a free EDA CAD
application.
     *
     * Copyright (C) 2007-2010 SoftPLC Corporation, Dick Hollenbeck
<dick@softplc.com>
     * Copyright (C) 2007 KiCad Developers, see change_log.txt for
contributors.
     *
     * This program is free software; you can redistribute it and/or
     * modify it under the terms of the GNU General Public License
     * as published by the Free Software Foundation; either version 2
     * of the License, or (at your option) any later version.
     *
     * This program is distributed in the hope that it will be useful,
     * but WITHOUT ANY WARRANTY; without even the implied warranty of
     * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
     * GNU General Public License for more details.
     *
     * You should have received a copy of the GNU General Public License
     * along with this program; if not, you may find one here:
     * http://www.gnu.org/licenses/old-licenses/gpl-2.0.html
     * or you may search the http://www.gnu.org website for the version 2
license,
     * or you may write to the Free Software Foundation, Inc.,
     * 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA
     */

    #ifndef RICHIO_H_
    #define RICHIO_H_


    // This file defines 3 classes useful for working with DSN text files
and is named
    // "richio" after its author, Richard Hollenbeck, aka Dick
Hollenbeck.


    #include <string>
    #include <vector>

    // I really did not want to be dependent on wxWidgets in richio
    // but the errorText needs to be wide char so wxString rules.
    #include <wx/wx.h>
    #include <cstdio>       // FILE



    /**
     * A class used to hold an error message and may be used to throw
exceptions
     * containing meaningful error messages.
```

```
     */
    struct IOError
    {
        wxString    errorText;

        IOError( const wxChar* aMsg ) :
            errorText( aMsg )
        {
        }

        IOError( const wxString& aMsg ) :
            errorText( aMsg )
        {
        }
    };


    /**
     * Read single lines of text into a buffer and increments a line
number counter.
     */
    class LINE_READER
    {
    protected:

        FILE*               fp;
        int                 lineNum;
        unsigned            maxLineLength;
        unsigned            length;
        char*               line;
        unsigned            capacity;

    public:

        /**
         * @param aFile is an open file in "ascii" mode, not binary mode.
         * @param aMaxLineLength is the number of bytes to use in the
line buffer.
         */
        LINE_READER( FILE* aFile, unsigned aMaxLineLength );

        ~LINE_READER()
        {
            delete[] line;
        }

        /*
        int  CharAt( int aNdx )
        {
            if( (unsigned) aNdx < capacity )
                return (char) (unsigned char) line[aNdx];
            return -1;
        }
        */
```

```
        /**
         * Read a line of text into the buffer and increments the line
number
         * counter.
         *
         * @return is the number of bytes read, 0 at end of file.
         * @throw IO_ERROR when a line is too long.
         */
        int ReadLine();

        operator char* ()
        {
            return line;
        }

        int LineNumber()
        {
            return lineNum;
        }

        unsigned Length()
        {
            return length;
        }
    };



    /**
     * An interface (abstract class) used to output ASCII text in a
convenient way.
     *
     * The primary interface is printf() like with support for
indentation control.
     * The destination of the 8 bit wide text is up to the implementer.
If you want
     * to output a wxString, then use CONV_TO_UTF8() on it before passing
it as an
     * argument to Print().
     * <p>
     * Since this is an abstract interface, only classes derived from
this one
     * will be the implementations.
     * </p>
     */
    class OUTPUTFORMATTER
    {

    #if defined(__GNUG__)   // The GNU C++ compiler defines this

    // When used on a C++ function, we must account for the "this"
pointer,
    // so increase the STRING-INDEX and FIRST-TO_CHECK by one.
```

```
    // See
http://docs.freebsd.org/info/gcc/gcc.info.Function_Attributes.html
    // Then to get format checking during the compile, compile with -Wall
or -Wformat
    #define PRINTF_FUNC       __attribute__ ((format (printf, 3, 4)))

    #else
    #define PRINTF_FUNC       // nothing
    #endif

    public:

        /**
         * Format and write text to the output stream.
         *
         * @param nestLevel is the multiple of spaces to preceed the
output with.
         * @param fmt is a printf() style format string.
         * @param ... is a variable list of parameters that will get
blended into
         *  the output under control of the format string.
         * @return the number of characters output.
         * @throw IO_ERROR if there is a problem outputting, such as a
full disk.
         */
        virtual int PRINTF_FUNC Print( int nestLevel, const char* fmt,
... ) = 0;

        /**
         * Return the quoting character required for aWrapee.
         *
         * Return the quote character as a single character string for a
given
         * input wrapee string.  If the wrappee does not need to be
quoted,
         * the return value is "" (the null string), such as when there
are no
         * delimiters in the input wrapee string.  If you want the quote
character
         * to be assuredly not "", then pass in "(" as the wrappee.
         * <p>
         * Implementations are free to override the default behavior,
which is to
         * call the static function of the same name.
         * </p>
         *
         * @param aWrapee is a string that might need wrapping on each
end.
         * @return the quote character as a single character string, or
""
         *   if the wrapee does not need to be wrapped.
         */
        virtual const char* GetQuoteChar( const char* aWrapee ) = 0;
```

```
        virtual ~OUTPUTFORMATTER() {}

        /**
         * Get the quote character according to the Specctra DSN
specification.
         *
         * @param aWrapee is a string that might need wrapping on each
end.
         * @param aQuoteChar is a single character C string which
provides the current
         *          quote character, should it be needed by the wrapee.
         *
         * @return the quote_character as a single character string, or
""
         *   if the wrapee does not need to be wrapped.
         */
        static const char* GetQuoteChar( const char* aWrapee, const char*
aQuoteChar );
    };


    /**
     * Implement an OUTPUTFORMATTER to a memory buffer.
     */
    class STRINGFORMATTER : public OUTPUTFORMATTER
    {
        std::vector<char>       buffer;
        std::string             mystring;

        int sprint( const char* fmt, ... );
        int vprint( const char* fmt,  va_list ap );

    public:

        /**
         * Reserve space in the buffer
         */
        STRINGFORMATTER( int aReserve = 300 ) :
            buffer( aReserve, '\0' )
        {
        }


        /**
         * Clears the buffer and empties the internal string.
         */
        void Clear()
        {
            mystring.clear();
        }

        /**
         * Remove whitespace, '(', and ')' from the internal string.
         */
```

```
        void StripUseless();


        std::string GetString()
        {
            return mystring;
        }


        //-----<OUTPUTFORMATTER>-------------------------------------------
-------
        int PRINTF_FUNC Print( int nestLevel, const char* fmt, ... );
        const char* GetQuoteChar( const char* wrapee );
        //-----</OUTPUTFORMATTER>-------------------------------------------
-------
    };


    #endif // RICHIO_H_
```

# 11. 其他资源

互联网上有很多关于 C++ 注意事项和编程规范的优秀资源。下面是其中一些。 这些资源的编程规范虽然不一定符合 KiCad 的编程规范，但是包含有很多其它有用的信息。 而且，他们中的大部分还挺幽默的，读起来还很有趣。说不定，你就学到一些新招了。

- C++ 编码标准
- Linux 内核编码风格
- C++ 运算符重载指南
- 维基百科的编程风格页面

# 图标设计指南

## 1. 外观和感觉

KiCad 图标设计指南旨在为 KiCad 中的图标创建一致的视觉样式。 视觉风格的目的是在每个图标都足够独特，便于定位，但又不够奇特， 以至于所有图标之间没有一致的主题， 看起来每个图标都是由不同的人设计的之间取得平衡。 这种视觉风格的一般原则是：

1. **平面设计**：图标不能使用渐变、阴影等 3D 效果。
2. **标准调色板**：图标应使用本文档中列出的受限调色板。 对此调色板的添加或更改必须与首席开发团队 (lead development team) 进行讨论， 因为为了保持一致性，可能需要更改许多图标。
3. **在形状足够的地方要慎用颜色**：形状相似的图标应该用颜色来区分。 仅能通过形状传达意义的图标 (不要与其他图标混淆) 应该是单色的， 只使用原色 (在灯光主题中为深灰色)
4. **像素对齐**：图标设计为向量，但用作位图。 矢量图形必须设计为以目标图标大小正确对齐像素， 以防止模糊。
5. **一致性**：描述相同概念的图标在使用颜色和形状来描述该概念时应该保持一致。 例如，所有包含控制点的图标都应该将这些控制点设置为蓝色。
6. **极简主义**：24x24 图标保留的细节不多。 避免使用精细特征，特别是当这些精细特征是区分图标的全部时。
7. **窗口主题感知**：即使撇开那些 “暗模式” 的操作系统不谈， KiCad 的窗口背景颜色在不同的操作系统上也会有很大的不同。 在使用可能会在窗口背景中消失的颜色时要小心。 特别是，在窗口背景下很难看到重音为灰色、白色和原理图的棕色。 仅在不同颜色的背景上使用这些颜色以创建足够的对比度。

## 2. 技术要求和许可

图标必须开发为 SVG 文件。编译和测试图标更改需要安装 Inkscape 和 pngcrush， 详见 bitmap_png 源目录中的 CMakeLists.txt 文件。 如果您的 pngcrush 版本与主要开发团队使用的版本 (1.8.13) 不匹配， 您将在源代码树中看到大量差异。

图标必须使用 SVG 文件中的嵌入式许可证作为署名-相同方式共享 (CC-by-SA) 进行许可。 可以使用文档属性对话框将其添加到 Inkscape 中。

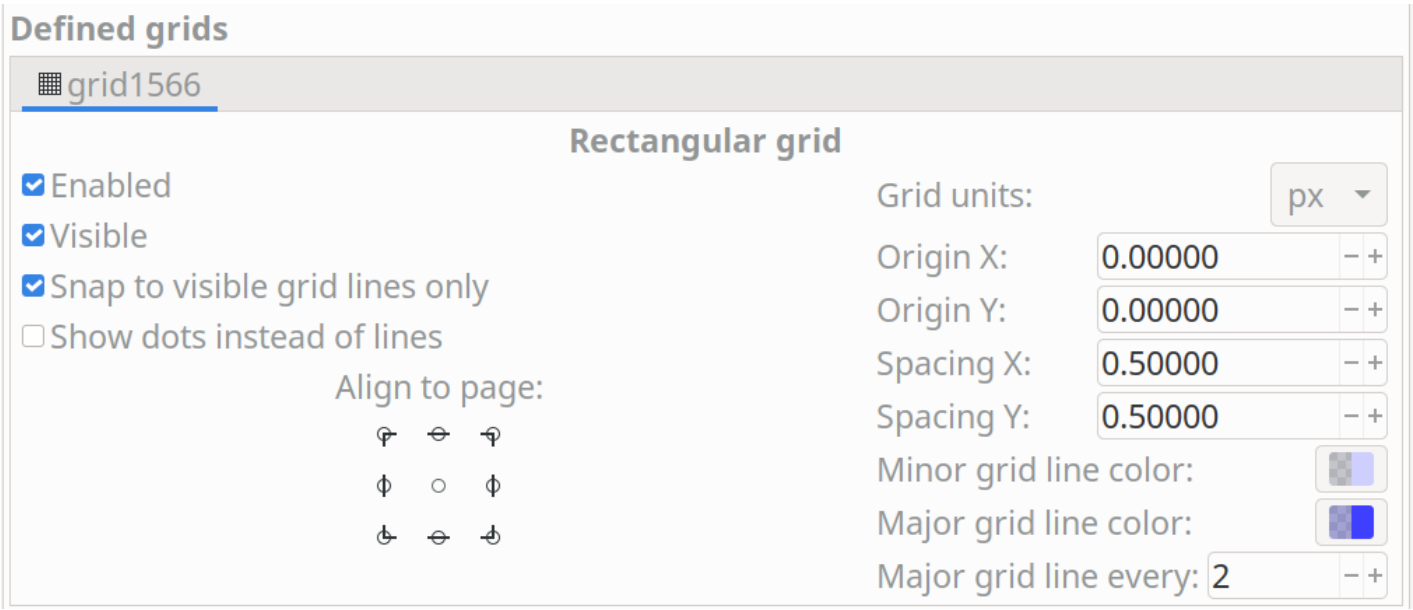
# 3. 创作软件

必须在 Inkscape 1.0 或更高版本中编辑 SVG 文件。 虽然其它软件包 (如 Adobe Illustrator) 可以导出 SVG 文件，但它们只能导出图标的外观。 他们经常跳过其他细节，例如图层、隐藏线、剪裁、网格和许可，这些细节使文件在将来更容易编辑。 我们建议使用 Inkscape 中的图标预览功能，以确保矢量图形与像素网格正确对齐，并且在目标大小下清晰可见。

# 4. 网格

图标文件必须在 Inkscape SVG 文件中设置网格。 网格单位必须为 `px`，原点为 `0,0`，间距为 `0.5px`。 次网格线颜色为 `#ceceff20` ，主网格线颜色为 `#3f3fff40` 。 每两条线应该有一条主要网格线。

这使得偶数和奇数大小的线宽都可以在网格上像素对齐。 这是在文档属性窗口中设置的，如下所示：




# 5. 图标大小

| 大小 | 用途 |
| --- | --- |
| 16x16 | • 一些 UI 按钮<br>• 外观面板可见性切换<br>• 文件图标 |
| 24x24 | • 所有工具栏按钮<br>• 所有菜单图标 (目前) |
| 48x48 | • KiCad 管理器中的主程序图标 |
| 128x128 | • 主程序图标 (ico 文件) |

# 6. 颜色

除极少数例外，仅将这些表中的颜色用于 SVG 对象 (由于混合，生成的位图中将存在其他颜色)。

## 6.1. 亮色主题

| 颜色名称 | 颜色值 | 用途/备注 |
| --- | --- | --- |
| Dark Gray (暗灰色) | `#545454` | 这是用于所有笔划的默认颜色。<br><br>请注意，在密集填充 (例如，保存、打印、粘贴图标) 中，它将比其他图标中相同的灰色显示得更暗。 对于这些情况，我们使用 `#606060` 。 |

| 颜色名称 | 颜色值 | 用途/备注 |
| --- | --- | --- |
| Medium Grey (中灰色) | `#848484` | 与深灰色相比，用于非活动对象以降低对比度 |
| Light Grey (浅灰色) | `#B9B9B9` | 用于通孔内的孔和图形形状内的填充 |
| Primary Blue (原色蓝) | `#1A81C4` | 用于强调某些细节，并用于 PcbNew 中的 “背面铜” |
| Light Blue (浅蓝色) | `#39B4EA` | 用于在某些地方显示区域填充，并作为强调色 |
| Primary Red (原色红) | `#BF2641` | 用于 PcbNew 中的强调、标识和 “前面铜” |
| Gold (金色) | `#F29100` | 用于通孔铜色，(很少) 用作强调 |
| PCB Green (PCB 绿色) | `#489648` | 用于布线的 PcbNew 程序图标 |
| PCB Dark Green (PCB 深绿色) | `#006400` | 用于阻焊层的 PcbNew 程序图标 |
| Schematic Tan (原理图色调) | `#D0C5AC` | 在 Eesschema 图标中使用，并用于表示原理图 |

| 颜色名称 | 颜色值 | 用途/备注 |
| --- | --- | --- |
| Pure White (纯白) | `#FFFFFF` | 用于与深色的对比和物体之间的分隔 |
| Off White (灰白色) | `#F3F3F3` | 用于较大的白色填充物 (例如新的)，而不是纯白色 |

## 6.2. 暗色主题

**尚未设计**

# 7. 线条和填充

SVG 文件中的线宽通常应为 `1.5px` 或更大，以确保在最终的位图中， 无论线条相对于像素网格的位置如何，都有具有真实颜色的像素。 水平线和垂直线通常应该是 `2px`，它们传达了图像的主要含义。 当 1px 线条是较大整体的一部分时，可以使用 1px 线条。

在可行的情况下，代表某些概念的线宽应该在整个图标集中保持一致。 例如，图形形状绘制工具总是使用 `2px` 线。

一般来说，图标集在适用的情况下使用 “扁平填充” 形状。 创建新图标时保持此视觉样式。 请注意，并非所有形状都必须填充 - 可以使用无填充的描边形状 (与窗口背景色一起) 来创建对比鲜明的填充区域。 笔画一般应该是等宽的，以避免看起来像手绘笔画 (“卡通风格”)。

几乎在所有情况下，填充都应该使用纯色。 渐变应该谨慎使用，通常不能用来创建 "3D" 效果或灯光/阴影的外观。

# 8. 字体

我们在图标中使用两种字体：Noto Sans 和 Tiresas LPFont Bold。 默认情况下，所有文本都使用 Noto Sans。 Tiresas 是用来做 “Ki” 标志的。

如果间距或线宽需要，Noto Sans 可以使用不同的重量和间距，例如 Noto Sans 粗体、半压缩。

Tiresas LPFont Bold 只能用作 Tiresas LPFont Bold，不得替换。

# 9. 标识

标识是覆盖在其他图标上以增加含义的符号。 这些符号来自一个共享库 (源目录中的 badges.svg)。 始终使用位置一致的适当标识 (您可以从标识 SVG 复制/粘贴)，而不是创建传达与标识相同含义的新图稿。

标识通常无需修改即可使用，但在某些情况下，在标识外部添加 1px-1.5px 的笔画边框 (对于浅色主题为纯白) 会很有帮助，以确保标识和下方图标内容之间的视觉分离。

# 提交规范

## 提交信息格式规范

提交消息应以简短的主题行开头。 请尝试将其限制为不超过 72 个字符。 邮件正文与主题行之间应用空行分隔，并以 72 个字符换行。 提交信息的内容应该解释这次提交的作用和原因。 不要描述改动是如何工作的，因为代码本身已经足够说明问题了。

## 将提交和问题联系起来

如果提交修复了 issue 跟踪器 中报告的问题， 请在提交消息中添加一行指示修复的问题编号。 在这种情况下，GitLab 将自动关闭该问题，并添加指向您在该问题中提交的链接。

比如，下面这一行就会自动关闭问题 #1234567：

```
Fixes https://gitlab.com/kicad/code/kicad/issues/1234567
```

有一个 alias 可以简化这一步骤。 您可以在 GitLab 文档 中阅读有关自动问题关闭的更多信息。

## 变更记录标签

为了帮助记录下面的代码改动，应该要包含一条给用户提示改动的变更记录标签。有三种变更 记录标签：

- `ADDED` 来表示新功能
- `CHANGED` 来表示对现有功能的改动
- `REMOVED` 来通知现有功能的删除

如果提交没有改变用户和软件的交互，比如代码重构，问题缺陷修复等，就没有必要添加变更 记录标签。变更记录标签的主要目的是生成发布说明来向文档维护者提示改动。一定要记得什 么情况下需要添加变更记录标签。

# 让文档开发者意识到改动

当推送有变更记录标签的提交时，提交者应该到 文档仓库 创建一个新问题，用于提示文档维护者。这个问题里包含一个关于这次提交的链接。

# 提取变更记录

得益于变更记录标签，我们很容易通过下面的 git 命令提取出变更记录：

```
    git log -E --grep="ADD[ED]?:|REMOVE[D]?:|CHANGE[D]?:" --since="1 Jan
2017"
    git log -E --grep="ADD[ED]?:|REMOVE[D]?:|CHANGE[D]?:" <commit hash>
```

KiCad 提供了一个 alias 来简化变更记录提取命令。

# 例子

下面是一个正确的提交信息格式的例子：

```
    Eeschema: Adding line styling options

    ADDED: Add support in Eeschema for changing the default line style,
    width and color on a case-by-case basis.

    CHANGED: "Wire" lines now optionally include data on the line style,
    width and color if they differ from the default.

    Fixes https://gitlab.com/kicad/code/kicad/issues/594059
    Fixes https://gitlab.com/kicad/code/kicad/issues/1405026
```

# git 别名文件

这个 `helpers/git/fixes_alias` 文件包含有一些很有用的 git 别名。可以通过 在代码仓库运行下面的命令来安装它：

```
git config --add include.path $(pwd)/helpers/git/fixes_alias
```

# 'fixes' 别名

一旦别名配置文件安装好后，就可以用它来修改最近一次提交的信息，使之包含问题报告的 链接：

```
git fixes 1234567
```

这个例子中，这条命令会在最后一次提交的信息里添加下面这一行内容：

```
Fixes https://gitlab.com/kicad/code/kicad/issues/1234567
```

# 'changelog' 别名

别名配置文件安装好后，你就可以通过下面的别名来提取变更日志：

```
git changelog --since="1 Jan 2017"
git changelog <commit hash>
```

# UI 规范

# 用户界面指南

本文档定义了 KiCad 中用户界面开发的指导原则。 开发人员在向 KiCad 项目贡献用户界面代码时应尽可能遵循这些指导原则。

## 文本大写

对于 KiCad 中使用的所有可见文本，请遵循中的建议。 GNOME 用户界面指南的写作风格部分 中的大写部分。 这适用于所有菜单、标题、标签、工具提示、按钮等。

应用程序名称的大写是 KiCad、Eesschema、CvPcb、GerbView 和 Pcbnew。 所有具有用户可见的应用程序名称的字符串都应按此方式大写。 在源代码注释中使用这种大写也是一个好主意，以防止混淆新的贡献者。

## 大写样式

GNOME 用户界面元素中使用了两种大写形式：标题大写和句子大写。 本节定义了大写样式以及应在何时使用每种类型的大写。

### 标题大写

使用标题大写时，除以下例外情况外，所有单词均为大写： * 文章: a, an, the. * 连词: and, but, for, not, so, yet ... * 三个或三个以下字母的介词： at, for, by, in, to ...

### 句子大写

当句子大写时，第一个单词的第一个字母大写，以及其他任何通常在句子中大写的单词，如专有名词。

## 大写单词表

下表指出了每种类型的用户界面元素要使用的大写样式。

| 元素 | 样式 |
| --- | --- |
| Check box labels (复选框标签) | Sentence (句子) |
| Command button labels (命令按钮标签) | Header (标题) |
| Column heading labels (列标题标签) | Header (标题) |
| Desktop background object labels (桌面背景对象标签) | Header (标题) |
| Dialog messages (对话框消息) | Sentence (句子) |
| Drop-down combination box labels (下拉组合框标签) | Sentence (句子) |
| Drop-down list box labels (下拉列表框标签) | Sentence (句子) |
| Field labels (字段标签) | Sentence (句子) |
| Text on web pages (网页上的文本) | Sentence (句子) |
| Group box and window frame labels (组框和窗口框架标签) | Header (标题) |
| Items in drop-down and list controls (下拉列表控件中的项) | Sentence (句子) |

| 元素 | 样式 |
| --- | --- |
| List box labels (列表框标签) | Sentence (句子) |
| Menu items (菜单项) | Header (标题) |
| Menu items in applications (应用程序中的菜单项) | Header (标题) |
| Menu titles in applications (应用程序中的菜单标题) | Header (标题) |
| Radio button labels (单选按钮标签) | Sentence (句子) |
| Slider labels (滑块标签) | Sentence (句子) |
| Spin box labels (数字显示框标签) | Sentence (句子) |
| Tabbed section titles (选项卡式部分标题) | Header (标题) |
| Text box labels (文本框标签) | Sentence (句子) |
| Titlebar labels (标题栏标签) | Header (标题) |
| Toolbar button labels (工具栏按钮标签) | Header (标题) |

| 元素 | 样式 |
| --- | --- |
| Tooltips (工具提示) | Sentence (句子) |
| Webpage titles and navigational elements (网页标题和导航元素) | Header (标题) |

# 对话框

本节定义应如何设计对话框。KiCad 项目使用 GNOME 用户界面指南 来布局对话框。 设计对话框时，请遵循 [GNOME 用户界面指南的可视化布局部分。 KiCad 的对话框可以用 wxformbuilder 设计，也可以手工创建。 但是，现有对话框必须以与其实现时相同的方式进行维护。 请使用 wxformbuilder-releases 以避免开发人员之间的版本不匹配。

## 退出键终止

请注意，仅当存在使用 ID wxID_Cancel 定义的对话框按钮，或者在对话框初始化期间调用使用 wxDialog::SetEscapeID( MY_ESCAPE_BUTTON_ID ) 设置退出按钮 ID 时，退出键终止才能正常工作。 前者是处理退出键对话终止的首选方法。 WxFormBuilder 中有一个用于设置 “默认” 控件的复选框，这是按下 “Enter” 键时触发的复选框。

## 使用 Sizers 的对话框布局

在所有对话框中使用wxWidget "sizers"，不管它们有多简单。 在 KiCad 中禁止在对话框中使用绝对大小调整。 有关使用 sizers 的更多信息，请参见 wxWidgets sizers 概述。 配置 sizers，以便在对话窗口展开时，sizers 自动对增加的对话窗口进行最合理的使用。 例如，在 Pcbnew 的 DRC 对话框中，应使用 sizers 来展开文本控件以使用全部可用的自由窗口区域， 以便用户在展开对话框窗口时最大化文本控件中项目的视图，从而更容易阅读更多 DRC 错误消息。 在没有一个组件比其他组件更重要的其他对话框中，可以将 sizers 配置为将控件定位到靠近。 对话框越变越大，不一定会将它们紧密捆绑在一起。 该对话框在大到足以显示所有用户界面元素的任何大小下都应该看起来很漂亮。

如果可能，请避免定义初始对话框大小。 让他们做好本职工作吧。 对话框适合大小调整后，请将最小大小设置为当前大小，以防止在调整对话框大小时遮挡对话框控件。 如果对话框控件的标签或文本是，请在运行时设置或更改。 重新运行 wxWindow::Fit() 以允许对话框调整大小并根据新的控件宽度进行调整。 这都可以在创建对话框之后、显示对话框之前完成，也可以根据需要使用类方法调整对话框大小。 将最小大小重置为更新后的对话框大小。

以 13 磅字体显示时，对话框窗口不应超过 1024 x 768。 请注意，最终用户使用的字体不是您可以从对话框中控制的，但出于测试目的，如果用户选择的字体大小为 13 磅， 请不要超过此对话框大小。 如果您的对话框超出此限制， 请使用选项卡或其他分页方法重新设计对话框，以减小对话框的大小。

# 基类对话框

KiCad 项目有一个基类，大多数对话框 (如果不是所有对话框) 都应该派生自该基类。 使用 wxFormBuilder 时，请在 “对话框” 选项卡中添加以下设置：

- subclass.name ← DIALOG_SHIM
- subclass.header ← dialog_shim.h

这将覆盖 Show( bool ) wxWindow() 函数，并提供会话的保留大小和位置。 有关更多信息， 请参见 DIALOG_SHIM 类源代码。

使用工具提示解释每个非明显控件的功能。 这一点很重要，因为帮助文件和维基经常落后于源代码。

# 在控件之间传输数据

对话框初始化时，对话框数据必须传输到对话框控件， 并在通过默认肯定操作 (通常单击 wxID_OK 按钮) 或单击 wxID_Apply 按钮关闭对话框时从控件传输。 WxWidgets 对话框框架通过使用验证器对此提供支持。 请阅读 wxWidgets 文档 中的 wxValidator 概述。 在过去，数据传输是在各种默认按钮处理程序中处理的，几乎所有的默认按钮处理程序都被破坏了。 不要在对话框代码中实现默认按钮处理程序。 使用验证器在控件之间传输数据，并允许默认对话框按钮处理程序按设计方式工作。

## 国际化

要生成对话框中出现的字符串列表，需要在项目属性中启用 'internationalize'(国际化) 复选框。 否则，将无法翻译该对话框。

# 字符串引号

通常会将文本字符串加引号以供显示，这些文本字符串可用于呈现 HTML 的控件中。 使用尖括号会给 HTML 呈现控件带来麻烦，所以文本应该用单引号 '' 引起来。 例如：

- 'filename.kicad_pcb'
- 'longpath/subdir'

- 'FOOTPRINTNAME'
- 'anything else'

# 组件

- 设置框架

  *设置框架管理应用程序设置以及工程。 本文档解释了如何使用该框架以及它的一些内部工作原理。 源代码指南 相关代码大部分在 common/settings 和 common/project 中。 C++ 类 描述 SETTINGS_MANAGER 加载和卸载设置文件和工程 JSON_SETTINGS 所有设置对象的基类。表示单个 JSON 文件。 NESTED_SETTINGS 存储在另一个 (即没有自己的文件) 中的 JSON_SETTINGS 对象 PARAM 参数：用于在 JSON_SETTINGS 中存储数据的 helper 类 APP_SETTINGS_BASE 应用程序 (框架) 设置的基类 COLOR_SETTINGS 设计用于存储颜色主题的 JSON_SETTINGS 的子类 COMMON_SETTINGS*

- 工具框架

  *本文档简要概述了 GAL 画布中工具系统的结构。 介绍 GAL (图形抽象层) 框架提供了一种强大的方法，可以轻松地向 KiCad 添加工具。 与旧的 “遗留” 画布相比，GAL 工具更灵活、更强大，也更容易编写。 GAL “工具” 是提供一个或多个要执行的 “操作” 的类。 动作可以是简单的一次性动作(例如 “放大” 或 “翻转对象” )， 也可以是交互过程(例如 “手动编辑多边形点”)。 Pcbnew GAL 中的一些工具示例如下： 选择工具 - "normal" 工具。此工具进入一种状态， 在该状态下可以将项目添加到选定对象的列表中， 然后其他工具即可对其执行操作。 (pcbnew/tools/selection_tool.cpp, pcbnew/tools/selection_tool.h) 编辑工具 - 此工具在组件被 "picked up" 时处于活动状态， 并跟踪鼠标位置以允许用户移动组件。 编辑的各个方面 (例如翻转) 也可由热键或其他工具使用。 (pcbnew/tools/edit_tool.cpp,pcbnew/tools/edit_tool.h) 绘图工具 - 此工具控制绘制图形元素 (如线段和圆) 的过程。 (pcbnew/tools/drawing_tool.cpp,pcbnew/tools/drawing_tool.h) 缩放工具 - 允许用户放大和缩小。 工具的主要部件工具有两个主要方面：操作和工具类。*

- 测试框架

  *单元测试 KiCad 的单元测试数量有限， 可用于检查某些功能是否正常工作。 测试是使用 CMake 的 CTest，组件注册的。 CTest 收集并运行所有不同的测试程序。 大多数 C++ 单元测试都是使用 Boost 单元测试框架编写的， 但这不是向测试套件添加测试所必需的。 测试 CMake 目标一般以 qa_ 开头，CTest 内的测试名称相同， 但没有 qa_ 前缀。 运行测试 您可以使用 make test 或 ctest 进行构建后的所有测试。 后一选项允许许多 CTest 选项，这些选项可能非常有用，尤其是在自动化或 CI 环境中。 运行特定测试 如果要运行特定的测试可执行文件，可以直接使用 ctest 运行，也可以直接运行该可执行文件。 在处理特定测试并且您希望访问测试可执行文件的参数时，直接运行通常是最简单的方式。 例如： # 运行 libcommon 测试 cd /path/to/kicad/build qa/common/qa_common [parameters] 对于 Boost 单元测试，可以使用 <test>--help 查看测试选项。 常见的有用模式： <test> -t "Utf8/*" 运行 Utf8 测试套件中的所有测试。*

- S-表达式

  *S-表达式 是与 XML 相同的文本流或字符串，由一系列元素组成。 每个元素要么是原子，要么是列表。 原子对应于字符串，而列表对应于 s-表达式。 下面的语法代表了我们对 s-表达式 的定义： sexpr ::= ( sx ) sx ::= atom sxtail | sexpr sxtail | NULL sxtail ::= sx | NULL atom :: quoted | value quoted :: "ws_string" value :: nws_string 原子可以是带引号的字符串 (由双引号括起来的包含空格的字符串)，也可以是不需要用引号引起来的非空格字符串。 KiCad 中使用的 s-expression 语法使用两种引用/语法策略， 这两种策略是由 Specctra DSN 规范的需求和我们自己的非 specctra 需求提供的。 Specctra DSN 规范在引用方面不是很清楚，最重要的是 Freerouter 的解释， 由于希望与 Freerouter 兼容，它实际上无论如何都会取代 Specctra DSN 规范中的任何内容。 我们有自己的需求，超出了 Specctra DSN 规范的需求，因此我们支持引用原子的两种语法或引用协议：*

# 设置框架

设置框架管理应用程序设置以及工程。 本文档解释了如何使用该框架以及它的一些内部工作原理。

## 源代码指南

相关代码大部分在 `common/settings` 和 `common/project` 中。

| C++ 类 | 描述 |
| --- | --- |
| `SETTINGS_MANAGER` | 加载和卸载设置文件和工程 |
| `JSON_SETTINGS` | 所有设置对象的基类。表示单个 JSON 文件。 |
| `NESTED_SETTINGS` | 存储在另一个 (即没有自己的文件) 中的 `JSON_SETTINGS` 对象 |
| `PARAM` | 参数：用于在 `JSON_SETTINGS` 中存储数据的 helper 类 |
| `APP_SETTINGS_BASE` | 应用程序 (框架) 设置的基类 |
| `COLOR_SETTINGS` | 设计用于存储颜色主题的 `JSON_SETTINGS` 的子类 |
| `COMMON_SETTINGS` | 适用于 KiCad 每个组件的设置 |

| C++ 类 | 描述 |
| --- | --- |
| `PROJECT_FILE` | 表示工程 (`.kicad_pro` ) 文件的 `JSON_SETTINGS` |
| `PROJECT_LOCAL_SETTINGS` | 表示工程本地状态 (`.kicad_prl` ) 文件的 `JSON_SETTINGS` |

# 存储设置的位置

设置可能存储在四个主要位置：

1. 在 `COMMON_SETTINGS` 中：这是 KiCad 所有部分共享的设置。
2. 在应用程序设置对象 (`APP_SETTINGS_BASE` 的子类) 中。 这些对象 (如 `EESCHEMA_SETTINGS` 和 `PCBNEW_SETTINGS` ) 存储特定于 KiCad 一部分的设置。 具体地说，这些对象是在其各自应用程序的上下文中编译的，因此它们可以访问可能不属于 `common` 的数据类型。
3. 在 `PROJECT_FILE` 中，它们将特定于加载的工程。 在 Board/Schematic Setup (电路板/原理图设置) 对话框中的大多数设置都是如此。 目前 KiCad 只支持一次加载一个工程，代码中的很多地方都希望有一个 `PROJECT` 对象始终可用。 因此，即使用户没有加载任何工程，`SETTINGS_MANAGER` 也会始终确保有一个 "dummy" 的 `Project_FILE` 可用。 此虚拟工程可以在内存中修改，但不能保存到磁盘。
4. 在 `PROJECT_LOCAL_SETTINGS` 对象中，它们将特定于加载的工程。 该文件用于 “本地状态” 的设置，例如哪些板层是可见的， 这些设置不应该 (至少对许多用户而言) 检出到源代码管理中。 这里的任何设置都应该是暂时的，这意味着如果删除整个文件不会有不良影响。

# JSON_SETTINGS

`JSON_SETTINGS` 类是设置基础结构的主干。 是 `thirdparty/nlohmann_json/nlohmann/json.hpp` 提供的 `nlohmann::json::basic_json` 类的子类。 因此，任何时候需要对底层 JSON 数据进行原始操作时，都可以使用

标准 nlohmann::json API。 JSON 内容表示 **文件在磁盘上的状态**，而不是向 C++ 公开的数据的状态。 两者之间的同步是通过参数(见下文)完成的，并且在从磁盘加载之后和保存到磁盘之前立即发生。

# 参数

参数建立 C 数据和 JSON 文件内容之间的链接。 一般情况下，参数由 **路径**、**指针** 和 **默认值** 组成。 路径是 `“x.y.z”` 形式的字符串，其中每个组件代表一个嵌套的 JSON 字典键。 指针是指向 C 成员变量的指针，该变量保存 `JSON_SETTINGS` 的使用者可访问的数据。 默认值用于在 JSON 文件中缺少数据时更新指针。

参数是 `include/settings/parameters.h` 中 `PARAM_BASE` 的子类。 创建了许多有用的子类，使在 JSON 文件中存储复杂数据变得更容易。

基本的 `PARAM` 类是模板化的，用于存储任何可以自动序列化为 JSON 的数据类型。 `PARAM` 的基本实例化可能如下所示：

```
    m_params.emplace_back( new PARAM<int>( "appearance.icon_scale",
                                           &m_Appearance.icon_scale, 0 )
);
```

这里，`m_Appearance.icon_scale` 是设置对象 (`struct` 内部的 `int`) 的公共成员。 `"appearance.icon_scale"` 为 JSON 文件中的 **路径**，默认为`0`。这将导致 JSON 如下所示：

```
{
    "appearance": {
        "icon_scale": 0
    }
}
```

请注意，自定义类型可以与 `PARAM<>` 一起使用，只要定义了 `to_json` 和 `from_json` 即可。 有关这方面的示例，请参阅 `COLOR4D` 。

对于存储复杂的数据类型，有时使用 `PARAM_LAMBDA<>` 最简单，它允许您将 "getter" 和 "setter" 定义为参数定义的一部分。 您可以使用它来构建一个 `nlohmann::json` 对象，并将其存储为您的参数的 "value"。 有关实现方法的示例，请参阅 `NET_SETTINGS` 。

# NESTED_SETTINGS

`NESTED_SETTINGS` 类类似于 `JSON_SETTINGS` ，但它使用另一个 `JSON_SETTINGS` 对象作为后备存储器的文件。 `NESTED_SETTINGS` 的全部内容作为特定键的值存储在父文件中。 这有两个主要好处：

1. 您可以将大型设置拆分成更易于管理的部分
2. 您可以向父设置对象隐藏有关嵌套设置的知识

例如，工程文件的很多部分都作为 `NESTED_SETTINGS` 对象存储在 `PROJECT_FILE` 中。 这些对象包括 `SCHEMATIC_SETTINGS` 、`NET_SETTINGS` 和 `BOARD_DESIGN_SETTINGS` ， 它们被编译为 eesschema 或 pcbnew 的一部分，因此它们可以访问公共不可用的数据类型 (其中编译了 `PROJECT_FILE` )。

加载外部文件时，嵌套设置的所有数据都在底层的 `nlohmann::json` 数据存储中 — 只是在加载适当的 `NESTED_SETTINGS` 之前不会使用它。

`NESTED_SETTINGS` 对象的生命周期可以比父对象短。 这是必需的，因为在某些情况下 (例如使用项目文件)， 父级可以驻留在一个帧中 (例如 KiCad 管理器)，而可以创建和销毁其中使用嵌套设置的帧。 当嵌套设置被销毁时，它会确保将其数据存储到父级的 JSON 数据中。 如果需要，可以将父 `JSON_SETTINGS` 保存到磁盘。

# 架构版本和迁移

设置对象有一个 **架构版本**，它是一个常量整数，在需要迁移时可以递增。 将代码中的模式版本与加载的文件中的模式版本进行比较，如果文件版本较低 (较旧)，则运行 **迁移** 以使文件中的数据保持最新。

**迁移** 是负责从一个架构版本到另一个架构版本进行必要更改的函数。 它们在参数加载之前作用于 **底层 JSON 数据**。

更改设置文件时并不总是需要迁移。 您可以自由添加或删除参数，而无需更改模式版本或编写迁移。 如果添加参数，它们将被添加到 JSON 文件并初始化为其默认值。 如果删除参数，它们将在下次保存设置时从 JSON 文件中静默删除。 只有在需要根据当前状态对 JSON 文件进行更改时，才需要迁移。 例如，如果您决定重命名设置键，但希望保留用户的当前设置。

如果需要对设置文件进行 “突破性更改”，请执行以下操作：

1. 增加模式版本

2. 编写迁移，对底层的 `nlohmann::json` 对象进行必要的更改

3. 在 object 的构造函数中调用 `JSON_SETTINGS::registerMigration`

# 工具框架

本文档简要概述了 GAL 画布中工具系统的结构。

# 介绍

GAL (图形抽象层) 框架提供了一种强大的方法，可以轻松地向 KiCad 添加工具。 与旧的 “遗留” 画布相比，GAL 工具更灵活、更强大，也更容易编写。

GAL “工具” 是提供一个或多个要执行的 “操作” 的类。 动作可以是简单的一次性动作(例如 “放大” 或 “翻转对象” )， 也可以是交互过程(例如 “手动编辑多边形点”)。

Pcbnew GAL 中的一些工具示例如下：

- 选择工具 - "normal" 工具。此工具进入一种状态， 在该状态下可以将项目添加到选定对象的列表中， 然后其他工具即可对其执行操作。 (pcbnew/tools/selection_tool.cpp, pcbnew/tools/selection_tool.h)
- 编辑工具 - 此工具在组件被 "picked up" 时处于活动状态， 并跟踪鼠标位置以允许用户移动组件。 编辑的各个方面 (例如翻转) 也可由热键或其他工具使用。 (pcbnew/tools/edit_tool.cpp,pcbnew/tools/edit_tool.h)
- 绘图工具 - 此工具控制绘制图形元素 (如线段和圆) 的过程。 (pcbnew/tools/drawing_tool.cpp,pcbnew/tools/drawing_tool.h)
- 缩放工具 - 允许用户放大和缩小。

# 工具的主要部件

工具有两个主要方面：操作和工具类。

## 工具操作

`TOOL_ACTION` 类充当 GAL 框架调用工具提供的操作的句柄。 一般来说，每个操作，不管是交互的还是非交互的，都有一个 `TOOL_ACTION` 实例。 这提供了：

- 名称，格式为 `pcbnew.ToolName.actionName` ，内部用于分配操作
- “范围”，确定工具何时可用：
- 当操作特定于特定工具时，返回 `AS_CONTEXT` 。 例如， `pcbnew.InteractiveDrawing.incWidth` 会在线条仍在绘制时增加线条的宽度。
- `AS_GLOBAL` ，当工具总是可以通过热键调用时，或者在执行其他工具时。 例如，在交互式选择过程中，可以从选择工具的菜单访问缩放操作。
- “默认热键”，当用户没有提供自己的配置时使用。
- “菜单项”，即从菜单访问此工具时显示的(可翻译的)字符串。
- “菜单描述”，它是菜单项工具提示中显示的字符串，并在需要时提供更详细的描述。
- 一个“图标”，显示在菜单和操作按钮上
- “标志” 包括：
- `AF_ACTIVATE` ，表示工具进入活动状态。当工具处于活动状态时， 它将不断接收 UI 事件，如鼠标单击或按键， 这些事件通常在事件循环中处理 (请参阅 TOOL_INTERACTIVE::Wait())。
- 一个参数，它允许不同的操作以不同的效果调用相同的函数，例如 “左一步” 和 “右一步”。

# 工具类

GAL 工具继承了 `Tool_BASE` 类。 Pcbnew 工具一般继承自 `PCB_TOOL` ，即 `TOOL_INTERACTIVE` ，即 `TOOL_BASE` 。 Eeschema 工具直接继承自 `TOOL_INTERACTIVE` 。

工具的工具类可以是相当轻量级的 - 很多功能都是从工具的基类继承而来的。 这些基类提供了对几项内容的访问，特别是：

- 访问父框架 (一个 `wxWindow` ，可用于修改视口、设置光标和状态栏内容等)。
- 使用函数 `getEditFrame<T>()` ，其中 `T` 是您想要的 frame 子类。 在 `PCB_TOOL` 中，可能是 `PCB_EDIT_FRAME` 。
- 访问 `TOOL_MANAGER` ，该 `TOOL_MANAGER` 可用于访问其他工具的操作。
- 访问支持该工具的 “模型” (某种 `EDA_ITEM` )。

- 使用 `getModel<T>()` 访问。在 `PCB_TOOL` 中，型号类型 `T` 为 `BOARD`， 可用于访问和修改 PCB 内容。
- 访问 `KIGFX::VIEW` 和 `KIGFX::VIEW_CONTROLS` ，用于操作 GAL 画布。

工具实现的主要部分是 `TOOL_MANAGER` 用于设置和管理工具的函数：

- 构造函数和析构函数来建立所需的任何类成员。
- TOOL_BASE 类需要传递一个字符串作为工具名称， 通常类似于 `pcbnew.ToolName` 。
- `Init()` 函数 (可选)，通常用于填写属于此工具的上下文菜单， 或访问其他工具的菜单并向其中添加项目。 当工具注册到工具管理器时， 此函数被调用一次。
- `Reset()` 函数，在重新加载模型 (例如， `BOARD`) 时、 切换 GAL 画布时以及在工具注册之后调用。 必须在此函数中释放从 GAL 视图或模型声明的任何资源， 因为它们可能会变为无效。
- `setTrantions()` 函数，该函数将工具操作映射到工具类中的函数。
- 调用操作时要调用的一个或多个函数。 如果需要，许多操作都可以调用相同的函数。 这些函数具有以下签名：
- int TOOL_CLASS::FunctionName( const TOOL_EVENT& aEvent )
- 返回 `0` 表示成功。
- 这些函数由 `TOOL_MANAGER` 在关联事件到达时调用 (关联是通过 TOOL_INTERACTIVE::Go() 函数创建的)。
- 它们通常可以是私有的，因为它们不是由任何其他代码直接调用的， 而是由工具管理器的协程框架根据 `setTrantions()` 映射调用的。

# 交互式操作

交互式动作的动作处理程序在循环中处理来自工具管理器的重复动作， 直到指示工具应该退出的动作为止。

交互式工具通常还会通过光标更改和设置状态字符串来指示它们处于活动状态。

```
    int TOOL_NAME::someAction( const TOOL_EVENT& aEvent )
    {
        auto& frame = *getEditFrame<PCB_EDIT_FRAME>();

        // 设置工具提示和光标（实际上看起来像十字准线）
        frame.SetToolID( ID_LOCAL_RATSNEST_BUTT,
                wxCURSOR_PENCIL, _( "Select item to move left" ) );
        getViewControls()->ShowCursor( true );

        // 激活该工具，现在它将是第一个接收事件的工具。
        // 如果您正在为单个操作（例如放大）编写处理程序，
        // 那么它将是第一个接收事件的工具，
        // 而不是需要更多事件才能操作（例如拖动组件）的交互式工具。

        Activate();

        // 主事件循环
        while( OPT_TOOL_EVENT evt = Wait() )
        {
            if( evt->IsCancel() || evt->IsActivate() )
            {
                // 交互式工具结束
                break;
            }
            else if( evt->IsClick( BUT_LEFT ) )
            {
                // 在这里做点什么
            }
            // 其他事件...
        }

        // 将印刷电路板框架重置为我们拿到它时的状态
        frame.SetToolID( ID_NO_TOOL_SELECTED, wxCURSOR_DEFAULT,
wxEmptyString );
        getViewControls()->ShowCursor( false );

        return 0;
    }
```

# 工具菜单

顶级工具（即用户直接输入的工具）通常提供其自己的上下文菜单。 仅从其他工具的交互模式调用的工具会将其菜单项添加到这些工具的菜单中。

要在顶级工具中使用 `TOOL_MENU` ，只需添加一个作为成员， 并在构造时引用工具进行初始化：

```
class TOOL_NAME: public PCB_TOOL
{
public:
    TOOL_NAME() :
        PCB_TOOL( "pcbnew.MyNewTool" ),
        m_menu( *this )
    {}

private:
    TOOL_MENU m_menu;
}
```

然后，您可以添加菜单访问器， 或提供自定义函数以允许其他工具添加您认为合适的任何其他操作或子集。

然后，您可以通过调用 `m_menu.ShowContextMenu()` 从交互式工具循环调用菜单。 单击此菜单中的工具条目将触发操作 - 在工具的事件循环中不需要进一步的操作。

# 提交对象

`COMMIT` 类管理对 `EDA_ITEMS` 的更改， 该更改将任意数量的项目的更改合并到单个撤消/重做操作中。 编辑 PCB 时，对 PCB 的更改由派生的 `BOARD_COMMIT` 类管理。

该类以 `PCB_BASE_FRAME` 或 `PCB_TOOL` 作为参数。 对于 GAL 工具，使用 `PCB_TOOL` 更合适， 因为如果不需要，则不需要查看 frame 类。

提交的过程是：

- 构造适当的 `COMMIT` 对象
- 在修改任何项之前，请使用 `Modify( item )` 将其添加到提交中， 以便将当前项状态存储为撤销点。
- 添加新项时，调用 `Add( item )` 。 除非您要中止提交，否则不要删除添加的项目。
- 移除条目时，调用 `Remove( item )` 。 您不应删除已删除的项目，它将存储在撤消缓冲区中。
- 使用 `Push( "Description" )` 完成提交。 如果您没有执行任何修改、添加或删除操作，则这是一个禁止操作，因此在推送之前不需要检查您是否做了任何更改。

如果您想要中止提交，可以直接销毁它，而不需要调用 `Push()` 。 基础模型将不会更新。

例如：

```
// 从当前 PCB_TOOL 构造提交
BOARD_COMMIT commit( this );

BOARD_ITEM* modifiedItem = getSomeItemToModify();

// 告诉提交我们要更改项目
commit.Modify( modifiedItem );

// 更新项目
modifiedItem->Move( x, y );

// 创建新项目
PCB_SHAPE* newItem = new PCB_SHAPE;

// ... 在此设置项目

// 添加到提交
commit.Add( newItem );

// 更新模型并添加撤消点
commit.Push( "Modified one item, added another" );
```

# 教程：添加新工具

不过详细介绍 GAL 工具框架是如何在表面下实现的， 让我们看看如何向 Pcbnew 添加一个全新的工具。 我们的工具将具有以下 (相当无用) 功能：

- 一个交互式工具，允许用户选择一个点， 从该点的项目中进行选择，然后将该项目向左移动 10 mm。
- 在此模式下，上下文菜单将有更多选项：
- "normal" 画布缩放和网格选项的使用
- 一种非交互式工具，它将在固定点添加一个固定的圆。
- 一种从 PCB_EDITOR_CONTROL 工具调用非交互式 "unfill all zones" 工具的方法。

# 声明工具操作 {#declare-actions}

第一步是添加工具操作。 我们将实施两个名为：

- `Pcbnew.UselessTool.MoveItemLeft` - 交互式工具
- `Pcbnew.UselessTool.FixedCircle` - 非交互式工具。

名为 `pcbnew.EditorControl.zoneUnfillAll` 的 “取消填充工具” 已存在

本指南假设我们将向 Pcbnew 添加一个工具， 但其他支持 GAL 的画布的过程与此类似。

在 `pcbnew/tools/pcb_actions.h` 中，我们在 `PCB_ACTIONS` 类中添加了以下内容， 该类声明了我们的工具：

```
static TOOL_ACTION uselessMoveItemLeft;
static TOOL_ACTION uselessFixedCircle;
```

动作的定义通常发生在相关工具的 .cpp 中。 定义发生在哪里实际上并不重要 (声明足以使用操作)，只要它最终是链接的。 类似的工具应该始终一起定义。

在我们的例子中，因为我们正在制作一个新工具， 所以它将位于 `pcbnew/tools/unuble_tool.cpp` 中。 如果向现有工具添加操作，则工具字符串的前缀 (例如 `"Pcbnew.UselessTool"` ) 将是在何处定义工具的强指示符。

工具定义如下所示：

```
TOOL_ACTION COMMON_ACTIONS::uselessMoveItemLeft(
        "pcbnew.UselessTool.MoveItemLeft",
        AS_GLOBAL, MD_CTRL + MD_SHIFT + int( 'L' ),
        _( "将项目左移" ), _( "选择并向左移动项目" ) );

TOOL_ACTION COMMON_ACTIONS::uselessFixedCircle(
        "pcbnew.UselessTool.FixedCircle",
        AS_GLOBAL, MD_CTRL + MD_SHIFT + int( 'C' ),
        _( "固定圆" ), _( "在固定位置添加固定大小的圆" ),
        add_circle_xpm );
```

我们为每个操作定义了热键，它们都是全局的。 这意味着您可以分别使用 `Shift+Ctrl+L` 和 `Shift-Ctrl-C` 访问各个工具。

我们为其中一个工具定义了一个图标，该图标应该出现在项目添加到的任何菜单中， 以及给定的标签和说明性工具提示。

我们现在定义了两个操作，但是它们没有连接到任何东西。 我们需要定义实现正确操作的函数。 您可以将这些添加到现有的工具中 (例如，`PCB_EDITOR_CONTROL` ，它处理许多常规的 PCB 修改操作，如区域填充)， 或者您可以编写一个全新的工具来保持独立，并为您添加工具状态提供更大的空间。

我们将编写自己的工具来演示该过程。

# 添加工具类声明

添加一个新的工具类头 `pcbnew/tools/inusable_tool.h` ， 包含如下类：

```
class USELESS_TOOL : public PCB_TOOL
{
public:
    USELESS_TOOL();
    ~USELESS_TOOL();

    ///> 对模型/视图更改做出反应
    void Reset( RESET_REASON aReason ) override;

    ///> 基本初始化
    bool Init() override;

    ///> 将处理程序绑定到相应的 TOOL_ACTIONs
    void setTransitions() override;

private:
    ///> “向左移动选定项” 交互式工具
    int moveLeft( const TOOL_EVENT& aEvent );

    ///> 执行左移操作的内部函数
    void moveLeftInt();

    ///> 添加固定大小的圆
    int fixedCircle( const TOOL_EVENT& aEvent );

    ///> 该工具显示的菜单模型。
    TOOL_MENU m_menu;
};
```

# 实现工具类方法

在 `pcbnew/tools/inusable_tool.cpp` 中，实现所需的方法。 在此文件中，您还可以添加免费函数助手、其他类等。

您需要将该文件添加到 `pcbnew/CMakeLists.txt` 中进行构建。

下面您将找到 useless_tool.cpp 的内容：

```cpp
#include "useless_tool.h"

#include <class_draw_panel_gal.h>
#include <view/view_controls.h>
#include <view/view.h>
#include <tool/tool_manager.h>
#include <board_commit.h>

// 用于框架工具 ID 值
#include <pcbnew_id.h>

// 用于操作图标
#include <bitmaps.h>

// 项目工具可以作用于
#include <class_board_item.h>
#include <class_drawsegment.h>

// 访问其他 PCB 操作和工具
#include "pcb_actions.h"
#include "selection_tool.h"


/*
  * 特定于工具的操作定义
  */
TOOL_ACTION PCB_ACTIONS::uselessMoveItemLeft(
        "pcbnew.UselessTool.MoveItemLeft",
        AS_GLOBAL, MD_CTRL + MD_SHIFT + int( 'L' ),
        _( "将项目左移" ), _( "选择并向左移动项目" ) );

TOOL_ACTION PCB_ACTIONS::uselessFixedCircle(
        "pcbnew.UselessTool.FixedCircle",
        AS_GLOBAL, MD_CTRL + MD_SHIFT + int( 'C' ),
        _( "固定圆" ), _( "在固定位置添加固定大小的圆" ),
        add_circle_xpm );

/*
  * USELESS_TOOL 实现
  */

USELESS_TOOL::USELESS_TOOL() :
        PCB_TOOL( "pcbnew.UselessTool" ),
        m_menu( *this )
{
}


USELESS_TOOL::~USELESS_TOOL()
{}
```

```
void USELESS_TOOL::Reset( RESET_REASON aReason )
{
}


bool USELESS_TOOL::Init()
{
    auto& menu = m_menu.GetMenu();

    // 添加我们自己工具的操作
    menu.AddItem( PCB_ACTIONS::uselessFixedCircle );
    // 添加 PCB_EDITOR_CONTROL's 的区域取消填充所有操作
    menu.AddItem( PCB_ACTIONS::zoneUnfillAll );

    // 添加标准缩放和栅格工具动作
    m_menu.AddStandardSubMenus( *getEditFrame<PCB_BASE_FRAME>() );

    return true;
}

void USELESS_TOOL::moveLeftInt()
{
    // 我们将调用选择工具上的操作来获取当前选择。
    // 选择工具将处理项目的歧义消除
    PCB_SELECTION_TOOL* selectionTool = m_toolMgr-
>GetTool<PCB_SELECTION_TOOL>();
    assert( selectionTool );

    // 调用操作
    m_toolMgr->RunAction( PCB_ACTIONS::selectionClear, true );
    m_toolMgr->RunAction( PCB_ACTIONS::selectionCursor, true );
    selectionTool->SanitizeSelection();

    const SELECTION& selection = selectionTool->GetSelection();

    // 未选择任何内容，返回到事件循环
    if( selection.Empty() )
        return;

    BOARD_COMMIT commit( this );

    // 迭代 BOARD_ITEM* 容器，移动每个项目
    for( auto item : selection )
    {
        commit.Modify( item );
        item->Move( wxPoint( -5 * IU_PER_MM, 0 ) );
    }

    // push commit - 如果选择为空，则这是无操作
    commit.Push( "Move left" );
}

int USELESS_TOOL::moveLeft( const TOOL_EVENT& aEvent )
{
```

```
    auto& frame = *getEditFrame<PCB_EDIT_FRAME>();

    // 设置工具提示和光标 (实际上看起来像十字准线)
    frame.SetToolID( ID_NO_TOOL_SELECTED,
            wxCURSOR_PENCIL, _( "选择要向左移动的项目" ) );

    getViewControls()->ShowCursor( true );

    Activate();

    // 只要工具处于活动状态，就处理工具事件
    while( OPT_TOOL_EVENT evt = Wait() )
    {
        if( evt->IsCancel() || evt->IsActivate() )
        {
            // 交互式工具结束
            break;
        }
        else if( evt->IsClick( BUT_RIGHT ) )
        {
            m_menu.ShowContextMenu();
        }
        else if( evt->IsClick( BUT_LEFT ) )
        {
            // 调用主操作逻辑
            moveLeftInt();

            // 保持显示编辑光标
            getViewControls()->ShowCursor( true );
        }
    }

    // 将 PCB 框架重置为我们得到的原样
    frame.SetToolID( ID_NO_TOOL_SELECTED, wxCURSOR_DEFAULT, wxEmptyString
);
    getViewControls()->ShowCursor( false );

    // 退出操作
    return 0;
}


int USELESS_TOOL::fixedCircle( const TOOL_EVENT& aEvent )
{
    // 要添加的新圆 (理想情况下使用智能指针)
    PCB_SHAPE* circle = new PCB_SHAPE;

    // 设置圆形属性
    circle->SetShape( S_CIRCLE );
    circle->SetWidth( 5 * IU_PER_MM );
    circle->SetStart( wxPoint( 50 * IU_PER_MM, 50 * IU_PER_MM ) );
    circle->SetEnd( wxPoint( 80 * IU_PER_MM, 80 * IU_PER_MM ) );
    circle->SetLayer(  LAYER_ID::F_SilkS );
```

```
    // 把圆形交到 BOARD 上
    BOARD_COMMIT commit( this );
    commit.Add( circle );
    commit.Push( _( "画一个圆形" ) );

    return 0;
}


void USELESS_TOOL::setTransitions()
{
    Go( &USELESS_TOOL::fixedCircle,
PCB_ACTIONS::uselessFixedCircle.MakeEvent() );
    Go( &USELESS_TOOL::moveLeft,
PCB_ACTIONS::uselessMoveItemLeft.MakeEvent() );
}
```

# 注册该工具

最后一步是在工具管理器中注册工具。

对于任何支持该工具的 `EDA_DRAW_FRAME` ， 这都是在框架的 `setupTools()` 函数中完成的。

# 编译并运行

完成所有操作后，您应该已经修改了以下文件：

- `pcbnew/tools/common_actions.h` - 操作声明
- `pcbnew/tools/useless_tool.h` - 工具头文件
- `pcbnew/tools/useless_tool.cpp` - 操作定义和工具实现
- `pcbnew/tools/tools_common.cpp` - 工具的注册
- `pcbnew/CMakeLists.txt` - 用于编译建新的 .cpp 文件

当您运行 Pcbnew 时，您应该可以按 `Shift+Ctrl+L` 进入 “向左移动项目” 工具-光标将变为十字准线， 并在右下角出现 “选择要向左移动的项目”。

当你点击鼠标右键时，你会看到一个菜单， 其中包含一个用于我们的 “创建固定圆” 工具的条目和 一个用于我们添加到菜单中的现有的 “取消填充所有区域” 工具的条目。 您也可以使用 `Shift+Ctrl+C` 进入固定圆操作。

恭喜您，您刚刚创建了您的第一个 KiCad 工具！

# 测试框架

## 单元测试

KiCad 的单元测试数量有限， 可用于检查某些功能是否正常工作。

测试是使用 CMake 的 CTest，组件注册的。 CTest 收集并运行所有不同的测试程序。 大多数 C++ 单元测试都是使用 Boost 单元测试框架编写的， 但这不是向测试套件添加测试所必需的。

测试 CMake 目标一般以 `qa_` 开头，CTest 内的测试名称相同， 但没有 `qa_` 前缀。

## 运行测试

您可以使用 `make test` 或 `ctest` 进行构建后的所有测试。 后一选项允许许多 CTest 选项，这些选项可能非常有用，尤其是在自动化或 CI 环境中。

### 运行特定测试

如果要运行特定的测试可执行文件，可以直接使用 `ctest` 运行，也可以直接运行该可执行文件。 在处理特定测试并且您希望访问测试可执行文件的参数时，直接运行通常是最简单的方式。 例如：

```
# 运行 libcommon 测试
cd /path/to/kicad/build
qa/common/qa_common [parameters]
```

对于 Boost 单元测试，可以使用 `<test>--help` 查看测试选项。 常见的有用模式：

- `<test> -t "Utf8/*"` 运行 `Utf8` 测试套件中的所有测试。
- `<test> -t "Utf8/UniIterNull"` 仅在特定套件中运行单个测试。
- `<test> -l all` 向输出添加更详细的调试。

- `<test> --list_content` 中列出测试套件和测试用例。 测试程序。您可以使用这些参数作为 `-t` 的参数。

您可以使用 CMake 只重建特定的测试，避免在小范围工作时重建所有测试， 例如 `make qa_common` 。

## 自动化测试

单元测试可以在自动化持续集成 (CI) 系统上运行。

默认情况下，测试输出人类可读的结果，这在开发或调试时很有用，但对于自动测试报告则不太有用。 可以解析 XML 测试结果的系统可以通过将 `KICAD_TEST_XML_OUTPUT` 选项设置为 `ON` 来启用这些功能。 然后将测试结果输出为 `qa` 子目录中以 `.xml` 结尾的文件。

测试结果按如下方式写入 Build 目录：

- Boost 单元测试：每个测试都有一个扩展名为的 XML 文件 `.boost-results.xml`
- Python 单元测试：每个扩展名为的测试一个目录 `.xunit-results.xml` . 这些目录中的每个 Python 测试用例文件都包含一个 `.xml` 文件。

# 编写 Boost 测试

Boost 单元测试编写起来很简单。 单独的测试用例可以注册到：

```
BOOST_AUTO_TEST_CASE( SomeTest )
{
    BOOST_CHECK_EQUAL( 1, 1 );
}
```

有一系列像 `BOOST_CHECK`这样的函数，它们记录在 这里。 最好使用最具体的函数，因为这样 `Boost` 可以提供更详细只报告不匹配， `BOOST_CHECK_EQUAL( foo, bar )` 将显示每个错误的值。

若要输出调试消息，您可以在单元测试中使用 `BOOST_TEST_MESSAGE` ， 只有在 `-l` 参数设置为 `message` 或更高的时候才能看到， 这样文本的颜色会有所不同，以区别于其他测试消息和标准输出。

您也可以使用 `std::cout` 、`printf` 、`wxLogDebug` 等来处理测试函数内部的调试消息（即您没有权限访问 Boost 单元测试头文件的地方）。这些文件总是打印出来的，所以在提交之前一定要删除它们，否则当 KiCad 正常运行时它们就会出现！

## 预期失败

有时，检查未通过的测试很有帮助。但是，故意检查提交中断编译（如果导致 `make test` 失败就会发生这种情况）是不好的做法。

Boost 提供了一种声明允许某些特定测试失败的方法。此语法并非在所有支持的 Boost 版本中都一致可用，因此应使用以下结构：

```
#include <unit_test_utils/unit_test_utils.h>

// 在使用较旧 boosts 的平台上，该测试将被完全排除
#ifdef HAVE_EXPECTED_FAILURES

// 声明一个测试用例有 1 个“允许的”失败（在本例中为 2 个）
BOOST_AUTO_TEST_CASE( SomeTest, *boost::unit_test::expected_failures( 1 )
)
{
    BOOST_CHECK_EQUAL( 1, 1 );

    // 此检查失败，但不会导致测试套件失败
    BOOST_CHECK_EQUAL( 1, 2 );

    // 进一步的失败 *会不会* 是测试套装失败
}

#endif
```

运行时，这会产生类似以下内容的输出：

```
qa/common/test_mytest.cpp(123): error: in "MyTests/SomeTest": check 1 ==
2 has failed [1 != 2
*** No errors detected
```

并且单元测试可执行文件返回 `0`（成功）。

检查失败的测试是一种严格意义上的临时性情况，用于说明在修复错误之前是否触发了该错误。这不仅从“项目历史”的角度来看是有利的，而且还可以确保您为捕获所讨论的 bug 而编写的测试实际上首先捕获了 bug。

## 断言

可以在单元测试中检查断言。 在运行单元测试时，会捕获 `wxASSERT` 调用并将其作为异常重新抛出。 然后，您可以使用 `CHECK_WX_ASSERT` 宏来检查这在调试版本中是否被调用。 在发布版本中，不会运行检查，因为在这些版本中禁用了 `wxASSERT` 。

您可以使用它来确保代码正确地拒绝无效输入。

# Python 模块

Pcbnew Python 模块在 `qa` 目录下有一些测试程序， 您必须在 CMake 中打开 `KICAD_SCRIPTING_MODULES` 选项， 才能编译模块并启用该目标。

主测试脚本为 `qa/test.py` ，测试单元在 `qa/testcases` 中。 所有测试单元都可以通过运行 `test.py` 的 `ctest python` 来运行。

您也可以手动运行单个案例，方法是确保编译模块， 将其添加到 `PYTHONPATH` ，然后从源码工作区运行测试：

```
make pcbnew_python_module
export PYTHONPATH=/path/to/kicad/build/pcbnew
cd /path/to/kicad/source/qa
python2 testcase/test_001_pcb_load.py
```

## 诊断分段故障

虽然该模块是 Python，但它链接到 C++ 库 (与 KiCad Pcbnew 使用的库相同)， 因此如果库有缺陷，它可以分段错误。

您可以在 GDB 中运行测试来跟踪以下内容：

```
$ gdb

(gdb) file python2
(gdb) run testcases/test_001_pcb_load.py
```

如果测试分段失败，您将得到熟悉的断点， 就像在 GDB 下运行 pcbnew 一样。

# 实用程序

KiCad 包括一些实用程序，可用于调试、剖析、分析或开发代码的某些部分， 而不必调用完整的 GUI 程序。

通常，它们是 `qa_*_tools` QA 可执行文件的一部分，每个可执行文件都包含该库的相关工具。 要列出给定程序中的工具，请传递 `-l` 参数。大多数工具都提供了对 `-h` 参数的帮助。 要调用程序，请执行以下操作：

```
qa_<lib>_tools <tool name> [-h] [tool arguments]
```

下面是一些可用工具的简要概述。 完整信息和命令行参数请参考工具使用测试 (`-h`)。

- `common_tools` (通用库和核心函数)：
- `coroutine` ：一个简单的协同例程示例
- `io_benchmark` ：显示使用各种 IO 技术读取文件的相对速度。
- `qa_pcbnew_tools` (pcbnew 相关函数)：
- `drc`：在用户提供的 `.kicad_pcb` 文件上运行某些DRC函数并对其进行基准测试
- `pcb_parser` ：解析用户提供的 `.kicad_pcb` 文件
- `polygon_generator` ：将 PCB 上发现的多边形转储到控制台
- `polygon_triangulation` ：对 PCB 上的区域多边形执行三角剖分

# 模糊测试

可以在 KiCad 的某些部分运行模糊测试。 要对泛型函数执行此操作，您需要能够将来自模糊测试工具的某种输入传递给被测函数。

例如，要使用 AFL 模糊工具，您需要：

- 一个测试可执行文件，它可以：
- 接收 `stdin` 的输入，由 `afl-fuzz` 运行。
- 可选：处理来自文件名的输入，以允许 `afl-tmin` 最小化输入文件。

- 使用 AFL 编译器编译此可执行文件，以启用允许模糊器检测模糊状态的指令插入。

例如，`qa_pcbnew_tools` 可执行文件（包含用于 `.kicad_pcb` 文件解析的模糊测试工具 `pcb_parser` ）可以这样编译：

```
mkdir build
cd build
cmake -DCMAKE_CXX_COMPILER=/usr/bin/afl-clang-fast++ -
DCMAKE_C_COMPILER=/usr/bin/afl-clang-fast ../kicad_src
make qa_pcbnew_tools
```

您可能需要在系统上禁用核心转储和 CPU 频率调节（AFL 会警告您是否应该这样做）。 例如，以 root 用户身份：

```
# echo core >/proc/sys/kernel/core_pattern
# echo performance ¦ tee cpu*/cpufreq/scaling_governor
```

要进行模糊处理，可以通过 `afl-fuzz` 运行可执行文件：

```
afl-fuzz -i fuzzin -o fuzzout -m500 qa/pcbnew_tools/qa_pcbnew_tools pcb_parser
```

在哪里：

- `-i` 是用作模糊输入 "seeds" 的文件目录。
- `-o` 是写入结果的目录（包括引起崩溃或挂起的输入）
- `-t` 是在被宣布为 “挂起” 之前允许运行的最长时间。
- `-m` 是否允许使用内存（这通常需要调整，因为 KiCad 代码往往会使用大量内存进行初始化）

然后，AFL TUI 将显示模糊进度，您可以根据需要使用挂起或引发崩溃的输入来调试代码。

# 运行时调试

KiCad 可以在运行时调试，既可以在完整的调试器 (如 GDB) 下调试， 也可以使用简单的方法 (如将调试记录到控制台) 进行调试。

# 打印调试

如果您自己编译 KiCad，只需在代码中的相关位置添加调试语句即可，例如：

```
wxLogDebug( "Value of variable: %d", my_int );
```

这会生成调试输出，只有在调试模式下编译时才能看到。

您也可以使用 `std::cout` 和 `printf` 。

请确保在提交代码时不要将这种调试留在原地。

# 打印跟踪

代码的某些部分具有可以根据 "掩码 有选择地启用的 "跟踪"，例如：

```
wxLogTrace( "TRACEMASK", "My trace, value: %d", my_int );
```

默认情况下不会打印此文件。 要显示它，请在运行 KiCad 时设置 `WXTRACE` 环境变量，以包含要启用的掩码：

```
$ WXTRACE="TRACEMASK,OTHERMASK" kicad
```

打印时，调试将以时间戳和跟踪掩码为前缀：

```
11:22:33: Trace: (TRACEMASK) My trace, value: 42
```

如果添加跟踪掩码，请将该掩码定义并记录为 `include/trace_helpers.h` 中的变量。 这会将其添加到 跟踪掩码文档中。

一些可用的掩码：

- KiCad 核心功能：

- `KICAD_KEY_EVENTS`
- `KicadScrollSettings`
- `KICAD_FIND_ITEM`
- `KICAD_FIND_REPLACE`
- `KICAD_NGSPICE`
- `KICAD_PLUGINLOADER`
- `GAL_PROFILE`
- `GAL_CACHED_CONTAINER`
- `PNS`
- `CN`
- `SCROLL_ZOOM` - 用于 GAL 中的滚轮缩放逻辑
- 插件特定 (包括 “标准” KiCad 格式):
- `3D_CACHE`
- `3D_SG`
- `3D_RESOLVER`
- `3D_PLUGIN_MANAGER`
- `KI_TRACE_CCAMERA`
- `PLUGIN_IDF`
- `PLUGIN_VRML`
- `KICAD_SCH_LEGACY_PLUGIN`
- `KICAD_GEDA_PLUGIN`
- `KICAD_PCB_PLUGIN`

# 高级配置

有一些高级配置选项，主要用于开发或测试目的。

要设置这些选项，您可以创建文件 `kicad_Advanced` 并根据需要设置密钥 (当前列表的高级配置文档)。 您应该永远不需要将这些键设置为正常使用 - 如果您这样做了，那就是一个错误。

通过高级配置系统启用的任何功能都被认为是试验性的，因此不适合生产使用。 这些功能显然没有得到支持或被认为是经过充分测试的。 对于发现的缺陷，问题仍然是受欢迎的。

# S-表达式

S-表达式 是与 XML 相同的文本流或字符串，由一系列元素组成。 每个元素要么是原子，要么是列表。 原子对应于字符串，而列表对应于 s-表达式。

下面的语法代表了我们对 s-表达式 的定义：

```
sexpr   ::= ( sx )
sx      ::= atom sxtail | sexpr sxtail | NULL
sxtail  ::= sx | NULL
atom    :: quoted | value
quoted  :: "ws_string"
value   :: nws_string
```

原子可以是带引号的字符串 (由双引号括起来的包含空格的字符串)，也可以是不需要用引号引起来的非空格字符串。

KiCad 中使用的 s-expression 语法使用两种引用/语法策略， 这两种策略是由 Specctra DSN 规范的需求和我们自己的非 specctra 需求提供的。 Specctra DSN 规范在引用方面不是很清楚，最重要的是 Freerouter 的解释， 由于希望与 Freerouter 兼容，它实际上无论如何都会取代 Specctra DSN 规范中的任何内容。

我们有自己的需求，超出了 Specctra DSN 规范的需求，因此我们支持引用原子的两种语法或引用协议：

1. Specctra 引用协议 (specctraMode)
2. KiCad 引用协议 (non-specctraMode)

通过为非 Specctra DSN 文件单独定义模式，我们可以更好地控制自己的命运。

总而言之，在 specctraMode 中，Freerouter 告诉我们需要做什么。 在可以被认为是 KiCad 模式的非 specctraMode 中，我们有自己的引用协议， 可以在不破坏 specctraMode 的情况下进行更改。

在任一模式下，如何保存文件和如何读回文件之间需要达成一致，以满足往返要求。 使用一种模式写入的文件不一定可以使用另一种模式读取，尽管它可能是可读的。 只是别指望它了。

在 KiCad 模式下：

OUTPUTFORMATTER::Quoted() 是包装 s表达式 原子的工具。 DSNLEXER::NexTok() 基本上是逆函数，它读回令牌。 这两个必须达成一致，这样写出来的东西才能原封不动地送回来。

是否对字符串进行换行的决定留给了 `Quoted()` 函数。 如果字符串被换行，它还将转义内部`双引号`、`\n` 和 `\r`。 任何空字符串都用引号括起来，任何以 '#' 开头的字符串都用引号括起来，这样就不会与 s表达式 注释混淆。

# KiCad S-表达式 语法和引用协议 (非 specctraMode)：

1. 有些原子被认为是关键字，它们构成了叠加在 s-表达式 上的语法。 所有关键字都是 ASCII 和小写。 这里不使用国际字符。
2. 所有 KiCad S-表达式 文件都使用 UTF8 编码保存，并且应该支持原子中任何非关键字的国际字符。
3. DSNLEXER::NextTok() 要求任何标记都在一行输入上。 如果要保存多行字符串，Quoted() 将自动转义 \n 或 \r 并将输出放在一行中。 往返应该没问题。
4. 带引号的字符串中只能有转义序列。 转义序列允许外来工具在输入流中生成字节模式。 支持 C 样式 2 字节十六进制代码，也支持 3 字节八进制转义序列。 有关转义序列的完整列表，请参见 DSNLEXER::NextTok()，方法是在 dsnlexper.cpp 文件中搜索字符串 “转义序列”。 在应用转义处理之后，任何转义机制的使用仍必须生成 UTF-8 编码的文本。
5. 仅仅因为输入上支持转义序列，并不意味着 OUTPUTFORMATTER::Quoted() 必须为输出生成这样的转义序列。 例如，在 S-表达式 文件中有真的制表符是可以的。 因此，这将不会在输出时被转义。 还有其他类似的案例。
6. 反斜杠是转义字节。

# PYTHON 插件

## 页面

- PcbNew 插件

  *KiCad 实现了一个 Python 插件接口，因此外部 Python 插件可以从 Pcbnew 内部运行。 使用 简化的包装器和接口生成器 或 SWIG 生成接口。 SWIG 被指令使用 interface 文件将特定的 C/C 头文件翻译成其他语言。 这些文件最终决定导出哪些 C/C 函数、类和其他声明， 这些声明可以在 pcbnew/swg/ 中找到。 在构建时，SWIG 界面文件用于生成相应的 .py 文件。 这些文件被安装到 Python 的系统级 dist-Packages 库中， 因此它们可以由系统上安装的任何 Python2 解释器导入。 现有 Pcbnew Python API 文档 Pcbnew Python API 可以独立使用，即没有 Pcbnew 实例在运行， 要操作的电路板工程被加载并保存到文件中。 此方法通过 用户文档 中的一些示例进行了说明。 另一个文档源是 API 的自动生成的 DOORT 参考。 它可以在 这里 找到。 "动作插件" 支持 除了独立使用生成的 Python 插件界面外，Pcbnew 还提供了关于在线操作电路板工程的额外支持。 使用此功能的插件称为 动作插件，可以使用 Pcbnew 菜单项访问它们，该菜单项可以在 工具 → 外部插件 下找到。 遵循 动作插件 约定的 KiCad 插件可以在该菜单中显示为外部插件，也可以选择显示为顶部工具栏按钮。 用户可以运行该插件，从而调用 Python 插件代码中定义的 Entry 函数。 然后可以使用此函数从 Python 脚本环境访问和操作当前加载的电路板。*

# PCBNEW 插件

KiCad 实现了一个 Python 插件接口，因此外部 Python 插件可以从 Pcbnew 内部运行。 使用 简化的包装器和接口生成器 或 SWIG 生成接口。 SWIG 被指令使用 `interface` 文件将特定的 C/C 头文件翻译成其他语言。 这些文件最终决定导出哪些 C/C 函数、类和其他声明， 这些声明可以在 **pcbnew/swg/** 中找到。

在构建时，SWIG 界面文件用于生成相应的 .py 文件。 这些文件被安装到 Python 的系统级 **dist-Packages** 库中， 因此它们可以由系统上安装的任何 Python2 解释器导入。

# 现有 Pcbnew Python API 文档

Pcbnew Python API 可以独立使用，即没有 Pcbnew 实例在运行， 要操作的电路板工程被加载并保存到文件中。 此方法通过 用户文档 中的一些示例进行了说明。

另一个文档源是 API 的自动生成的 DOORT 参考。 它可以在 这里 找到。

# "动作插件" 支持

除了独立使用生成的 Python 插件界面外，Pcbnew 还提供了关于在线操作电路板工程的额外支持。 使用此功能的插件称为 **动作插件**，可以使用 Pcbnew 菜单项访问它们，该菜单项可以在 **工具 → 外部插件** 下找到。 遵循 **动作插件** 约定的 KiCad 插件可以在该菜单中显示为外部插件，也可以选择显示为顶部工具栏按钮。 用户可以运行该插件，从而调用 Python 插件代码中定义的 Entry 函数。 然后可以使用此函数从 Python 脚本环境访问和操作当前加载的电路板。

## 典型的插件结构

**动作插件** 支持在 Pcbnew 中通过在启动时发现特定目录中的 Python 包和 Python 脚本文件来实现。 要使发现过程正常工作，必须满足以下要求。

- 插件必须安装在 KiCad 插件搜索路径中，如 `scriting/kicadplugins.i` 中所述。 您始终可以通过打开脚本控制台并输入以下命令来查找设置的搜索路径：

`import pcbnew; print pcbnew.PLUGIN_DIRECTORIES_SEARCH`

当前在 Linux 安装上，插件搜索路径为

- `/usr/share/kicad/scripting/plugins/`
- `~/.kicad/scripting/plugins`
- `~/.kicad_plugins/`

在 Windows 上

- `%KICAD_INSTALL_PATH%/share/kicad/scripting/plugins`
- `%APPDATA%/Roaming/kicad/scripting/plugins`

在 macOS 上，有一个安全特性，可以更容易地将脚本插件添加到 ~/Library... 路径比到 kicad.app 方便， 但搜索路径是

- `/Applications/kicad/Kicad/Contents/SharedSupport/scripting/plugins`
- `~/Library/Application Support/kicad/scripting/plugins`
- 或者，可以在 KiCad 插件路径链接中创建指向文件系统其他位置的插件文件/文件夹的符号链接。这对开发很有用。
- 插件必须编写为位于插件搜索路径中的简单 Python 脚本 (*.py)。 请注意，这种方法是由单个 .py 文件组成的小型插件的首选方法。
- 或者，必须将插件实现为符合 Python 包标准定义的 Python 包 (请参见 6.4.包)。 请注意，此方法是由多个 .py 文件和资源文件 (如对话框或图像) 组成的较大插件项目的首选方法。
- Python 插件必须包含一个派生自 `pcbnew.ActionPlugin` 的类，并且它的 `register()` 方法必须在插件内调用。

以下示例演示了插件要求。

# 简单插件示例

这个简单插件的文件夹结构相当简单。 单个 Python 脚本文件放置在 KiCad 插件路径中的目录中。

```
+ ~/.kicad_plugins/ # KiCad 插件路径中的文件夹
    - simple_plugin.py
    - simple_plugin.png (可选)
```

`simple_plugin.py` 文件包含以下内容。

PYT

```
import pcbnew
import os

class SimplePlugin(pcbnew.ActionPlugin):
    def defaults(self):
        self.name = "插件名称，如 Pcbnew：Tools->External Plugins 中所示"
        self.category = "描述性类别名称"
        self.description = "对插件及其功能的描述"
        self.show_toolbar_button = False # 可选，默认为 False
        self.icon_file_name = os.path.join(os.path.dirname(__file__),
'simple_plugin.png') # 可选，默认为 ""

    def Run(self):
        # 在用户操作时执行的插件的入口函数
        print("Hello World")

SimplePlugin().register() # 实例化并注册到 Pcbnew
```

请注意，如果指定的 `icon_file_name` 必须包含插件图标的绝对路径。 必须是 PNG 文件，建议大小为 26x26 像素。 支持不透明度的 Alpha 通道。 如果未指定图标，将使用通用工具图标

`show_toolbar_button` 只定义了插件工具栏按钮的默认状态。 用户可以在 pcbnew 首选项中覆盖它。

# 复杂插件示例

复杂插件示例表示在 Pcbnew 启动时导入的单个 Python 包。 在导入 Python 包时，会执行 `__init__.py` 文件， 因此非常适合实例化插件并将其注册到 Pcbnew。 这里最大的优势是，您可以更好地模块化您的插件并包含其他文件， 而不会使 KiCad 插件目录变得杂乱无章。 此外，可以使用 `python -m` 独立执行相同的插件。 例如对 Python 代码执行测试。 以下文件夹结构显示了复杂插件的实现方式：

```
+ ~/.kicad_plugins/ # 此目录必须位于插件路径中
    + complex_plugin/ # 插件目录 (Python 包)
        - __init__.py # 此文件在导入软件包时执行 (在 Pcbnew 启动时)
        - __main__.py # 此文件是可选的。见下文
        - complex_plugin_action.py # ActionPlugin 派生类位于此处
        - complex_plugin_utils.py # 插件的其他 Python 部分
        - icon.png
        + otherstuff/
            - otherfile.png
            - misc.txt
```

建议将包含 ActionPlugin 派生类的文件命名为 `<package-name>_action.py` 。 在本例中，文件名为 `complex_plugin_action.py` ，内容如下：

PYT

```
import pcbnew
import os

class ComplexPluginAction(pcbnew.ActionPlugin)
    def defaults(self):
        self.name = "一个复杂的动作插件"
        self.category = "描述性类别名称"
        self.description "该插件的说明"
        self.show_toolbar_button = True # 可选，默认为 False
        self.icon_file_name = os.path.join(os.path.dirname(__file__),
'icon.png') # 可选

    def Run(self):
        # 在用户操作时执行的插件的入口函数
        print("Hello World")
```

然后使用 `init.py` 文件实例化插件并将其注册到 Pcbnew，如下所示。

PYT

```
from .complex_plugin_action import ComplexPluginAction # 注意相对的导入！
ComplexPluginAction().register() # 实例化并注册到 Pcbnew
```

正如 Python PEP 338 中所述可以将包 (或模块) 作为脚本执行。 这对于以最小的工作量实现 KiCad 插件的命令行独立版本非常有用。 为了实现这一功能，我们在包目录下创建了一个 `main.py` 文件。 可以通过运行以下命令来执行此文件。

```
python -m <package_name>
```

运行该命令时，请确保您的当前目录是软件包目录的父目录。 在这些示例中，这将是 `~/.kicad_plugins` 。 运行该命令时，Python 解释器会运行 `/complex_plugin/*init*.py` 后跟 `/complex_plugin/*main*.py` .

# 源代码文档

正在重定向至 Doxygen...

如果未发生重定向，您可以 单击这里

# 贡献

# 为 KiCad (应用程序) 开发做出贡献

KiCad 一直是工程师在有限的时间内进行开发的社区驱动的努力。 我们欢迎任何人贡献错误修复、改进和新功能。

## 开发者邮件列表

做任何事之前要做的第一件事就是加入。 KiCad 开发人员邮件列表 在这里，您可以提出广泛的问题，并提出您的想法或计划，如果它超出了错误修复。

## 获取代码

KiCad 使用 Git 用于源代码控制。

如果您是 git 新手，强烈建议您。 查找并遵循在线提供的许多教程，如。 这一个 和/或阅读 git 文档 。

我们还致力于使用 GitLab 作为我们 Git 仓库的主要托管平台。

所有仓库都可以在 这里 找到。

主应用程序的仓库可以在 这里 找到

## 编译代码

按照您的平台对应的 文档 中的说明设置工作构建环境， 并从源代码成功构建 KiCad。

## 文档代码

要熟悉代码库，您可以阅读 Doxygen 生成的文档。 提交新代码时，请记住更新文档说明。 您可以运行 `make doxygen-docs` 在本地生成文档。

Jenkins 服务器上也提供了相同文档的最新版本， 请参阅 C++ API 的 KiCad 开发人员文档。

您还可以在 Doxygen 文档中找到各种开发人员备注， 请参见 这个相关页面。

# 提交之前

进行更改后，请确保您的代码符合以下 KiCad 编码样式策略。

您的提交消息应遵循说明的规则中 KiCad 提交消息格式策略

如果您想在用户界面上工作，您会发现阅读 用户界面指南。

让所有开发人员都能读懂代码库对我们来说很重要。 除非遵守这些政策中规定的规则，否则不会接受您的补丁程序。

编程是一个迭代的过程，如果您必须回去更改您的补丁，请不要担心，我们都是这样做的。

# 提交代码

所有补丁都需要在 GitLab 上以 **合并请求** 的形式提交。 有关如何创建合并请求的信息，请参阅此 如何为新用户创建合并请求。

**注意：KiCad 有一个 GitHub 镜像，但所有拉取请求都会被忽略，我们只接受 GitLab 上的更改**

# 修复所有 CI 问题

GitLab 仓库启用了持续集成 (CI)。这意味着它会自动对传入的提交运行处理步骤，以执行从确保代码可以编译到编码样式策略等几个任务。

您可以在合并请求页面上找到配置项作业的状态并查看其输出。如果您需要帮助，您可以在合并请求中要求开发人员评论如何解决问题。

# 初学者 / 初学者补丁

大体上没有什么想法，但想帮忙吗?

你可能需要调查一下 标签为 ‘Starter’ 的问题。

例如，如果您是 macOS 用户，您可能需要检查以下内容： 标记为 'macOS' 的问题。

# 互联网中继聊天(*IRC*)

欢迎在 #kicad@freenode. 上加入 irc 频道。 那里有一群不错的人在闲聊，所以如果你有任何问题，不知道去哪里问，你应该试着在这里问。 在各种时区都有各种各样的人，既有开发 KiCad 的人，也有普通的热心用户。