

高级语言程序设计实验报告

南开大学 计算机学院 田佳业 2013599 1013班

2023年5月15日

作业题目

代码编辑器QLion

开发软件

Qt 6.4.2

CLion 2023.1.2

Qt Designer

课题要求

- 采用C++语言编写
- 采用面向对象的设计理念

项目计划完成情况

☒ CLion界面风格

☒ 文本编辑器基本功能

☒ 代码高亮

☒ 文件目录树

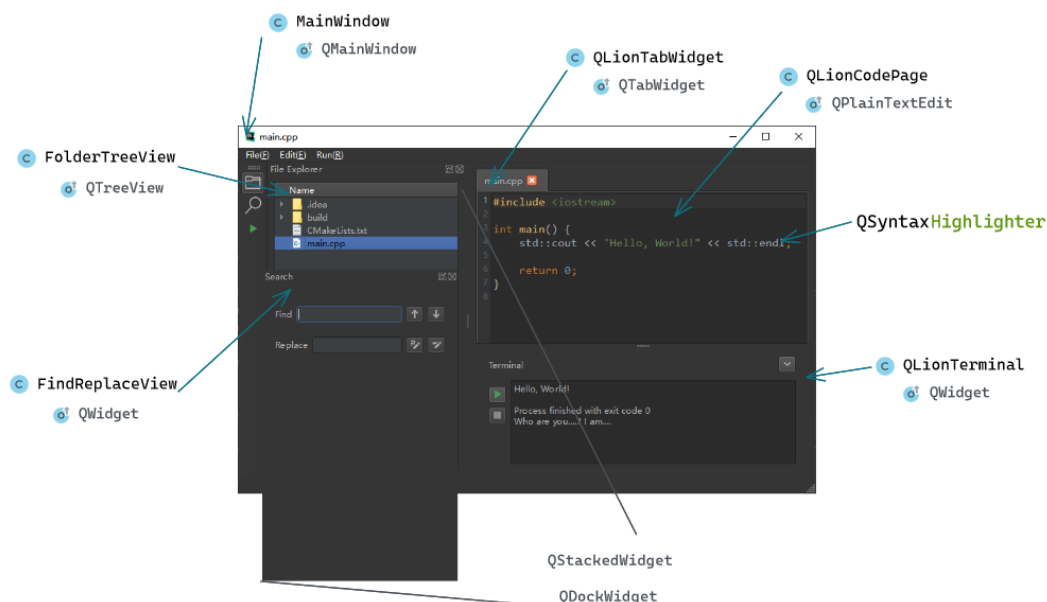
☒ 查找替换

☒ 快速注释

☒ Cmake项目运行

□ 主题及快捷键配置

项目结构



主要流程

由于整个项目较为复杂，下面仅简要介绍重要部分的实现思路，有一些实现细节可能难免不能面面俱到。

界面风格

使用QT提供 **fusion** 风格，并结合使用 **platte** 和 **stylesheet** 完成暗黑风格的绘制。在此代码片段基础上进行了微调。

文本编辑器

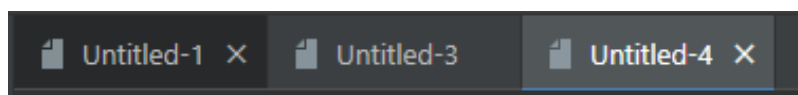
文本编辑器部分主要的难点在于tab的有效管理。

添加标签页

添加标签页有两种情况，一种是新建的没有与文件关联的标签页，另一种是通过文件打开的标签页。

这一部分的主要亮点：

- 实现了类似VSCode中打开同名但不同路径的文件时，可以自动通过显示文件路径来区分来自不同路径的文件。
- 注意到VSCode新建若干个未与文件关联的标签页时，会选取当前可用的最小标签页。如下图所示新建标签页会命名为 `Untitled-2`。在这个项目中也复现了这样的设计。



第一点的实现可以逐个标签页进行遍历，也可以采用字典树等方式进行优化。在这个项目中采用了较容易实现的遍历方式。

第二点的实现使用了 `unordered_set` 存储了当前存在的未命名标签页，逐ID比对。

为了便于通过路径快速寻找到对应的标签页，可以将 `<filePath, tabIndex>` 的对应关系存到一个 `unordered_map` 中。哈希表可以实现近似 `O(1)` 的查找替换的复杂度。

下面是添加标签页的部分代码，有两个重载函数，分别对应了是否关联文件的标签页。

关联文件的标签页：

```
void QLionTabWidget::addNewTab(const QString &text, const QString
&filePath) {
    if (mainWindow) {
        bool needToDistinguish = false;
        QString fileName = QFileInfo(filePath).fileName();
        for (int i = 0; i < count(); i++) {
            QString filePathOfCurrentTab = getCodePage(i)-
>getFilePath();
            if (filePathOfCurrentTab == filePath) {
```

```

        setCurrentIndex(i);
        return;
    } else if (QFileInfo(filePathOfCurrentTab).fileName()
== fileName) {
        setTabText(i, filePathOfCurrentTab);
        needToDistinguish = true;
    }
}
if (needToDistinguish) {
    fileName = filePath;
}
// if the text is too long, do not init Highlighter
if (text.length() > 10000) {
    addTab(new QLionCodePage(this, false), fileName);
} else {
    addTab(new QLionCodePage(this), fileName);
}
usingFilePath[filePath] = count() - 1;
setCurrentIndex(count() - 1);
auto *codePage = getCurrentCodePage();
// do not forget to set the parentTabWidget
getCurrentCodePage()->setParentTabWidget(this);
codePage->setFilePath(filePath);
codePage->setPlainText(text);
}
}

```

由于文字较长时代码高亮会有卡顿，因此当大于一定长度的时候取消高亮。现在的编辑器如VSCode也是这么做的。

新建的标签页

```

void QLionTabWidget::addNewTab() {
    if (mainWindow) {
        // get the minimum unused untitledID from the set
        int newID = 1;
        // it will iterate at most size() times
        while (usingUntitledID.count(newID)) {

```

```

        newID++;
    }
    usingUntitledID.insert(newID);
    QString title = "Untitled-" + QString::number(newID);
    addTab(new QLionCodePage(this), title);
    setCurrentIndex(count() - 1);
    QLionCodePage *currentCodePage = getCurrentCodePage();
    currentCodePage->setParentTabWidget(this);
    currentCodePage->setUntitledID(newID);
}
}

```

关闭标签页

关闭标签页时，若标签页未保存，需要提示是否进行保存。同时如果没有标签页，应当把菜单栏上复制粘贴等按钮禁用掉，也不应当进行查找替换等。

行号事件响应

点击事件

VSCode中点击行号可以跳转到对应行号，选中该行文本并将光标置于下一行。对此进行实现：

```

void mousePressEvent(QMouseEvent *event) override{
    codeEditor->lineNumberAreaMouseEvent(event);
}

```

```

void QLionCodePage::lineNumberAreaMouseEvent(QMouseEvent
*mEvent) {
    // select the current line and jump the cursor to the
    beginning of the next line
    int clickedLineNumber=qRound(mEvent->
position().y())/fontMetrics().height()+verticalScrollBar()-
>value();
    //      qDebug()<<clickedLineNumber;
    QTextBlock clickedBlock=document()-
>findBlockByLineNumber(clickedLineNumber);
    QTextCursor cursor(clickedBlock);

    cursor.movePosition(QTextCursor::QTextCursor::NextBlock,QTextCurs
or::KeepAnchor);
    setTextCursor(cursor);
}

```

滚动事件

和TextEdit部分保持同步即可。

绘制事件

需要获取当前可见区域的第一个Block获取其行号。Block按照链表组织，一直next获取下一个Block直到看不见为止，绘制可见区域数字。绘制宽度和位置需要根据字体和数字位数动态调整。参见代码中 `lineNumberAreaPaintEvent` 部分。

代码高亮

代码高亮使用了QT提供的 `QSyntaxHighlighter` 利用正则表达式进行高亮。由于C++不是LR(1) 文法，必然不能使用正则表达式进行准确的高亮。但是可以使用一些trick让高亮尽可能的准确。

这一部分需要注意的主要是：

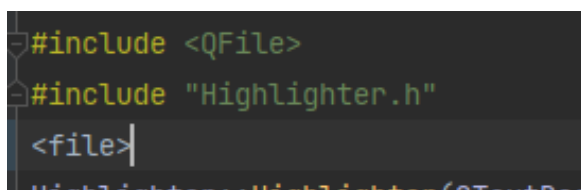
- 后添加的规则的高亮会覆盖掉先添加的规则
- 完全可以部分高亮匹配的文本，也可以为不同的部分添加不同的高亮颜色

基于第一条，我们必须选择合理的顺序渲染；同时还可以弥补一些渲染上的缺陷。比如浮点数的渲染。搜索尝试了多种正则表达形式，最后采用了编译原理中编写了一条几乎适用所有浮点形式的正则表达式：

```
((([0-9]*[.][0-9]*([eE][+-]?[0-9]+)?)|([0-9]+[eE][+-]?[0-9]+))  
[fLlL]?)
```

它可以匹配 `.xxx` 和 `xxx.` 形式的浮点数，且不与变量名冲突(这也是为什么编译原理词法分析选择用这个表达式)，还能匹配科学计数法和带显式浮点格式的浮点数。但是唯一的缺陷是会把单独的点高亮。而通常在作为类成员访问符时是不高亮的。类成员的高亮规则刚好解决了这个问题。我们认为点后面加变量名这种方式作为类成员的高亮定义。又根据第二条，我们可单独为此时的点设置不同的颜色。把这条规则放到浮点数后面，就能完美的解决点高亮的问题。

第二条的其他应用比如我们可以将头文件定义作为一条规则来匹配。CLion中高亮是这样：



```
#include <QFile>  
#include "Highlighter.h"  
<file>  
Highlighter::Highlighter(QTextDoc
```

单独的尖括号包裹的字符串不会被高亮，只有与 `#include` 配合使用时才会高亮。这只有通过单一匹配规则和部分高亮实现：

```
void Highlighter::addIncludeFormat(const QString &text) {  
    HighlightRule rule;  
    rule.pattern = QRegularExpression(R"(#include\s*[<"])[a-zA-Z0-9_./\\"*]*[>"]");  
    QColor stringColor(106, 135, 89);  
    QColor includeColor(255, 198, 109);  
    rule.format.setForeground(stringColor);  
    rule.format.setFont(QFont(mFontFamily, mFontSize));  
    QRegularExpressionMatchIterator matchIterator =  
    rule.pattern.globalMatch(text);  
    while (matchIterator.hasNext()) {  
        QRegularExpressionMatch match = matchIterator.next();
```

```

        // set the "#include" to includeColor and others to
stringColor
        setFormat(match.capturedStart(), 8, includeColor);
        setFormat(match.capturedStart() + 8,
match.capturedLength() - 8, rule.format);
    }
}

```

由于时间原因没做主题配置功能，因此颜色暂时硬编码到了代码中。

另外，不同于单行的高亮内容。若支持高亮注释，需要记录每一行的状态。QT提供了 `setCurrentBlockState()` 函数供我们记录行的信息。对每一行来说，如果上一行是注释，跳过检测前面的 `/*`。代码实现如下：

```

//notice: it was called line by line
void Highlighter::addMultiLineCommentFormat(const QString &text) {
    //mark the start of the comment
    setCurrentBlockState(0);
    QRegularExpression startExpression(R"(/\*)");
    QRegularExpression endExpression(R"(\*/)");
    QColor color(128, 128, 128);
    QTextCharFormat multiLineCommentFormat;
    multiLineCommentFormat.setForeground(color);
    multiLineCommentFormat.setFont(QFont(mFontFamily, mFontSize));
    long long startIndex = 0;
    // that is, if the previous line is not a comment
    if (previousBlockState() != 1)
        startIndex = startExpression.match(text).capturedStart();
    //if the previous line is a comment, we should start from the
beginning of the line (startIndex=0)
    while (startIndex >= 0) {
        QRegularExpressionMatch endMatch =
endExpression.match(text, startIndex);
        long long endIndex = endMatch.capturedStart();
        long long commentLength = 0;
        if (endIndex == -1) {
            // we still in a comment
            setCurrentBlockState(1);

```

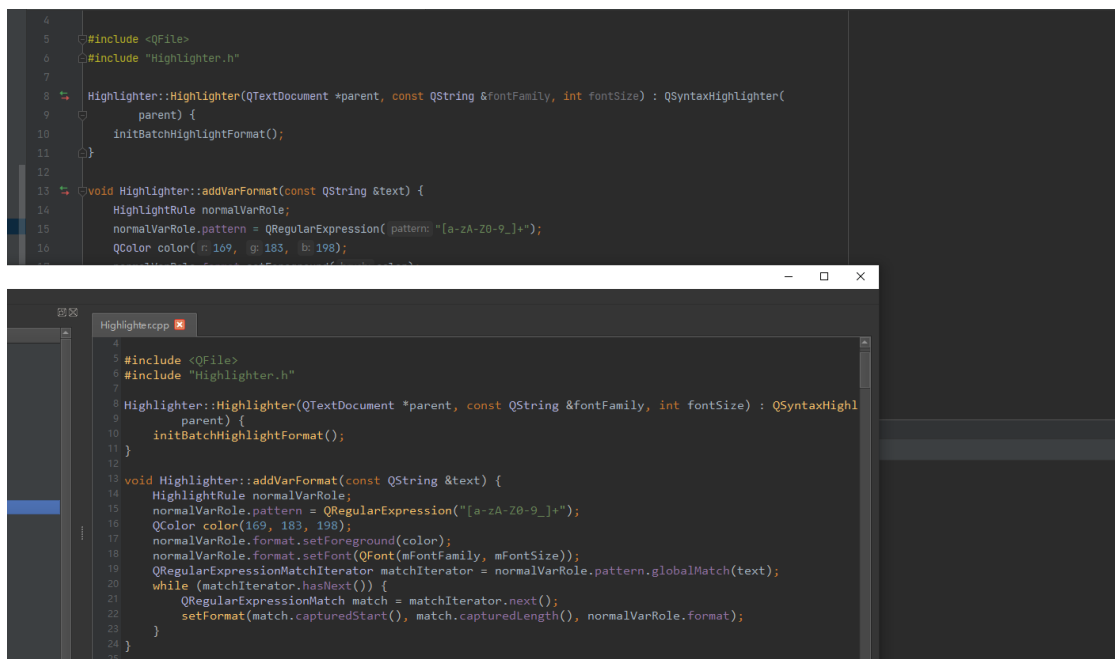


```

        commentLength = text.length() - startIndex;
    } else {
        // we find the end of the comment
        commentLength = endIndex - startIndex +
endMatch.capturedLength();
    }
    setFormat(startIndex, commentLength,
multilineCommentFormat);
    startIndex = startExpression.match(text, startIndex +
commentLength).capturedStart();
}
}

```

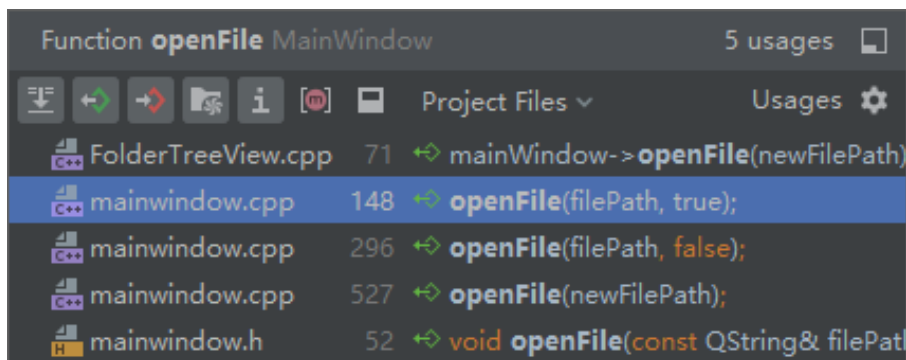
可以看到几乎还原了在真实代码编辑器中的显示效果。



菜单栏功能

打开文件

打开文件是文本编辑器的基本操作。需要判断文件是否有效，保存打开路径以便下次从这个目录再寻找文件，还要判断是否需要在标签栏上区分路径。得益于面向对象和封装的思想，`mainwindow` 提供打开文件的接口后，除了通过菜单栏打开文件，后续通过文件目录树打开文件，通过拖拽打开文件等都可以直接调用接口，而不必再关心怎样区分标签栏路径这样的细节。



保存文件

在项目的设计中将保存文件交给了 `QLionCodePage`，即让打开的标签页自己处理保存事件。当新建的文件保存时会产生文件路径，需要更新对应的数据结构，并区分标签页。同样的，后续运行项目，关闭标签页提示等需要用到保存操作时，不必关心实现的细节。

编辑操作

直接交由对应标签页处理即可。注意到若光标没有选中文本，复制会直接复制一行。

查找替换操作

这一部分是查找替换功能实现后写的，获取选中文本，跳转到对应页面设置文本即可。

文件目录树

这一部分采用了QT中模型-视图结构。通过 `QTreeView` 和 `QFileSystemModel` 结合使用可以实现显示文件目录。`Treeview` 上 `mouseReleaseEvent` 操作可以实现对文件的创建，重命名，删除操作。

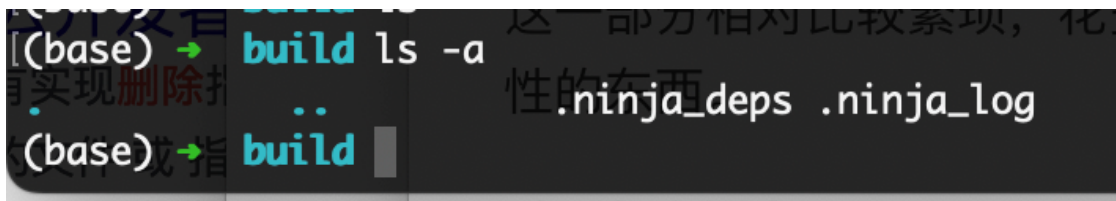
需要注意，如果点击的 `idx` 无效，说明点的是空白的地方，并不是什么都不做，而是在根目录进行操作。

新建和重命名操作都是新弹出一个Widget，设置好出现的尺寸，相应回车操作。

重命名操作是其中最复杂的操作，经过多次测试才确保无误。重命名后，需要根据文件路径查找是否该文件已经在tabWidget上显示，如果显示了需要更新其路径。当然也会出现名字冲突的现象，代码中已经将其抽象为 `distinguishFileName`。

拖拽操作需要重载三个 `event`，其中 `dropEvent` 需要将文件添加到正确的位置，其他的只需要判断拖拽的是不是文件即可。

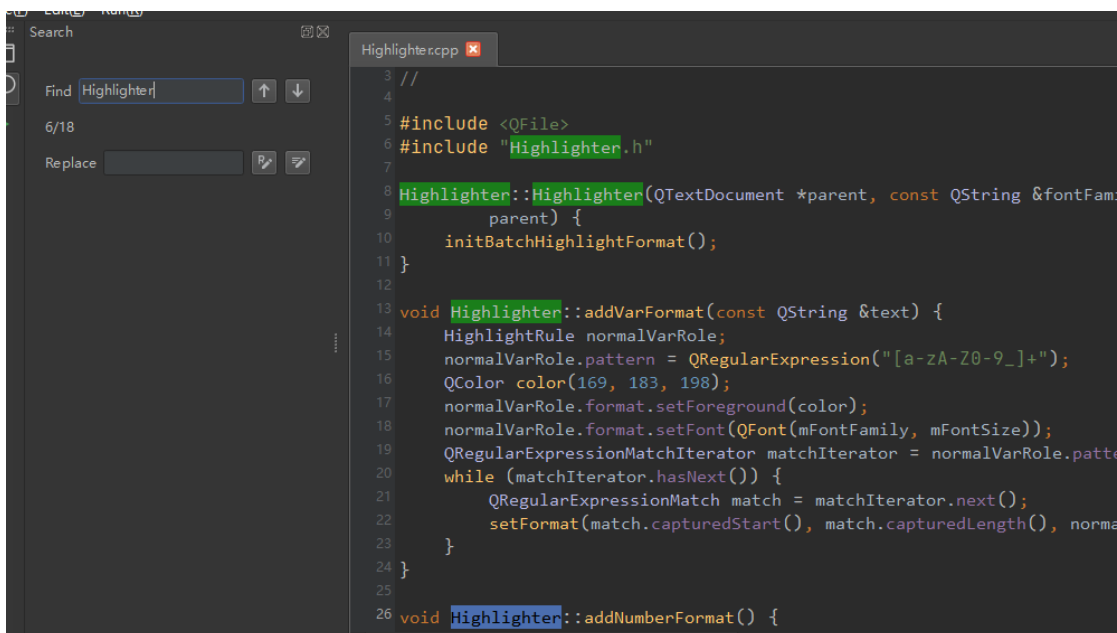
上述操作目录相关的操作需要对其下的文件进行递归操作，一开始看书只看到有 `rmdir` 这个接口，自己造了轮子结果演示的时候发现 `build` 目录本身删不掉。后续测试发现其他目录都是可以的。为什么呢？阅读文档发现删除目录需要目录非空。`build` 看上去是空的，但是获取文件列表的时候隐藏文件获取不到。实际上它就不是空的：



后来发现有 `removeRecursively`，这个是可以正常工作的。

这一部分相对比较繁琐，花费时间也较长，但主要为API调用和操作逻辑为主，没有什么特别的技巧性的东西。

查找替换



首先比较坑的是需要处理好切换逻辑。可以使用 `QActionList` 完成互斥动作处理，不过由于只有两个动作，暂时直接在代码里写切换逻辑也不是不行。

查找替换的高亮(和文本选中)操作也需要更新。切换标签页时需要在新的标签栏高亮，取消旧标签栏高亮，切换到文件目录树也是如此。这需要为标签页切换额外增加槽函数。但信号的 `index` 是切换后的 `index`，我们需要增加一变量保存切换前的 `index`。但这个切换前的 `index` 使用时需要判断其有效性，比如关闭标签页时可能会导致保存的这个 `index` 不可用，造成内存泄漏。

高亮所有匹配目标仍旧是采用的 `Highlighter`。为其动态额外增加高亮目标即可。不想高亮了将其设为空字符串。但是由于高亮器是按行高亮的，为关键字设置背景色时与 `plaintextEdit` 为当前行添加 `extraSelection` 可能有冲突。目前没找到好的解决方案，不过这个问题也不影响正常使用。

注意到替换文本可能会为查找位置带来偏移，一开始保存的查找到的位置可能会失效。如果 `Highlighter` 提供某个正则表达式第 `n` 个匹配位置之类的接口，这就不需要我们太担心这个问题，只需要利用 `highlighter` 实时获取位置即可。理论上现在的编辑器支持查找过程中改变文本，也是采用的实时正则匹配。但是一开始没有考虑到偏移问题，采用的是第一次查询把所有位置保存下来，后面查找直接从向量中取位置出来即可。后面发现这是一种很蠢的方案。毕竟，这些位置是编辑敏感的，编辑或替换文本后位置就变了。因此后续引入了偏移，以在替换时根据查找词长度更新位置。但是这还是没解决编辑的问题。因此查找替换时冻结了编辑页面。这样可以经过简单的偏移计算保证位置准确性。不过由于每次替换都需要更新后续所有的位置，复杂度还是很高。后来意识到没有有效利用高亮器提供的匹配功能本身就带有长度和位置信息。但是由于时间有限，这一部分并没有进行比较优雅的设计。后续再去进行调整。比较合理的思路是高亮器获取当前查找的 `index`，高亮的过程中记录位置和匹配个数，分别发送信号给 `codePage` 和 `mainWindow` (再传给 `FindReplaceView`，这样设计的原因是有较明确的主从关系，而非任何两个对象都能直接交流，造成较强的耦合) 去改变选中文本和查找的情况。抛开底层算法(字符串和正则匹配)的复杂性，将这一部分的业务逻辑进行合理的设计也是需要费些心思的。这也是在这个项目上面投入时间有些后悔的原因：比起业务逻辑和 API 调用，算法和系统设计才是我们最应当关注的地方。除开智商的因素，尽可能有意识的培养这一方面的直觉还是给常重要的。

快速注释

快速注释可以通过按 `ctrl` + `/` 来注释和取消注释。看上去只是在行首添加或去掉 `/` 的问题，但更重要的是需要将光标恢复到原来的位置。如果处理不好将导致光标恢复到错误的位置甚至有效范围之外，导致不可预知的行为。另外需要额外处理单个 `/` 的情况，虽然这种情况在实际编辑中并不常见。

```
void QLionCodePage::denoteCurrentLine() {
    QTextCursor cursor = textCursor();
    // record the current position
    int position = cursor.position();
    int positionInBlock = cursor.positionInBlock();
    // qDebug() << position << " " << positionInBlock;
    cursor.movePosition(QTextCursor::StartOfLine);
    QString text = cursor.block().text();
    int i;
    for(i=0;i<text.length();i++){
        if(text[i]==' '||text[i]=='\t'){
            continue;
        }
        else if(text[i]=='/'){
            if(i<text.length()-1){
                if(text[i+1]=='/') {
                    //it is a line with spaces and a double '/',
                    remove the denotation here
                    cursor.movePosition(QTextCursor::Right,
                    QTextCursor::MoveAnchor, i);
                    cursor.movePosition(QTextCursor::Right,
                    QTextCursor::KeepAnchor, 2);
                    cursor.removeSelectedText();
                    if(positionInBlock<=i){
                        // if the cursor is at the left of the
                        denotation, move the cursor to the original position
                        cursor.setPosition(position);
                    }
                    else if(positionInBlock==i+1){
```

```

        // if the cursor is at the middle of the
denotation, move the cursor to the original position with a offset
        cursor.setPosition(position-1);
    }
    else{
        // if the cursor is at the right of the
denotation
        cursor.setPosition(position-2);
    }
}
else{
    //it is a line with spaces and a single '/',
add a single '/' here
    cursor.movePosition(QTextCursor::Right,
QTextCursor::MoveAnchor, i+1);
    cursor.insertText("/");
    if(positionInBlock<=i+1){
        // if the cursor is at the right of the
denotation, move the cursor to the original position
        cursor.setPosition(position);
    }
    else{
        // if the cursor is at the left of the
denotation, move the cursor to the original position with a offset
        cursor.setPosition(position+1);
    }
}
}
else{
    //it is a line with spaces and a single '/', add a
single '/' to the end of the line
    cursor.movePosition(QTextCursor::EndOfLine);
    cursor.insertText("/");
    cursor.setPosition(position);
}
break;
}

```



```

        else{
            // not start with spaces and '/', denote at the start
of the line
            cursor.insertText(R"("//)");
            cursor.setPosition(position+2);
            break;
        }
    }
    // move the cursor to the original position
    setTextCursor(cursor);
}

```

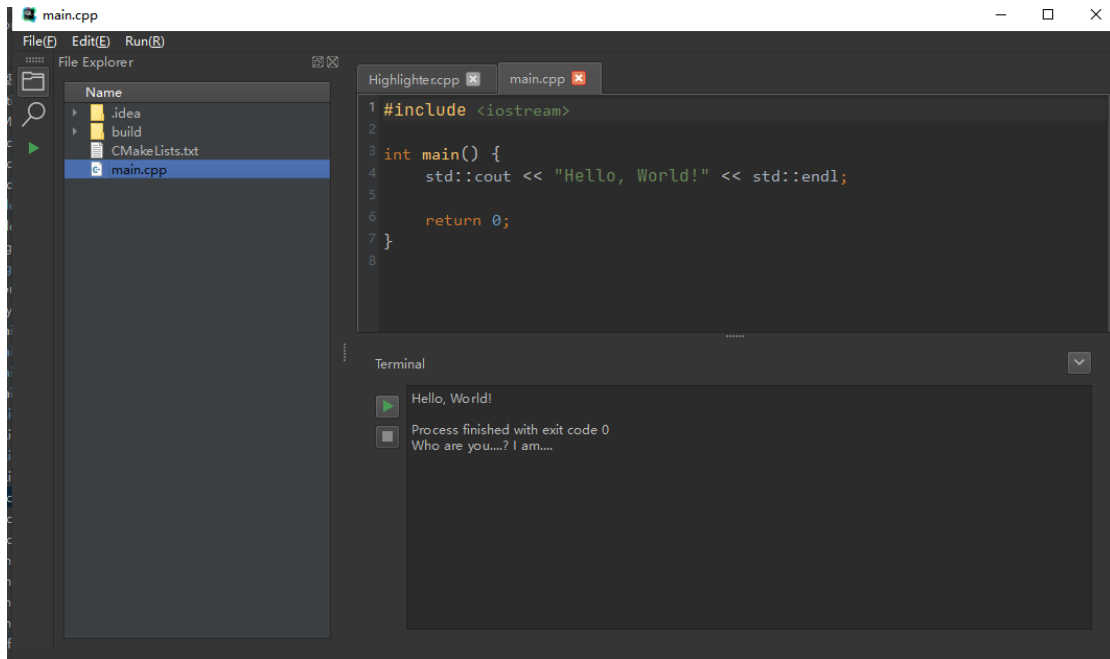
Cmake项目运行

众所周知，QT的强大之处在于跨平台。而CMake作为跨平台的构建工具，Clion为了支持跨平台也是用的CMake构建项目。我们的编辑器可以利用CMake完成平台和脚本无关的构建。我们只要设置好Cmake和构建器的路径，并交由用户自定义构建选项，CMake可以构建的项目，我们的编辑器也可以构建。

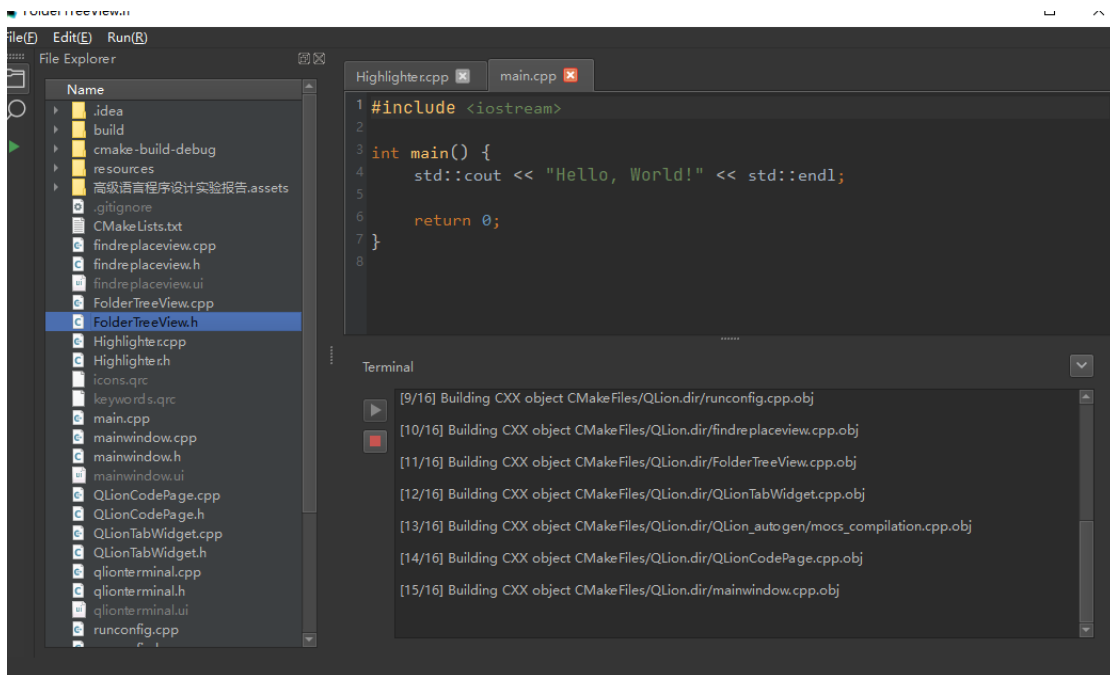
Qprocess可以实现跨平台的命令运行。代码中定义了QLionTerminal类，完成可视化终端的设计并对QProcess类的功能进行封装。

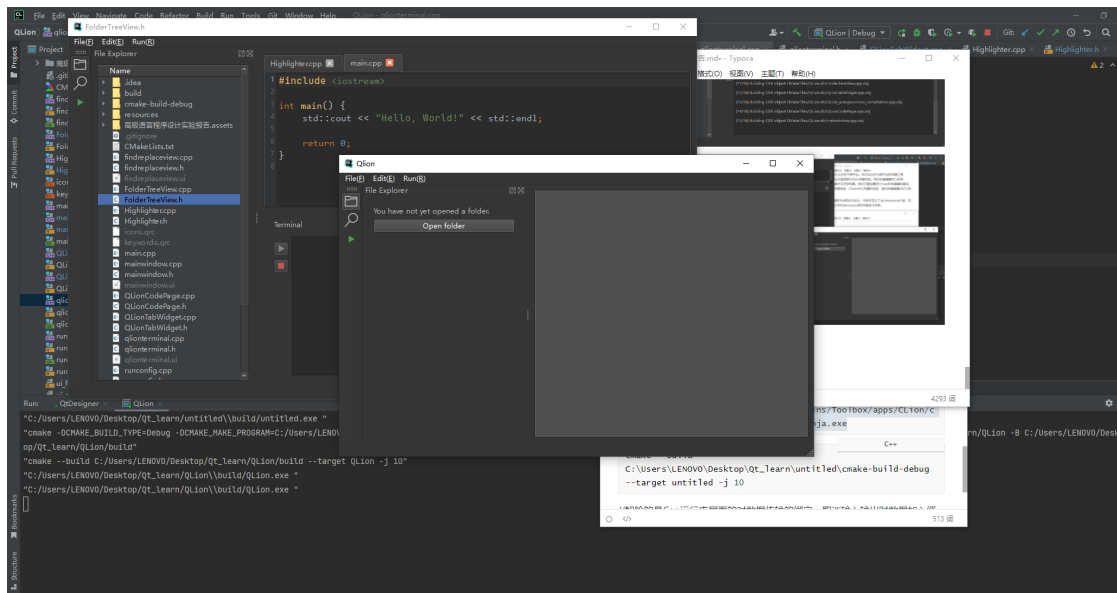
一个Cmake项目的运行需要经历生成，构建，执行三个阶段。将这三个阶段以及空闲阶段作为状态构建枚举类，并作为QLionTerminal的状态机。同时在类中自定义了信号，当命令执行结束后showFinished槽函数被调用其功能是将命令执行状态打印到模拟终端UI上(使用PlaintextEdit实现)。并根据状态机的状态发射自定义信号给mainWindow，mainWindow根据状态判断是否要执行下一步命令，以及执行哪个命令。通过读取项目中的CmakeList.txt可以判断可执行对象(add_executable)。一个项目中可能有多个可执行对象，由于时间有限只选取第一个可执行对象执行，多个对象执行的原理是一样的。若进行拓展，可读取所有可执行对象在右上角展示供用户选择(像Clion一样)。

Hello world程序执行:



同样的道理，我们甚至可以让它构建自己：





单元测试及收获

如PA中所述，未经测试的代码永远是错的。尽管已经在编写代码中进行了大量单元测试，在此一一列举出成功的样例也没有意义。这一部分可以参见项目视频。而且我们只是体会文本编辑器的工作方式和原理，而非真的去造这么一个轮子出来，像PA中提到的KISS法则样，只是实现了最基础的功能，更高级的设计，安全甚至性能都不是在一开始的实现过程需要考虑的，追求面面俱到只会增加代码维护的难度。即便如此，由于时间仓促，前面的分析可以看到，设计上仍旧有失误，也难免会出现这样那样未全面测试到的问题。当然，这个项目除了完成作业以外纯粹是兴趣驱动，没有功利性的目的，更没有任何实际价值(我们为什么不用VSCode呢)，因此暂时就容许这些失误出现吧(毕竟还有很多更重要的事情要做)，后续对bug的修复和对代码的重构在自己功力更深厚，思路更清晰的时候做，会更容易一些。

借物表

续加仪的代码编辑器给了我灵感和启动项目的动力。基本的高亮部分和行号绘制部分参考了这个项目。

JetBrains Icons

VSCode Icons