



信息与软件工程学院

## 企业实习总结报告

学 号： 2021091202022

姓 名： 刘治学

专业方向： 软件工程（互联网+）

企业名称： 成都中科合迅科技有限公司

实习岗位名称： 实习生

企业指导教师： 刘鹏江

院内指导教师： 王伟东



---

## 摘要

关键词:

---

## ABSTRACT

Keywords:

---

## 1 实习概况

### 1.1 实习公司情况和岗位职责

#### 1.1.1 实习公司概况

本人实习单位，成都中科合迅科技有限公司，成立于 2013 年，总部位于成都市高新区科园南路 1 号海特国际广场 4 号楼 7-8 层，注册资本 2766 万元，现有员工 300 余人。是专业从事自主安全可控军用软件研发和网络空间军事装备研制的高科技企业。公司在北京、上海、武汉、西安设有分支机构，全国有超过 50 家的军队、军工科研客户。

公司的口号是“铸造军工品质，磨砺国产匠心”。国家目前的军工行业现状分析主要有三点。第一，一带一路、走出国门，必须建设能打胜仗的全球化现代军队，国家军队体制改革孕育重大行业发展机会。第二，现有科研体系无法满足国家强国强军梦想“军民融合、民参军”，给民营科技企业提供历史发展机遇；西方主要发达国家军民通用技术已超过 80%，军事专用技术已不到 15%。第三，我国国防军费 2017 年突破万亿大关，武器装备建设有望持续加速，战略空军、远洋海军、国防信息化成为军队建设的重中之重。

公司就在这样的背景下成立与发展，至今已过十年。公司的发展阶段大致可分三个阶段。第一个阶段，2013 年到 2017 年，是基础能力建设阶段，主要成绩有军工二级保密资格，GJB9001C-2017 质量管理体系，国家级高新技术产业等。第二个阶段，2017 年到 2020 年，是核心能力聚焦阶段，主要成绩有第二批国家级专精特新“小巨人”，四川省瞪羚企业，连续四年成都市新经济百强企业等。第三个阶段，2020 年至今，是核心业务产品化阶段，主要成绩有四川省雁阵企业——省发改委定向培育，四川省首批 30 家“新经济示范”四川省计算机学会科学技术一等奖等 8 个省、部级奖项等。公司还获得了很多其他的资质与荣誉。

1.1.2 岗位职责

公司的核心业务可以简单概括为“一个核心产品，两个基本能力，两个数字化方向”。具体见图 1-1：



图 1-1 公司核心业务

我在实习期间就参与在合迅智灵产品的研发中。这也是公司的核心产品。合迅智灵产品完成了在国产化软硬件环境下多平台的全覆盖，支持桌面、移动、嵌入式等多种平台，满足了军工科研院所适用的更高要求，并与银河麒麟、元心 OS、锐华 Reworks 等公司签订了战略合作协议，目前合迅智灵已进入国防科工局国产化工具软件推广应用目录。

公司合迅智灵产品的功能架构见图 1-2。进入公司以后我所在项目组一直在进行基础平台的研发工作。这一部分的具体工作将在具体任务部分进行阐述。



图 1-2 合迅智灵产品功能架构

---

## 1.2 项目的背景、意义以及国内外研究现状

LarkSDK 是一款通用 C++ 开发框架。框架的最终目的是简化应用程序的开发。为达此目的，LarkSDK 也提供了一整套语义直观明晰、设计模块化的 C++ 类库，为用户封装了操作系统平台与硬件差异、提供了开箱即用的基础编程工具。

和 STL、Boost 相比，LarkSDK 可以用于构建图形界面。

和 MFC、GTK 相比，LarkSDK 具备跨平台能力，同一套源码可以同时工作在 Linux 和 Windows 之下。

当然提到跨平台的 C++ 图形开发框架，Qt 在业界仍然是占据统治地位的产品。Qt 从上个世纪九十年代开始发展，直到现在都是业界的一面大旗，也为 LarkSDK 的研发提供了很多参考。Qt 所提供的功能已经非常完备，绝大多数从事 C++ 软件开发的工程师基本上都以 Qt 为第一选择。

但是可能由于战线过长或者历史遗留等问题，在研发 LarkSDK 的过程中，我们发现 Qt 中部分功能的设计也许并不是最优解，甚至部分功能的设计相当糟糕。同时加上国际形势和国产生态的需求，LarkSDK 应运而生。相比 Qt，LarkSDK 更轻量、更简单、更年轻，同时更懂国产生态。我们并不奢望取代 Qt，我们仅仅是为业界提供另一个选择。这条路并不容易，但我们坚信这是一条正确的路，难而正确的路。

随着自主可控 IT 行业重点部分建设日趋完善，国家现已正式步入产业链改革深水区。其中软件开发平台是 IT 行业的血液，它可以支持各种底层的芯片和操作系统，将下层硬件和操作系统细节封装起来，对上层应用软件提供统一、易用、简洁的开发接口与工具，是打通芯片、操作系统与应用软件的关键，更贯穿了应用软件的设计、开发、测试与维护全生命周期。

目前，国内在软件开发平台领域的缺失，导致国内软件行业在开发软件时不得不大量使用国外厂商的开发平台，从而遭遇信息安全存在隐患、国产操作系统环境适配性差、维护服务没有支持、版权使用存在风险等重要问题，没有国产软件开发平台支持，软件开发环节效率低、软件开发成果不稳定也成为国内软件行业普遍存在的问题。国产 C++ 软件开发平台的目的，是打造通用跨平台的国产化基础开发套件，提供对应用程序的关键共性运行逻辑的支持，进而构建完整的国产化软件集成开发环境，以构建完整的基于 C++ 语言的国产应用程序开发生态。这也是我们需要努力做一个属于国人自己的 LarkSDK 产品的真正原因。

LarkSDK 在设计之初就是致力于跨平台的，它封装了不同操作平台的特性和底层调用，隐藏了各个操作系统平台的不同之处，让用户可以专注于业务开发而无

---

需过多关心平台差异；同时，LarkSDK 也并非单纯的浅层 UI 库，它覆盖了从底层的操作系统事件监听与分发，到通用的跨平台用户组件的外观与行为定制；它既提供各种各样的底层代码工具，如常见元素容器和加解密算法，以提升 C++ 的开发效率和体验，又能够通过自带的用户界面框架直接构建图形用户程序，未来还将提供界面编辑器及自动化测试工具等，从而完成应用软件开发的全生命周期闭环。

和 Qt Framework 一样，LarkSDK 是一款单纯而完整的 C++ 开发框架。万丈高楼平地起，除必要的基础轮子外，LarkSDK 并不依赖任何其他框架而存在。

由于一些原因，目前国内的软件行业语境下，对于诸如语言、框架、工具、平台等技术概念，存在一定程度上的混淆使用。虽然不太愿意这么表达，但是在国内 LarkSDK 确实是唯一的一款通用基础开发框架，技术上并没有真正意义的竞品。这也是我们所看到的绝大部分需要使用 C++ 开发应用软件的场合都是 Qt 的原因。

而合迅智灵是一款全国产的、自主可控的、拥有完全自主知识产权的基础软件开发平台。LarkSDK 则是平台为应用软件开发提供的底层软件开发框架。我们致力于成为 Qt 之外的第二选择。

不过值得一提的是，站在市场层面解决实际需求的视角下，确实存在一些产品，和 LarkSDK 存在一定的功能覆盖，例如：

### 1、统信 DTK

DTK(Development ToolKit)是统信基于 Qt 开发的一整套简单且实用的通用开发框架，处于统信 UOS 操作系统的核心位置。其提供丰富的开发接口与支持工具，满足日常图形应用、业务应用、系统定制应用的开发需求，提供 30 余个预定义组件，如统信 UOS 浏览器、音乐、邮件等 40 余款 UOS 应用均使用 DTK 开发。

其本质上是一个在 Qt 的基础之上构建的扩展组件库。利用 Qt 框架与操作系统底层对接，借助 Qt 的能力实现各种具体的用户组件。用户本质上还是在使用 Qt 开发。其具备的跨平台能力本质上也是 Qt 本身的能力。

### 2、华为 ArkUI

ArkUI 是一套用于构建图形用户界面的声明式 UI 开发框架。它使用极简的 UI 信息语法，提供丰富的 UI 组件及包含实时界面预览工具在内的集成开发环境等。提供基于 ArkTS 开发语言的应用程序接口，支持各种 HarmonyOS 设备。

严格说 ArkUI 和 LarkSDK 在技术上并无关联，其本质是一套专门用于鸿蒙 HarmonyOS 生态的开发工具链的一部分，完整的鸿蒙 HarmonyOS 生态包含 ArkTS(基于 TypeScript 的开发语言)、ArkUI(基于 ArkTS 的一套界面组件语言)、



---

ArkCompiler (用于处理 ArkTS 的编译工具)，以及 DevEco Studio(基于 VSCode 构建的集成开发环境)，构成完整的生态工具链。也即是说，ArkUI 是生态专有生态的开发工具，和直接面向操作系统底层的通用开发工具 LarkSDK 并不处于一条技术路线上。

### 3、致远电子 AWTK

AWTK 全称为 Toolkit AnyWhere，是 ZLG 倾心打造的一套基于 C 语言开发的 GUI 框架。旨在为用户提供一个功能强大、高效可靠、简单易用、可轻松做出炫酷效果的 GUI 引擎，支持跨平台同步开发，一次编程，到处编译，跨平台使用。

然而 AWTK 本质上也是一套基于 SDL(opens new window)构建的 GUI 库。其主要能力，如图形渲染、跨平台与底层交互等，均由 SDL 框架提供。

总的来讲，LarkSDK 产品的意义是重大的，我个人也很幸运能参与到项目的研发当中。

---

## 1.3 实习目标和具体任务

### 1.3.1 实习目标

对于实习目标，我将其分为公司目标和个人目标。对于公司目标，我希望严格按照公司的安排，全身心投入合迅智灵基础平台的研发。对于个人目标，我希望能在实习期间，能跟随前辈的脚步，保质保量完成好安排到的工作，同时能够在实习期间学到更多知识和技术，以学习为主，以功利为辅，提升培养自己的技术和能力。

### 1.3.2 具体任务

#### 1、LarkSDK

前面提到，我目前参与在合迅智灵基础平台的产品研发当中，下面我对该基础平台的业务进行相关介绍。

合迅智灵基础平台，全称叫合迅智灵国产化基础开发套件，英文名叫 LarkSDK。LarkSDK 是合迅智灵产品 LarkStudio5 的核心部件，是一套跨平台的 C++基础开发库。

LarkSDK 对标 Qt。Qt 是一款历史悠久，发展稳定的跨平台 C++基础开发库。在全世界被广泛使用，但美中不足的是，该产品并不是国人开发。为顺应军工国产化的大趋势，公司在 2020 年起进入合迅智灵产品的研发，其中 LarkSDK 的部分作为基础平台，又是重中之重。

LarkSDK 分为以下三个主要子模块和三个扩展模块。

三个主要子模块：

- (1) LarkSDK-Core (larkcore)：核心类库，包含对象模型、事件机制、线程管理和主程序框架等。
- (2) LarkSDK-Util (larkutil)：实用跨平台工具类库，包含常用的数据结构和算法。
- (3) LarkSDK-GUI (larkgui)：图形绘制类库，包含跨平台图形绘制接口、基础窗体和常用组件，以及多绘图引擎支持支持等。

三个扩展模块：

- (4) LarkSDK-DB (larkdb)：数据库支持模块。
- (5) LarkSDK-Network (larknetwork)：网络编程支持模块。
- (6) LarkSDK-XML (larkxml)：XML 支持模块。

公司与本校本院某实验室在 2021 年末达成了合作，校方开始了 LarkSDK 部分的初步编写。到 23 年末为止已经经过一期和二期的阶段，已初步完成开发。但由

于学校学生对实际工程的接触较少以及理论知识不牢固，编写的代码是存在很多问题的。在我进入公司以前，公司的前辈们就针对部分问题进行了优化重构，但是整体来看仍难以达到标准。

因此在进入公司以后，我的第一份任务就是进行代码走查。何为代码走查？简单来讲就是读别人的代码发现问题，但是这其中的工作量和难度也是不小的。领导对我们的要求是对于该部分涉及到的内容，包括涉及到的其他代码，都要阅读和理解，然后汇报。虽然枯燥甚至有些难，但我确实学到了很多知识和技巧。我的老总曾经说过：“提升技术的最好办法就是阅读代码。”我觉得说的有理，阅读别人的代码，首先是理解别人的思路，然后延申思考，思考他这里为什么写得好，写得不好，想想如果是自己该如何构思，如何下笔，这对一个问题从 0 到 1 的剖析是非常有帮助的，而我在学习的过程中，不仅要学习知识和技术，更重要的是培养工程上的思维和方法。

以下列出实习过程中的代码走查记录表 1-1：

表 1-1 代码走查表

迭代时间	走查代码
1.15 - 1.26	LStack、LQueue、LByteArray
1.29 - 2.8	LObject、LApplication、LSignal
2.19 - 3.1	线程管理、线程数据、互斥锁、读写锁部分
3.4 - 3.15	LPen、LBrush、LLinearGradient、LMenu、LMenuItem、LMenuItemSeparator
4.7 - 4.19	LarkXML 模块

既然做了代码走查，那么必然需要记录，后续优化重构。代码走查的问题是发现问题，记录问题，而最终的解决依赖于后续的代码优化重构。在实习期间中，我完成了线程部分代码、LPen、LBrush 等的优化重构，完成了栈 LStack 和队列 LQueue 容器、字符串列表（LStringList）、关联性容器（LHash、LMap、LSet）、日期时间数据类型（LDateTime、LDate、LTime）以及三元组容器（LTrio）等工具类代码的优化重构。同时我自主研究 QDir 和 QFileInfo 的设计，去掉设计不合理的地方，设计出了我们自己的 LFileSystemPath 和 LFileSystemEntry，与上面的结构分别对应。在组内成员的共同努力下，util 部分目前已经基本全部处理完毕。经过测试，目前功能稳定可用，具体见图 1-3。

	A	B	C	D		K	L	M	N	O	P	Q
	模块	功能	C++ 类			质量情况评估	建议措施	具体问题与修改意见	优先级	处理人	处理状态	处理完成时间
LarkSDK-Util	字符串容器	字符串列表	LStringList	文件才能通过编译 [数] = +, -, ==, !=	初始可用	可用	代码重构	1. 后续继承 LVector 中重新实现相关功能 2. 重新添加 string.h 文件, 不使用前置声明 3. 继承 LVector 后调用父类方法实现拷贝操作 4. 继承 LVector 后调用父类方法实现 sort 操作 5. <3.0 更新: 删除所有无用的接口和除 * 号以外的运算符重载, 只保留构造函数 6. 后续实现 LVariant 数据类型支持	中	刘海洋	待测试	2024Q2
	关联容器	哈希容器	LHash<key, T>	不完全适用 (参考 QSet), 类似 LSet.	不可用	重新实现	以 std::unordered_map 重构	中	刘海洋	待测试	2024Q2	
		集合容器	LSet<T>	不完全适用 (参考 QSet), 类似 LSet.	不可用	重新实现	以 std::unordered_set 重构	中	刘海洋	待测试	2024Q2	
		字典容器	LMap<key, T>	blue_type 是 T, key- value 组成的 pair	不可用	重新实现	以 std::map 重构	中	刘海洋	待测试	2024Q2	
	日期时间数据模型		LDateTime LDate LTime	的又内联 []()和isvalid()多余 冗余 使用 int Time 一般初始化为 入的读读字符串格	初始可用	优化修改	1. 规范代码, 合理使用内联和外联 2. 目前存在操作修正, 无需对无效时间做异常 3. 多个接口中多次使用了将 LDateTime 转换为 std::tm 且代码量较大, 提供私有接口 4. 移除 isNull() 和 isInvalid() 接口 5. 操作运算符重载: <, <=, ==, >, >= 的实现直接对私有数据成员进行比较即可, 且在实现 <=, >= 时使用操作运算符重载 <, <=, >, >= 即可 6. time_t 表示时间戳, 接口参数传递和返回值为 time_t, 部分转换时保留接口保留 time_t 7. LDateTime 构造函数初始时间修改为 0 8. 设置时间的相关接口判断改为三目运算符实现 9. LDateTime 的 fromString(const LString &string) 接口需要传入的格式进行预处理 10. LDateTime 的操作运算符 < 不使用 std::time 实现, 直接比较即可 11. 直接使用操作修正时间设置即可, 不需要 isInvalid 判断且不需要返回 bool	中	刘海洋	待测试	2024Q2	

图 1-3 util 模块优化重构工作记录表

第三，课题调研。绝大多数的功能都是在产品不断的发展和迭代中，因为需求而应运而生的。面对一个新的需求或者问题，需要进行大量的调研才能解决问题。这不仅是工作的过程，更是一个学习和进步的过程。在实习期间，我很幸运参与了一些课题的完整调研。我从中学到了很多知识和技术，也为公司的产品贡献了自己的力量。具体见表 1-2。

表 1-2 课题调研表

迭代时间	调研课题
4.7 - 4.19	标准库 string 的 sso 优化对 LVector 插入影响的探究
4.22 - 4.30	LDir 和 LFileInfo 的语义和设计
5.13 - 5.24	Qt Graphics View Framework 预研
5.27 - 6.7	一些关于空间数据结构的简单研究与实现
6.17 - 6.28	在 X11 下使用 Cairo 引擎绘制图形
7.1 - 7.12	使用 Woboq CodeBrowser 搭建源代码网站

2、LarkTestKit

LarkSDK 作为一个跨平台的 C++ 基础开发库。我们是采用 googleTest 开源框架进行测试的，为了做到国产化适配，故提出了自动化测试框架 LarkTestKit 的需求。该部分源代码交由校方某同学负责，提交了初版，我的任务是审核该部分代码，与校方同学和测试沟通，并设计编写测试用例。巧合的是，负责 LarkTestKit 的学长在四月中旬也来到公司也进行线下实习。因而在中期的工作中，我协助他一起完成 LarkTestKit 的工作，一起讨论 LarkTestKit 中的部分设计思路、实现方案等。目前 LarkTestKit 已经测试完毕，初步测试结果良好，功能稳定可用。

---

## 2 复杂工程问题和解决方案

本部分将针对上述具体任务当中的某个环节、某个步骤遇到的工程问题出发，分析问题的来龙去脉，并设计合理的解决方案应对，总结于此。有部分问题在早期的报告中已阐述过，这里不再赘述。

### 2.1 Qt Graphics View Framework 预研

该架构涉及到的最主要的三个类是 `QGraphicsScene`、`QGraphicsView` 和 `QGraphicsItem`。

#### 2.1.1 整体流程

##### 1. 绘制流程（`QGraphicsItem`->`QGraphicsScene`->`QGraphicsView`）

`QGraphicsItem` 当中保存了自身的“场景坐标”供 `QGraphicsScene` 进行管理。在绘制时，由 `QGraphicsView` 对象调用渲染方法，根据自身所设置的可视化相关属性，基于“视图坐标”确定将要绘制的 `QGraphicsScene` 当中有哪一部分“场景坐标”内的图元需要渲染，随后通过 `QGraphicsScene` 提供的方法，将属于这部分“场景坐标”内的图元(也就是 `QGraphicsItem`)全部找出，并渲染这些图元到可视化 viewport 中。

##### 2. 事件流转（`QGraphicsView`->`QGraphicsScene`->`QGraphicsItem`）

由 `QGraphicsView` 绘制出的 viewport 是与用户直接交互的对象，用户发起的 UI 事件都由 `QGraphicsView` 首先接收，它在接收到事件以后对其中的部分参数进行适当的处理(如鼠标事件的坐标进行转换)，随后将事件转发给 `QGraphicsScene` 对象，由 `QGraphicsScene` 确定事件发送到哪个具体的图元(如鼠标事件发送到符合坐标位置的图元，键盘事件发送到当前焦点所在的图元)，图元在接收到事件以后作自行处理。

##### 3. `QGraphicsScene` 的查询图元的 BSP 树相关算法

在实际的工程场景中，极有可能会出现一个 `QGraphicsScene` 管理非常多个 `QGraphicsItem`，例如几百万个，当用户与 `QGraphicsView` 交互的时候，需要经过 `QGraphicsScene` 将本事件传递到某个或者某些具体的图元，如果挨个遍历每个图元，并且判断是否触发事件，那么耗费的代价就太大了。我们希望做到的效率是几百万条数据在几毫秒以内能够确定目标图元，因此需要上相应的算法。

##### 4. 图元实现优先级以及顺序和事件实现优先级以及顺序

`QGraphicsItem` 本质上是一个抽象父类，为子类规定了各种需要覆写的方法，

子类通过重写这些方法能够很好的被 QGraphicsScene 所管理，最终呈现出来。因此不仅支持官方提供的矩形、椭圆、文本框等，还能处理自定义类型的图元。

在现阶段，先不考虑纯虚的抽象类，先把 QGraphicsItem 当成矩形框来做，先和 QGraphicsScene 把流程跑通，后续在扩展的时候也能非常快就完成工作。

事件处理是一个大块。经讨论，事件处理的优先级是鼠标事件大于键盘事件大于拖拽事件。因此第一步实现的时候，先实现最基本的 handleMouseEvent，当这个流程通了以后，后续的所有操作都是一模一样的了。

## 5. 整体实现顺序

先把 QGraphicsView 剔出来，QGraphicsView 实际上只负责可视化的绘制，真正做处理的操作在 QGraphicsScene 这一层，因此管理类 QGraphicsScene 和图元类 QGraphicsItem 是第一步需要实现的。

目前的工作计划经商讨以后，决定先调研 QGraphicsScene 和 QGraphicsItem 的源码，明确如何以最小的代价实现我们目前需要的功能，包括事件转发、绘制逻辑、坐标系统等，这样明确需求和设计以后再实现初版，搭好整体架构。

### 2.2.2 重要功能总结

#### 1、QGraphicsScene

QGraphicsScene 类提供了一个管理大量 QGraphicsItem 的容器。

QGraphicsScene 类只管理所有的图元，若想要可视化场景，需要配合 QGraphicsView。

QGraphicsItem 作为 QGraphicsScene 类的图元，自然有很多接口都是围绕着 QGraphicsItem 来的，例如：

(1) 添加图元：除了最基本的 addItem()，还有更多针对特殊类型图元的 add\*() 函数。

```
void addItem(QGraphicsItem *item);
QGraphicsEllipseItem *addEllipse(const QRectF &rect, const QPen &pen = QPen(), const QBrush &brush = QBrush());
QGraphicsLineItem *addLine(const QLineF &line, const QPen &pen = QPen());
QGraphicsPathItem *addPath(const QPainterPath &path, const QPen &pen = QPen(), const QBrush &brush = QBrush());
QGraphicsPixmapItem *addPixmap(const QPixmap &pixmap);
QGraphicsPolygonItem *addPolygon(const QPolygonF &polygon, const QPen &pen = QPen(), const QBrush &brush = QBrush());
QGraphicsRectItem *addRect(const QRectF &rect, const QPen &pen = QPen(), const QBrush &brush = QBrush());
QGraphicsTextItem *addText(const QString &text, const QFont &font = QFont());
QGraphicsSimpleTextItem *addSimpleText(const QString &text, const QFont &font = QFont());
QGraphicsProxyWidget *addWidget(QWidget *widget, Qt::WindowFlags wFlags = Qt::WindowFlags());
inline QGraphicsEllipseItem *addEllipse(qreal x, qreal y, qreal w, qreal h, const QPen &pen = QPen(), const QBrush &brush = QBrush())
{ return addEllipse(rect: QRectF(x, y, w, h), pen, brush); }
inline QGraphicsLineItem *addLine(qreal x1, qreal y1, qreal x2, qreal y2, const QPen &pen = QPen())
{ return addLine(line: QLineF(x1, y1, x2, y2), pen); }
inline QGraphicsRectItem *addRect(qreal x, qreal y, qreal w, qreal h, const QPen &pen = QPen(), const QBrush &brush = QBrush())
{ return addRect(rect: QRectF(x, y, w, h), pen, brush); }
```

图 2-1 addItem() 系列函数

(2) 查找图元：在大规模数据的情况下保证在极短的时间内查询到对应的图元，Qt 使用的是索引算法，使用的是 BSP（二进制空间分区）树。能够在极短的时间内在极大的数据规模中确定某个图元的位置，这也是 QGraphicsScene 的最大优势之一。

(3) 移除图元：removeItem()。

(4) 管理图元状态，处理图元选择和焦点处理等。

- setSelectedArea(): 可以传递一个形状范围来选择图元项目
- clearSelection(): 清除当前选择
- selectedItems(): 获取被选择的图元的列表
- setFocusItem()、setFocus(): 设置图元获取焦点
- focusItem(): 获取当前焦点

(5) 接受事件：用户通过 QGraphicsView 触发事件，并通过 QGraphicsScene 将事件转发到对应的图元：例如鼠标按下、移动、释放和双击事件，鼠标悬停（LarkSDK 目前好像没有）、滚轮事件，键盘输入焦点和按键事件，拖拽事件等。

```
protected:
    bool event(QEvent *event) override;
    bool eventFilter(QObject *watched, QEvent *event) override;
    virtual void contextMenuEvent(QGraphicsSceneContextMenuEvent *event);
    virtual void dragEnterEvent(QGraphicsSceneDragDropEvent *event);
    virtual void dragMoveEvent(QGraphicsSceneDragDropEvent *event);
    virtual void dragLeaveEvent(QGraphicsSceneDragDropEvent *event);
    virtual void dropEvent(QGraphicsSceneDragDropEvent *event);
    virtual void focusInEvent(QFocusEvent *event);
    virtual void focusOutEvent(QFocusEvent *event);
    virtual void helpEvent(QGraphicsSceneHelpEvent *event);
    virtual void keyPressEvent(QKeyEvent *event);
    virtual void keyReleaseEvent(QKeyEvent *event);
    virtual void mousePressEvent(QGraphicsSceneMouseEvent *event);
    virtual void mouseMoveEvent(QGraphicsSceneMouseEvent *event);
    virtual void mouseReleaseEvent(QGraphicsSceneMouseEvent *event);
    virtual void mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event);
    virtual void wheelEvent(QGraphicsSceneWheelEvent *event);
    virtual void inputMethodEvent(QInputMethodEvent *event);
```

图 2-2 事件转发系列函数

QGraphicsScene 另一个重要的功能就是转发 QGraphicsView 的事件，通过一系列操作，例如通过鼠标点击的坐标计算出到底是选中了哪个图元，键盘事件对应的哪些图元具有焦点等，能够将这些事件转发给对应的图元，最后进入真正的事件循环进行处理。



---

## 2、QGraphicsView

QGraphicsView 类真正提供可视化 QGraphicsScene 的内容的功能,它在一个可滚动的 viewport 之内将一个 QGraphicsScene 中的内容实现可视化。

QGraphicsView 主要功能包括但不限于:

(1) 设置可视化操作的属性: QGraphicsView 中提供了大量可设置的属性用以指示在实现可视化操作时的各种具体事项,如 RenderHints 提供参数初始化用于绘制的 QPainter, Alignment 提供当前视图中所绘制的场景的对齐方式。

```
QPainter::RenderHints renderHints() const;
void setRenderHint(QPainter::RenderHint hint, bool enabled = true);
void setRenderHints(QPainter::RenderHints hints);

Qt::Alignment alignment() const;
void setAlignment(Qt::Alignment alignment);

ViewportAnchor transformationAnchor() const;
void setTransformationAnchor(ViewportAnchor anchor);

ViewportAnchor resizeAnchor() const;
void setResizeAnchor(ViewportAnchor anchor);

ViewportUpdateMode viewportUpdateMode() const;
void setViewportUpdateMode(ViewportUpdateMode mode);

OptimizationFlags optimizationFlags() const;
void setOptimizationFlag(OptimizationFlag flag, bool enabled = true);
void setOptimizationFlags(OptimizationFlags flags);

DragMode dragMode() const;
void setDragMode(DragMode mode);

#ifdef QT_CONFIG(rubberband)
    Qt::ItemSelectionMode rubberBandSelectionMode() const;
    void setRubberBandSelectionMode(Qt::ItemSelectionMode mode);
    QRect rubberBandRect() const;
#endif

CacheMode cacheMode() const;
void setCacheMode(CacheMode mode);
void resetCachedContent();

bool isInteractive() const;
void setInteractive(bool allowed);
```

图 2-3 设置可视化操作的接口

(2) 对场景(Scene)进行可视化与视觉效果调整: QGraphicsView 对象的成员方法 render 对场景进行可视化的绘制呈现在 viewport 中,并提供了一系列方法对 viewport 整体的视觉效果进行调整,如 centerOn 方法将滚动 viewport 中的内容以确保场景坐标 pos 在视图居中,fitInView 方法将缩放并滚动 viewport 中的内容使得场景内的矩形区域 rect 铺满当前 viewport。



---

```

void centerOn(const QPointF &pos);
inline void centerOn(qreal x, qreal y);
void centerOn(const QGraphicsItem *item);
void ensureVisible(const QRectF &rect, int xmargin = 50, int ymargin = 50);
inline void ensureVisible(qreal x, qreal y, qreal w, qreal h, int xmargin = 50, int ymargin = 50);
void ensureVisible(const QGraphicsItem *item, int xmargin = 50, int ymargin = 50);
void fitInView(const QRectF &rect, Qt::AspectRatioMode aspectRatioMode = Qt::IgnoreAspectRatio);
inline void fitInView(qreal x, qreal y, qreal w, qreal h,
    Qt::AspectRatioMode aspectRatioMode = Qt::IgnoreAspectRatio);
void fitInView(const QGraphicsItem *item,
    Qt::AspectRatioMode aspectRatioMode = Qt::IgnoreAspectRatio);

void render(QPainter *painter, const QRectF &target = QRectF(), const QRect &source = QRect(),
    Qt::AspectRatioMode aspectRatioMode = Qt::KeepAspectRatio);

```

图 2-4 进行可视化与视觉效果调整的接口

(3) 管理“场景（Scene）坐标”与“视图（View）坐标”之间的数学关系，并提供方法对视图内容施行各种坐标变换。QGraphicsScene 对象当中的各个图元有其其在 QGraphicsScene 中的坐标即“场景坐标”，它们代表了各个图元在 QGraphicsScene 中的位置信息；而 QGraphicsView 对各个图元进行绘制以及调整变换时则是通过由自身管理的“视图坐标”，它们代表了各个要绘制的图形在 viewport 中的位置信息。QGraphicsView 可以由用户设置“场景坐标”向“视图坐标”变换的方式，对 viewport 实现旋转、伸缩等坐标变换，同时由于 QGraphicsView 的绘图使用“视图坐标”，因此这个过程不会干扰图元自身的“场景坐标”。此外还提供 mapToScene/mapFromScene 方法供用户调用实现这两种坐标之间的数学换算。

(4) 接受鼠标和键盘的事件，并通过处理传递给 QGraphicsScene 对象，进而通过索引算法转发给对应的图元。

### 3、QGraphicsItem

QGraphicsItem 是所有图元的基类，可以派生出各种典型的形状（例如矩形、椭圆、文本等）和自定义的形状。

QGraphicsItem 主要功能包括但不限于：

(1) 接受 QGraphicsScene 传递的事件：进行事件处理，例如鼠标按下、移动、释放和双击事件，鼠标悬停、滚轮事件，键盘输入焦点和按键事件，拖拽事件等。

---

```
protected:
    bool sceneEvent(QEvent *event) override;
    void mousePressEvent(QGraphicsSceneMouseEvent *event) override;
    void mouseMoveEvent(QGraphicsSceneMouseEvent *event) override;
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *event) override;
    void mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event) override;
    void contextMenuEvent(QGraphicsSceneContextMenuEvent *event) override;
    void keyPressEvent(QKeyEvent *event) override;
    void keyReleaseEvent(QKeyEvent *event) override;
    void focusInEvent(QFocusEvent *event) override;
    void focusOutEvent(QFocusEvent *event) override;
    void dragEnterEvent(QGraphicsSceneDragDropEvent *event) override;
    void dragLeaveEvent(QGraphicsSceneDragDropEvent *event) override;
    void dragMoveEvent(QGraphicsSceneDragDropEvent *event) override;
    void dropEvent(QGraphicsSceneDragDropEvent *event) override;
    void inputMethodEvent(QInputMethodEvent *event) override;
    void hoverEnterEvent(QGraphicsSceneHoverEvent *event) override;
    void hoverMoveEvent(QGraphicsSceneHoverEvent *event) override;
    void hoverLeaveEvent(QGraphicsSceneHoverEvent *event) override;
```

图 2-5 QGraphicsScene 传递的事件的方法

## (2) 坐标系

每个 Item 都有自己的本地坐标系，一般以自身的中心为(0, 0)，自身的方向作为基准方向建立，多个 Item 的情况就如图。因此需要通过某些机制将不同 Item 的坐标联系在一起。

为了统一方便的管理，引入 parent-child 的关系。每个对象的变换都依赖于其父对象的坐标。子对象的 pos()接口返回的是其在父对象坐标系统中的坐标。子对象的坐标处理是首先通过父对象不断向上传递，最终得到一个真实的坐标。同理，父对象的坐标变换也会同理影响到子对象的真实位置（批量处理），但是注意子对象存储的坐标没有变化，这样就非常好维护了。

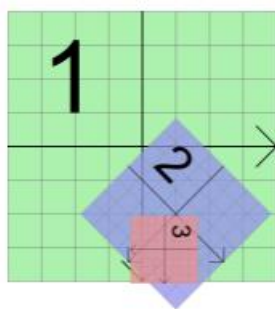


图 2-6 坐标变换示意图

## (3) 坐标变换

QGraphicsItem 除了提供基本位置 pos()以外，还支持坐标变换，例如 setRotation() 旋转，setScale()缩放等。

---

同理，父对象的变换会影响子对象，例如父对象顺时针转 90 度，子对象会跟着一起转 90 度。

#### (4) 提供分组功能（后续考虑）

通过前面的坐标系统可以知道，每个 `QGraphicsItem` 都有一个父对象，也可以有一系列子对象，这是类似于对象树机制的在构造的时候确定的关系。当然也可以手动分组。`QGraphicsItemGroup` 是一个特殊的派生类，该类记录了一系列的图元为一个组，在该组的所有图元通过 `Group` 调用的时候移动、事件处理等都会进行批量处理。

#### (5) 提供编写自定义图元的接口（后续考虑）

创建一个 `QGraphicsItem` 的子类，然后覆写两个纯虚函数 `boundingRect()` 返回该图元项目所绘制区域的估计值，`paint()` 实现实际绘图。

`boundingRect()` 返回的是一个 `QRectF` 类型，将该图元的外部估计边界定义为矩形，这个方法也可以为不同 `Item` 的范围做大致的估算，可以被其他的方法所调用，省去一些暴力查找的过程。当然对于真正的矩形 `boundingRect()` 可以返回精确的范围，对于其他的曲线或者不规则的形状只能做大致的范围。

#### (6) 碰撞检测（后续考虑）

通过 `shape()` 函数和 `collidesWith()` 这两个虚拟函数，可以支持碰撞检测。`shape()` 函数返回一个局部的坐标 `QPainterPath`。目前没有细节调研 `QPainterPath`，只知道 `QPainterPath` 记录了绘图的路径，比如 2D 图形的形状是由某些直线、曲线等构成的，通过这个能够确定图形的形状。

`QGraphicsItem` 会根据默认的 `shape()` 函数自动处理碰撞检测，实现合理的效果，比如在碰撞区域应该如何进行绘制。如果用户想要定义自己的碰撞检测，可以通过 `collidesWith()` 实现。

#### (7) 图元顺序（后续考虑）

难免会发生两个图元的范围出现重叠的情况。合理处理顺序决定了鼠标点击的时候哪些场景会接受鼠标事件。一个比较合理的想法是子对象位于父对象的顶部，而同级对象之间按照定义的顺序进行堆叠。例如添加对象 A、B，那么对象 B 位于 A 的顶部。这是比较符合自然逻辑的，Qt 也是这样做的。

Qt 提供了一些可以更改项目的排序方式的接口。例如可以在一个图元项目上调用 `setZValue()`，以将其显式堆叠在其他同级项目之上或之下。项的默认 Z 值为 0，具有相同 Z 值的项按插入顺序堆叠。还可以设置 `ItemStacksBehindParent` 标志以将子项堆叠在其父项之后。

---

### 2.2.3 2D BSP 树在 QGraphicsScene 中的应用

BSP 树构造一个  $n$  维空间到凸子空间的分层细分 (a BSP tree is a heirarchical subdivisions of  $n$  dimensional space into convex subspaces)。每个节点都有一个前叶子节点和后叶子节点。从根节点开始, 所有后续插入都由当前节点的超平面划分。在二维空间中, 超平面是一条线。在 3 维空间中, 超平面是一个平面。BSP 树的最终目标是让超平面的分布情况满足“每个叶节点都在父节点超平面的前面或后面” (The end goal of a BSP tree if for the hyperplanes of the leaf nodes to be trivially "behind" or "infront" of the parent hyperplane.)。

BSP 树对于与静态图像的显示进行实时交互非常有用。在渲染静态场景之前, 需要计算 BSP 树。可以非常快地 (线性时间) 遍历 BSP 树, 以去除隐藏的表面和投射阴影。通过一些工作, 可以修改 BSP 树以处理场景中的动态事件。

下面是在对象空间构建 BSP 树的过程:

(1) 首先, 确定要划分的世界区域以及其中包含的所有多边形。为了方便讨论, 我们将使用一个二维世界。

(2) 创建一个根节点  $L$ , 该节点本身对应一个分区超平面(在二维世界中, 分区超平面就是直线); 同时这个节点维护一个多边形列表, 在节点刚刚创建时, 多边形列表中保存着它对应的超平面所划分的目标区域(对  $L$  而言, 它对应的直线所划分的目标区域就是整个二维世界)当中的所有多边形。

(3) 使用  $L$  对世界进行分区, 假设分为了两个区域  $A$  和  $B$ 。在根节点上创建两个叶子节点分别对应  $A$  和  $B$ , 并将世界中的所有多边形移动到  $A$  或  $B$  的多边形列表中。遵循以下规则:

对于世界中的每个多边形:

- 如果该多边形完全位于  $A$  区域, 请将该多边形移动到  $A$  区域的节点列表中。
- 如果该多边形完全位于  $B$  区域, 请将该多边形移动到  $B$  区域的节点列表中。
- 如果该多边形与  $L$  相交, 则将其拆分为两个多边形, 并将它们移动到  $A$  和  $B$  的相应多边形列表中。这种情况下, 算法必须找到多边形与分割线  $L$  的交点, 以确定多边形的哪一部分位于哪个区域。
- 如果该多边形与  $L$  重合(也就是说, 恰好是一条位于  $L$  上的线段), 请保持其仍位于节点  $L$  处的多边形列表中。

在上述步骤完成后, 节点  $A$  和  $B$  的多边形列表已经生成, 但是  $A$  和  $B$  尚未完成创建分区超平面并划分的步骤, 也就是说, 对于当前的节点  $A$ , 它的状态与步骤 1 中的“整个二维世界”一致(区域确定, 包含的多边形确定), 只不过区域大小有

---

差别，B 同理。

(4) 在 A 和 B 上继续划分其所对应的二维区域，并将上述算法递归地应用于 A 和 B 的多边形列表。

最终的切分效果图如图 2-7 所示：

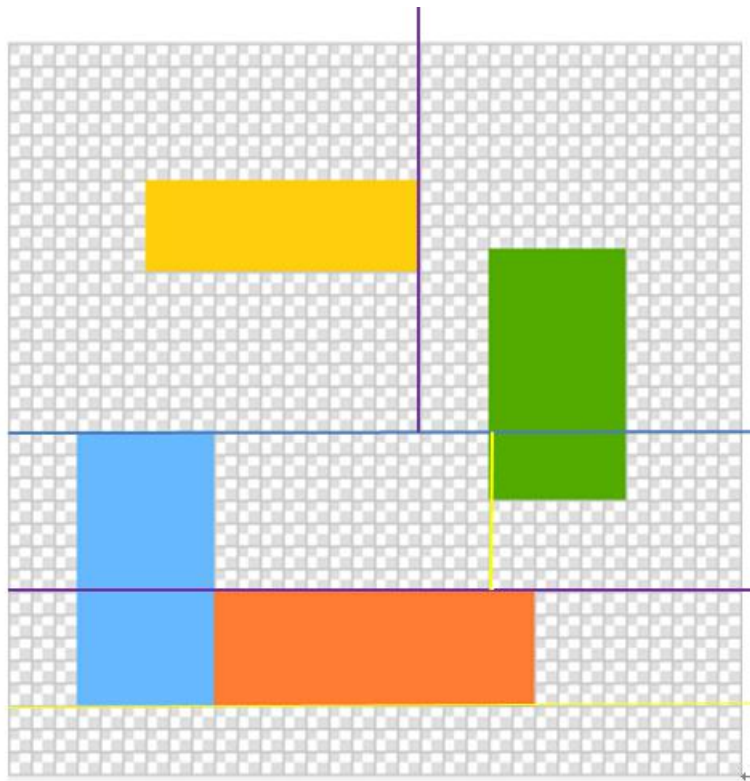


图 2-7 切分效果图

---

## 2.2 一些关于空间数据结构的简单研究与实现

### 2.2.1 前言

首先，我们来了解一下计算机中的“视图”“图形”等这些概念。我们日常的语境当中在说“视图”、“照片”、“图形”等等这些和图像有关的概念时，一般都认为它单纯指代一个具有视觉效果的画面——比如高考卷子上立体几何的那个题，你会把它对应到“图形”的概念上；你相册里面那张让你看了就来气的跟前对象的合照，你会把它对应到“照片”的概念上，等等等等。不过在计算机中，“视图”“图像”这些概念的涵义还包含了其他的方面，对于一个图像，计算机所要关注的不仅是它呈现的视觉效果，还要使用合适的数字格式存储图像，并按照用户的要求对图像进行分析、处理和加工。我们可以把计算机中的“视图场景”看作一个“虚拟的视图场景”：它在概念上更多地指代的是对图像的数字格式存储以及处理加工的相关操作，不妨回想一下我们前面叙述“空间数据结构”的时候所说的“保存图元信息”“定位查找元素”这些关键词，就对应的是这样的概念，很好理解，数字格式便是图像在计算机中的存在形式。这也是前面为什么要用“虚拟”一词，在计算机中说到“图像”“视图”等等时，我们不应直接以日常的语境将它理解成一幅纯粹的“视觉画面”，而是要关注到计算机中实际存储并操作的对象——“数字格式的图像”。我们下面的内容当中也都基于上述概念，讨论的是“数字格式的图像”。

假如我们现在要设计一个视图场景(Scene)，在其中包含很多图元对象(Item)。就像这样：



图 2-8 视图场景实例



---

上面就是一个场景，其中包含了大量的线条与色块等等，这些所有的图形要素都统称为“图元”。而场景就负责管理并绘制图元、接收并向对应的图元转发 UI 事件。我们现在假设场景是静态场景，也就是场景中的图元在初始化以后就不会有任何改变。

现在我们想让这个场景当中的图元响应用户的 UI 操作，比如说，用鼠标点击场景中的某个位置，场景就需要找到有哪根线条覆盖到了那个位置，然后让那个线条按照某种规则动起来。虽然我们在实现这个逻辑的时候是多么地希望能有一个“点信息表”可以根据鼠标点下的位置直接检索出对应的点上有哪些图元覆盖到这样的相关信息，但场景保存的是“各个图元”相关的信息成一个“图元信息表”，而不太可能把“每个点”相关的信息保存成“点信息表”——如果想用“点”来衡量一整个视图场景的大小，那真是让人吐血，想想你那 8K 超清的大屏幕上算出来有多少个点吧，如果想要保存“每个点被哪些图元覆盖到”这样的信息，需要一个什么规模的二维数组(.....)。所以我们知道用户的鼠标落在哪个“点”上之后，要做的是去“图元信息表”寻找“哪些图元覆盖了这个点”，毕竟，场景保存的是“各个图元”的“图元信息表”，不会丧心病狂到去把“每个点”的信息保存成一个“点信息表”。

你可能会说，这多简单，对图元信息表中的每个图元遍历一次，看看哪个图元覆盖了那个位置呗。我们一般管这种做法叫暴力搜索，当然这不是说它完全错误，一个方法只要能达到目的那它就不是绝对的错误；但是必须考虑的事情是，宇宙是有限的，连一个葛立恒数大小的物理概念在我们的宇宙中都找不到，那就不可能在讨论一个事情的时候抛开它的时空限制、可用资源限制这些事实层面的东西不谈。比如说，现在场景当中有一千万个图元(不要惊讶，这种情况当然有可能出现，比如说卫星地图或者高品质游戏等一些高精绘图场景，或者你哪天想起来要给你朋友展示一下属于码农的浪漫于是把参数调成一千万)，那用户每点一下就要暴力搜索数秒，如果这景象出现在一个游戏里那它早已被市场淘汰。但是前面我们又说过，“遍历图元”这样的操作逃不掉的，那么有什么能提高效率的方法呢？

计算机图形学对此问题早有研究，并提出了“空间数据结构”(Spatial Data)的概念。很多时候我们需要能够方便地在空间中定位和查找元素的数据结构来处理物体，这称为空间数据结构。空间数据结构将空间划分为多个层次多个区域，并在保存图元信息时使用对应的数据结构记录每个划分出的区域中完全或部分包含的图元并保存，这样就更方便定位和查找空间中的元素，被广泛用在图形学场景中用来加快运算。例如，现在场景中有一千万个图元，原本在遍历的时候需要对这一千万个图元都遍历，但是现在有一种空间数据结构将整个场景不重不漏地划分

---

成了 1000 个区域，那现在平均每个区域就只有一万个图元了，而用户点到的那个位置一定只落在某个区域当中(谁让它这么没出息只是一个点呢)，所以现在只需要遍历一万个左右的图元了。当然这只是一个比较理想化的模型，实际操作起来还会遇到其他的细节问题，比如说如果有很多图元它就是比你能划出来的区域更大该怎么办呢。而下文所要探讨的，就是均匀网格、BSP 这两种空间数据结构在二维空间下的情形以及实现时的各种细节问题。

### 2.2.2 均匀网格的概念

均匀网格(Grid)是一种空间数据结构，它使用了一个最为简单朴素的做法，就是将一个空间均匀地划分为大小相等的网格。在二维空间中，均匀网格在一个平面区域内使用等距的直线将其划分为大小相等的网格子区域。把空间划分成均匀网格，使用数组记录每个网格中包含的图元，就形成了一个简单的空间数据结构。下图展示了一个二维均匀网格的示例：

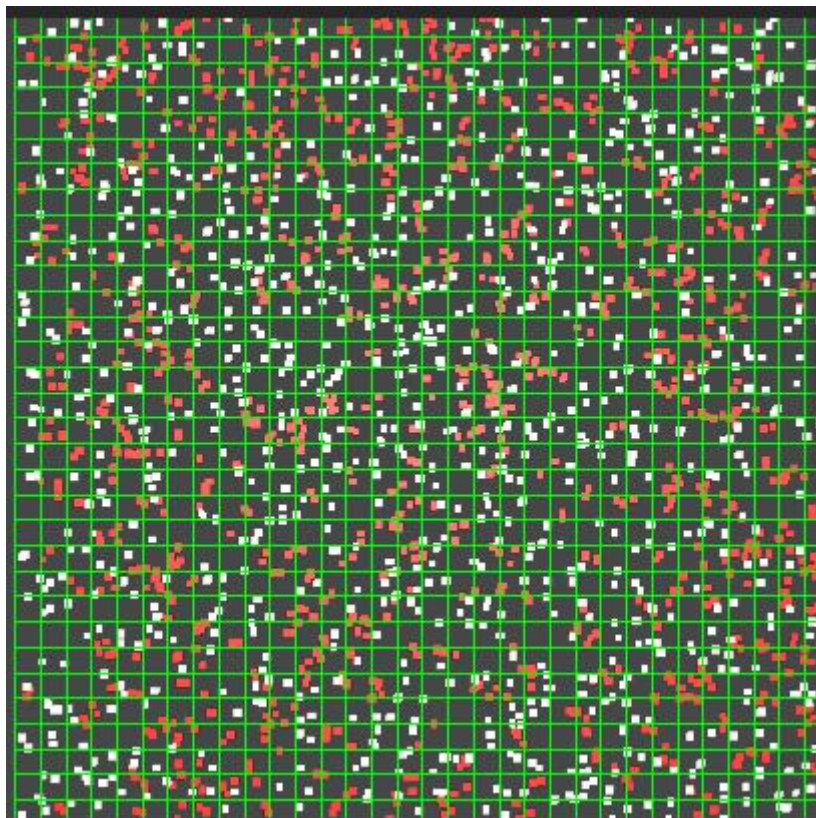


图 2-9 二维均匀网格的实例

### 2.2.3 BSP 的概念

BSP (Binary Space Partitioning, 空间二叉划分)也是一种空间数据结构，它可以对一个二维或三维空间进行划分，本文探讨的是二维空间的情形。每次将空间



---

划分为两个部分并对每个划分出的子空间递归重复这个过程，然后使用树结构将空间组合起来，最后得到的就是一棵二叉树，这棵二叉树的每个叶节点对应一块区域，每个非叶节点对应一次划分所用的分割线。这个道理其实很容易想到，因为在进行划分的过程中，所有非叶节点都经历了派生出子节点的过程，也就是说它们经历了“从叶节点变成了非叶节点”的过程，这个过程自然会让这个节点的角色发生改变：当一个节点还没有派生子节点时，它在这棵(尚未完成)的树上是一个叶节点，代表了一个区域；而当它派生出子节点后，它自己就变成了非叶节点，那么它现在就代表了一条分割线，其实很好理解，从一个原本的叶节点“派生子节点”对应的操作就是对原本节点代表的区域进行划分，在划分后这个节点就代表了分割线，而它派生的子节点现在就代表分出的两个子区域，当然，子节点也可以继续对自己进行上面这个操作。

一种简单的 **BSP** 划分方式是“轴向划分”，它每次划分都简单地在每个子空间的对称轴上进行划分，对于矩形场景而言，对称轴就是每个子空间水平或垂直的中线轴，而且这样划分出的子空间同样是矩形，对这些子空间也依然从对称轴上进行划分。通常来说，轴向划分最终形成一棵满二叉树：所有非叶节点的度都是 2，所有叶节点都在同一层次上，也就是说每一个同级的子区域都会进行划分，直到达到所要求的树深度。我们在划分区域时一般都采取“横竖交替”的策略(很好理解，有谁会一直竖着画呢？双缝干涉实验？)，如果一个节点是按照水平中线轴划分，那么它所派生的子节点再划分的时候就是按照垂直中线轴划分。

我们会发现，轴向划分最终得到的空间数据结构与网格无异。实际上，**BSP** 划分时可以使用任意位置、任意方向的分割线，轴向划分是 **BSP** 最简单的划分方式，而在很多实际应用当中，都会基于图元的形状大小以及位置等信息而使用更加灵活的分割线，以使得尽可能多的图元都只落在一个子区域当中，能够提升运算的性能。

#### 2.2.4 简单的研究与实现

##### 1、QT 的 BSPTree

我们先来看有名的图形界面框架 QT。QT 中的视图场景 `QGraphicsScene` 使用的是轴向划分 **BSP**：

```
1. void QGraphicsSceneBspTree::initialize(const QRectF &rect, int
    depth, int index)
2. {
3.     Node *node = &nodes[index];
```

---

```
4.     if (index == 0) {
5.         node->type = Node::Horizontal;
6.         node->offset = rect.center().y();
7.     }
8.     if (depth) {
9.         Node::Type type;
10.        QRectF rect1, rect2;
11.        qreal offset1, offset2;
12.        if (node->type == Node::Horizontal) {
13.            type = Node::Vertical;
14.            rect1.setRect(ax: rect.left(), ay: rect.top(), aaw: rect.w
                idth(), aah: rect.height() / 2);
15.            rect2.setRect(ax: rect1.left(), ay: rect1.bottom(), aaw: r
                ect1.width(), aah: rect.height() - rect1.height());
16.            offset1 = rect1.center().x();
17.            offset2 = rect2.center().x();
18.        } else {
19.            type = Node::Horizontal;
20.            rect1.setRect(ax: rect.left(), ay: rect.top(), aaw: rect.w
                idth() / 2, aah: rect.height());
21.            rect2.setRect(ax: rect1.right(), ay: rect1.top(), aaw: rec
                t.width() - rect1.width(), aah: rect1.height());
22.            offset1 = rect1.center().y();
23.            offset2 = rect2.center().y();
24.        }
25.        int childIndex = firstChildIndex(index);
26.        Node *child = &nodes[childIndex];
27.        child->offset = offset1;
28.        child->type = type;
29.        child = &nodes[childIndex + 1];
30.        child->offset = offset2;
31.        child->type = type;
32.        initialize(rect: rect1, depth: depth - 1, index: childIndex);
33.        initialize(rect: rect2, depth: depth - 1, index: childIndex +
            1);
34.    } else {
35.        node->type = Node::Leaf;
36.        node->leafIndex = leafCnt++;
37.    }
38. }
```

---

`nodes` 数组与 `index` 参数用于保存二叉树的信息，如何保存二叉树不是本文探讨的重点。看看这个二叉树是如何生成的：

由用户指定初始的 `rect` 和 `depth`，`rect` 参数是矩形对象用于指定 BSP 所要划分的区域，`depth` 参数用于指定二叉树的深度。二叉树节点 `Node` 为如下的数据结构：

```
1. struct Node
2. {
3.     enum Type { Horizontal, Vertical, Leaf };
4.     union {
5.         qreal offset;
6.         int leafIndex;
7.     };
8.     Type type;
9. };
```

这里使用了一个非常巧妙的设计：非叶节点对应的是分割线，但是我们并不需要保存“一条线”下来，由于使用轴向划分的 BSP，因此一条分割线只需要它的分割方向(水平/竖直)和它距离场景原点坐标轴的距离就可以确定位置，所以非叶节点保存的是 `type(Horizontal/Vertical)` 以及当前分割线与 `type` 对应坐标轴之间的距离 `offset`。叶节点的 `type` 是 `Leaf`，它对应一个区域，区域的信息同样不保存在 `Node` 本身当中，而是将每个区域中所包含的图元的信息保存在另一个单独的数组 `leaves` 中，叶节点的 `Node` 保存一个 `leafIndex` 作为 `leaves` 数组的下标，这样叶节点对应的区域就是 `leaves[leafIndex]`。由于一个节点只可能是叶节点或非叶节点之一，所以使用 `union` 联合 `offset` 和 `leafIndex`。

生成二叉树的过程递归调用 `initialize` 函数，递归调用的参数 `rect` 是本次调用后切出的子区域，`depth` 每次递归都减 1。在函数中判断 `depth`，当 `depth` 为 0 时表示已经到达目标深度，则当前的节点为叶节点，`type` 设为 `Leaf` 且记录 `leafIndex`；否则当前的节点为非叶节点，并且根据当前节点的 `type` 和 `rect` 计算出下层节点 (`child`) 在递归调用时的参数：下层节点的 `type` 应与当前节点的相反，下层节点的 `rect` 是当前分割线所分出的两个子区域。

## 2、BSP

这里的 BSP 就是在吸收 QT 的经验以后自己实现的 BSP 结构了。相关思路同上，不再赘述。

BSP 的数据结构为 `Node` 所串联成的二叉树。`Node` 的结构如下：

```
1. /**
2.  * @enum SplitType
```

```
3.  * @brief 枚举区域分割的类型。
4.  */
5.  enum SplitType
6.  {
7.      Horizontal = 0, ///< 水平分割
8.      Vertical,      ///< 竖直分割
9.      Leaf           ///< 叶子节点
10. };
11. /**
12.  * @class Node
13.  * @brief 树的节点, 对应分割形成的某块区域。
14.  */
15. struct Node
16. {
17.
18.     /**
19.      * @brief 默认构造。
20.      */
21.     Node() = default;
22.     /**
23.      * @brief 析构函数, 处理本节点以及子节点的释放。
24.      */
25.     ~Node();
26.     /**
27.      * @brief 该节点的分割类型。
28.      */
29.     SplitType m_splitType;
30.     /**
31.      * @details offset 和 leafIndex 分别对应非叶子节点和叶子节点的数据信息,
32.      * 对于同一个节点这两条数据不可能共存。因此为了节省内存, 采用 union
33.      */
34.     union
35.     {
36.         /**
37.          * @brief 分割线的横坐标或纵坐标的偏移量, 是横坐标还是纵坐标取决于分割类
38.          * 型。
39.          */
40.         int m_offset;
41.         /**
42.          * @brief 节点的下标, 在外部的 leaves 数组中使用。
43.          */
44.         int m_index;
```

```

42.     int m_leafIndex;
43. };
44. /**
45.  * @brief 左子节点指针。
46.  */
47. struct Node *m_left = nullptr;
48. /**
49.  * @brief 右子节点指针。
50.  */
51. struct Node *m_right = nullptr;
52. };
53. Node::~~Node()
54. {
55.     if (!m_left && !m_right) return;
56.
57.     delete m_left;
58.     m_left = nullptr;
59.
60.     delete m_right;
61.     m_right = nullptr;
62. }

```

BSP 包含的数据成员如下：

```

1.  /**
2.   * @brief 整棵 BSP 树的根节点。
3.   * @todo 后续考虑自己实现简单的对象树机制，不使用智能指针
4.   */
5.   Node *m_root = nullptr;
6.
7.  /**
8.   * @brief 存储每个叶子节点中的 PicItem (图元) 列表。
9.   * @details 树构建成功以后，所有的 PicItem 都存储在叶子节点的区域中，为了方便获取，将数据提取到整棵树的数据结构中，叶子节点中存储下标方便访问。注意，每个 "Vector<PicItem *>" 对应一个节点，因此 m_leaves 实际上以一维的方式组织各个节点。
10.  */
11.  Vector<Vector<PicItem *>> m_leaves;
12.
13.  /**
14.   * @brief 树的深度，对应分割的次数。
15.   */

```

```
16. int m_depth = 0;
17.
18. /**
19.  * @brief 整棵树作用的 2D 平面范围。
20.  */
21. Rect m_region;
```

由用户传入矩形区域 **region** 与树的深度 **depth**，程序传入初始根节点并递归调用 **init** 函数创建各个子节点。

注：本代码中 **Rect** 对象的 **x1()**和 **y1()**返回矩形对象左上角的坐标，**x2()**和 **y2()**返回矩形对象右下角的坐标，下同。

```
1. void init(Node *node, const Rect &region, int depth)
2. {
3.     // depth > 0, 继续向下分割
4.     if (depth > 0)
5.     {
6.         // 为了统一命名, 使用 left/right 对应逻辑上的 左/右 子节点
7.         // 水平 Horizontal : left 为上半边, right 为下半边
8.         // 垂直 Vertical : left 为左半边, right 为右半边
9.         int offsetLeft = 0, offsetRight = 0;
10.        Rect leftRect, rightRect;
11.        SplitType newSplit;
12.
13.        if (SplitType::Horizontal == node->m_splitType)
14.        {
15.            // 当前节点为水平分割 Horizontal, 则子节点为 Vertical, left 为上
            // 半边, right 为下半边
16.            newSplit = SplitType::Vertical;
17.            leftRect = Rect(region.x1(), region.y1(), region.width(),
                region.height() / 2);
18.            rightRect = Rect(leftRect.x1(), leftRect.y2(), region.wid
                th(), region.height() / 2);
19.            offsetLeft = leftRect.x1() + leftRect.width() / 2;
20.            offsetRight = rightRect.x1() + rightRect.width() / 2;
21.        }
22.        else
23.        {
24.            // 当前节点为垂直分割 Vertical, 则子节点为 Horizontal, left 为左
            // 半边, right 为右半边
25.            newSplit = SplitType::Horizontal;
```

```

26.     leftRect = Rect(region.x1(), region.y1(), region.width() /
    2, region.height());
27.     rightRect = Rect(leftRect.x2(), leftRect.y1(), region.wid
    th() / 2, region.height());
28.     offsetLeft = leftRect.y1() + leftRect.height() / 2;
29.     offsetRight = rightRect.y1() + rightRect.height() / 2;
30. }
31.
32.     node->m_left = new Node;
33.     node->m_left->m_splitType = newSplit;
34.     node->m_left->m_offset = offsetLeft;
35.
36.     node->m_right = new Node;
37.     node->m_right->m_splitType = newSplit;
38.     node->m_right->m_offset = offsetRight;
39.
40.     init(node->m_left, leftRect, depth - 1);
41.     init(node->m_right, rightRect, depth - 1);
42. }
43. // 遇到叶子节点
44. else
45. {
46.     node->m_splitType = SplitType::Leaf;
47.     node->m_leafIndex = m_leaves.size();
48.     m_leaves.append(Vector<PicItem *>());
49. }
50. }

```

查询是空间数据结构应当提供的基本用途，根据用户传入的区域查询当前空间数据结构中该传入区域命中的子区域(在 BSP 中，是叶节点)；在查询的基础上，用户或程序可以对命中的区域或对应区域中的图元执行操作，这个操作可以是在区域中增删图元、对区域中指定特征的图元进行进一步查询，等等等等。

```

1. /**
2.  * @brief 定义回调函数类型, 用于对叶节点执行操作
3.  */
4. using Visitor = std::function<void(LList<LCanvasItem *> &)>;
5.
6. // 以 addItem 为例
7. void addItem(LCanvasItem *item)
8. {

```

---

```
9.     auto func = [&item](LList<LCanvasItem *> &items)
10.     {
11.         items.append(item);
12.     };
13.
14.     update(func, m_root, item->boundingRect());
15. }
16.
17. /**
18.  * @brief 根据所给的区域查询命中的叶子节点, 并执行指定的操作。
19.  * @param visitor 函数对象。用于对查找到的叶子节点执行操作
20.  * @param node 根节点
21.  * @param rect 目标区域矩形
22.  */
23. void update(const Visitor &visitor, Node *node, const Rect &rect)
24. {
25.     if (m_leaves.isEmpty()) return;
26.
27.     switch (node->m_splitType)
28.     {
29.         case SplitType::Leaf:
30.             visitor(m_leaves[node->m_leafIndex]);
31.             break;
32.
33.         case SplitType::Vertical:
34.             {
35.                 if (rect.x1() < node->m_offset)
36.                 {
37.                     update(visitor, node->m_left, rect);
38.
39.                     if (rect.x2() >= node->m_offset) update(visitor, node->m
                        _right, rect);
40.                 }
41.                 else
42.                 {
43.                     update(visitor, node->m_right, rect);
44.                 }
45.
46.                 break;
47.             }
48.
```



```

49.     case SplitType::Horizontal:
50.     {
51.         if (rect.y1() < node->m_offset)
52.         {
53.             update(visitor, node->m_left, rect);
54.
55.             if (rect.y2() >= node->m_offset) update(visitor, node->m
                _right, rect);
56.         }
57.         else
58.         {
59.             update(visitor, node->m_right, rect);
60.         }
61.
62.         break;
63.     }
64. }
65. }

```

### 3、Grid

均匀网格的数据成员如下：

```

1.  /**
2.   * @brief 存储每个网格中的 PicItem 列表。
3.   * @details 树构建成功以后，所有的 PicItem 都存储在网格的区域中，为了方便获
      取，将数据提取到整个网格的数据结构中，通过数学计算得出下标方便访问。注意，每个
      "Vector<PicItem *>"对应一个网格，因此 m_grids 实际上以"一维数组排列二维网
      格"的方式组织各个网格。
4.   */
5.   Vector<Vector<PicItem *>> m_grids;
6.
7.  /**
8.   * @brief 网格分割出的每边的区间个数。
9.   */
10.  int m_sections = 0;
11.
12.  /**
13.   * @brief 整个网格作用的 2D 平面范围。
14.   */
15.  Rect m_region;

```

与 BSP 不同的是，BSP 有代表节点 Node 的“单元结构”，而均匀网格当中

没有设置“代表网格单元结构”Grid。这是基于两种空间数据结构的基本用途“查询”逻辑上的不同：BSP 树本身的结构和内容就决定了在查询时必须与非叶节点的分割线比较才能找到最终位于叶节点的子区域，树结构的特性就是不提供直接访问任何特定叶节点的快速途径，访问树中的指定节点必须从根节点开始，无法避开各个 Node 之间的逻辑结构(即使用数组保存它也是如此)，也因此 BSP 树中需要保存每条分割线的信息；而 Grid 本身的结构与数组对应，并且均匀划分使其在查询时能够通过数学计算的方式直接获知子区域的数组下标，因此就不需要专门去保存每个网格本身的信息，只需要将每个网格对应的图元列表保存为“图元列表的数组”作为查询时访问的目标。

均匀网格的成员函数通过用户传入的目标区域与分割线数量进行初始化：

```
1.  /**
2.   * @brief 带参构造。
3.   * @param region 需要作用的区域
4.   * @param splitNum 网格的分割线数量(经纬两个方向分割线数量相同)
5.   */
6.  Grid::Grid(const Rect &region, int splitNum) : m_region(region),
    m_sections(splitNum + 1), m_grids(Vector<Vector<PicItem *>>((splitNum + 1) * (splitNum + 1))) {}
```

均匀网格的 m\_grids 在初始化时规定好大小也就是网格数量，不过此时并不存储任何内容，因为初始化时还没有图元。与 BSP 不同的是，BSP 中 init 函数进行了划分区域、创建各个节点等操作，而均匀网格没有这一步骤。这也是因为 Grid 不需要保存每个网格的信息。

根据用户传入的区域查询当前空间数据结构中该传入区域命中的子区域(在均匀网格中，是数组下标)；在查询的基础上，用户或程序可以对命中的区域或对应区域中的图元执行操作，这个操作可以是在区域中增删图元、对区域中指定特征的图元进行进一步查询，等等等等。

```
1.  void update(const Visitor &visitor, const Rect &rect)
2.  {
3.      int x1Index = rect.x1() < m_region.x1() ? 0 : rect.x1() * m_sections / m_region.width();
4.      int y1Index = rect.y1() < m_region.y1() ? 0 : rect.y1() * m_sections / m_region.height();
5.      int x2Index = rect.x2() >= m_region.x2() ? (m_sections - 1) : rect.x2() * m_sections / m_region.width();
6.      int y2Index = rect.y2() >= m_region.y2() ? (m_sections - 1) : rect.y2() * m_sections / m_region.height();
```

---

```
7.  
8.    for (int y = y1Index; y <= y2Index; ++y)  
9.    {  
10.     for (int x = x1Index; x <= x2Index; ++x)  
11.     {  
12.         visitor(m_grids[y * m_sections + x]);  
13.     }  
14. }  
15. }
```

---

## 2.3 在 X11 下使用 Cairo 引擎绘制图形

cairo 是一个方便和高性能的第三方 C 库。它可以作为绘制引擎，帮助我们绘制各种图形，并且提供多种输出方式。这里开始将在 Linux 下结合 X11 图形显示协议绘制简单的图形。

这是效果图：

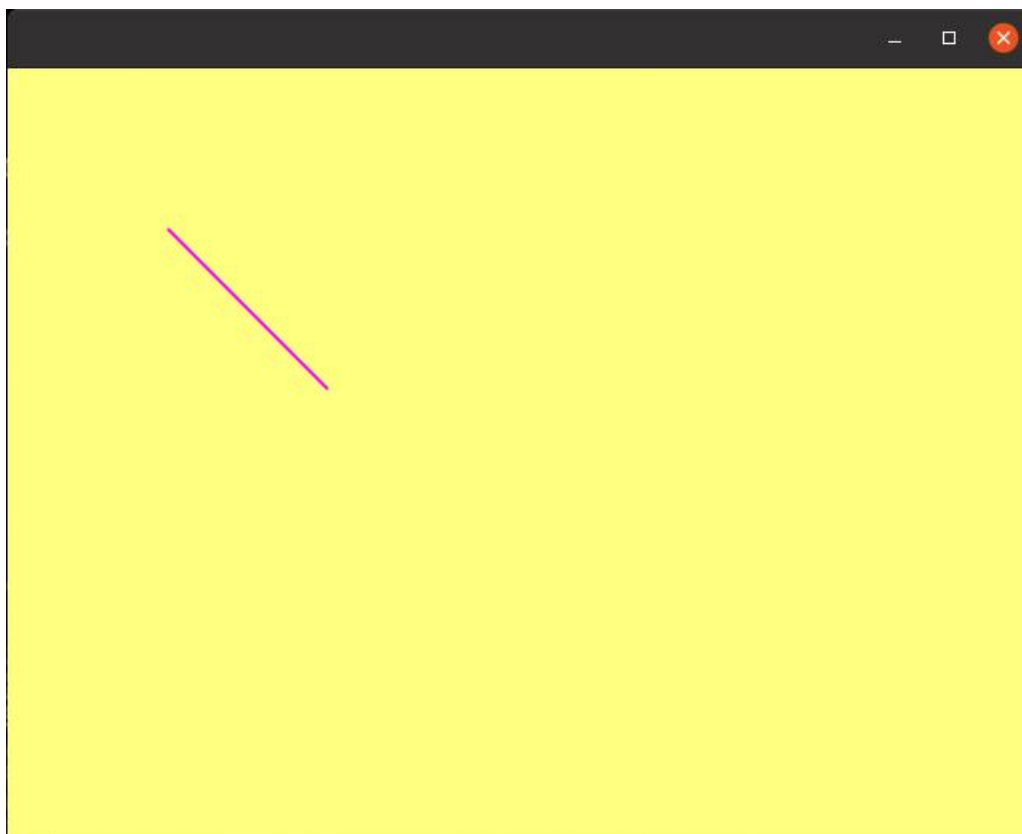


图 2-10 cairo 引擎绘制的示例图

### 2.3.1 安装 Cairo 库

cairo 的官方网站是 <https://cairographics.org>。上面对 cairo 图形库做了一个完整的介绍，包括如何下载、API 接口、示例等等。

#### 1、通过包管理器安装

通过官方文档知道，在 Linux 下可以直接通过包管理器进行下载，以 Ubuntu 为例。

```
1. sudo apt install libcairo2-dev
```

下载好以后头文件和动态库就安装好了。头文件安装在 `/usr/include/cairo/` 中，静态库和动态库分别位于 `/usr/lib/x86_64-linux-gnu/libcairo.a` 和 `/usr/lib/x86_64-linux-gnu/libcairo.so`。因此能直接被系统识别，直接引入头文件，编

---

译的时候链接 `cairo` 库即可。

## 2、通过 Conan 安装

对于个人而言，`apt` 安装自然是非常友好的。但是对于 LarkSDK 这样一个面向用户的基础框架而言，除非最基本的系统库例如 X 窗口系统 `libx11-dev`，Wayland 窗口系统 `libwayland-dev` 和 Wayland 键盘处理 `libxkbcommon-dev` 等，其他的第三方库均最好不以 `apt` 包的方式引入。鉴于 C++ 没有自己的包管理器，因此需要借助第三方的包管理器，例如 `conan`，`cpm` 等。本项目使用 `conan` 管理第三方包。

`conan` 是一个 `python` 语言编写的 C++ 的包管理器，官方网址是 <https://conan.io/>。`conan` 管理了众多第三方的 C++ 库，通过命令行操作就能方便的在自己的项目中引入 `conan` 包。需要通过 `conanfile.py` 或者 `conanfile.txt` 进行配置，当然这不是重点，具体见文档 <https://docs.conan.io/2/>。

`conan` 官方提供了自己的 `conan` 仓库，<https://conan.io/center>。在上面能找到很多我们熟知的第三方库，例如 `gtest`，`qt`，`boost`，`fmt` 等。当然还有一些更基础的工具库，这里不赘述。当然 `cairo` 库也在其中。

现在让我们尝试安装 `cairo` 库并尝试用 `CMake` 将其引入。在新开的项目中创建 `conanfile.txt`，引入项目依赖 `cairo`。使用 `conanfile.py` 主要用于生成和发布自己的 `conan` 包，`conan` 提供了丰富的选项供用户操作。这里只是为了测试，因此使用最简单的 `conanfile.txt` 即可。

```
1.  [requires]
2.  cairo/1.18.0
3.
4.  [generators]
5.  cmake
```

编写自己的 `CMakeLists.txt`，配置项目的相关信息，引入 `conan` 的部分类似如下：

```
1.  ...
2.
3.  include (${PROJECT_BINARY_DIR}/conanbuildinfo.cmake)
4.  conan_basic_setup (NO_OUTPUT_DIRS)
5.
6.  ...
7.
8.  add_executable (xxx
9.      ...
10. )
```

---

```
11. target_link_libraries (xxx ${CONAN_LIBS})
```

之后执行一般的构建流程即可。

```
1. conan install ..
```

```
2. cmake ..
```

```
3. make # windows 下默认没有 make 命令, 使用 cmake --build ./ 代替
```

conan 官方的包很多，很全，但是 conan 本身还有很多 bug，单就 cairo 包的使用过程中就有很多问题。最典型的，执行 `conan install ..` 可能会失败，并且遇到很多错误。因此我们需要了解 conan 包是个什么东西，才能明确问题是如何形成的。

现在对 conan 包做一个简述。对于 C++ 库而言，为了让用户方便的使用，把 .h 和 .cpp 代码全部打包出去是不合适的，这些代码不应该在用户的机器上再被编译一次，而应该在需要用的时候被直接使用。因此需要通过库的方式进行发布，也就是使用静态库和动态库。在发布的包中，最重要的文件就是头文件和库文件。当然可能会携带一些其他必要的文件，例如资源文件、版本说明文件等。

我们都知道，编译 C/C++ 的代码需要依赖于 C++ 和编译器的版本。进一步的，由于 C/C++ 是非常接近底层的代码，虽然标准库是跨平台的，但是如果需要写平台相关的程序，还需要注意操作系统的版本。因此，对于同一个版本的 conan 包，不同的系统，不同的编译环境，是静态库还是动态库，是 Debug 包还是 Release 包，甚至依赖包的版本，都可能对最后的编译造成一定影响。在本地的 conan 配置中会保存相关的这些信息，在 conan 拉包的时候会匹配本地的配置拉取合适的包。配置文件默认位于 `~/.conan/profile/default` 中。

在 Linux 下的配置类似于这样，看其参数很明显就能知道对应的含义。

```
1. [settings]
```

```
2. os=Linux
```

```
3. os_build=Linux
```

```
4. arch=x86_64
```

```
5. arch_build=x86_64
```

```
6. compiler=gcc
```

```
7. compiler.version=9
```

```
8. compiler.libcxx=libstdc++11
```

```
9. build_type=Release
```

```
10. [options]
```

```
11. [build_requires]
```

```
12. [env]
```

在 Windows 下类似于这样：

```
1. [settings]
```

```
2. os=Windows
```

```

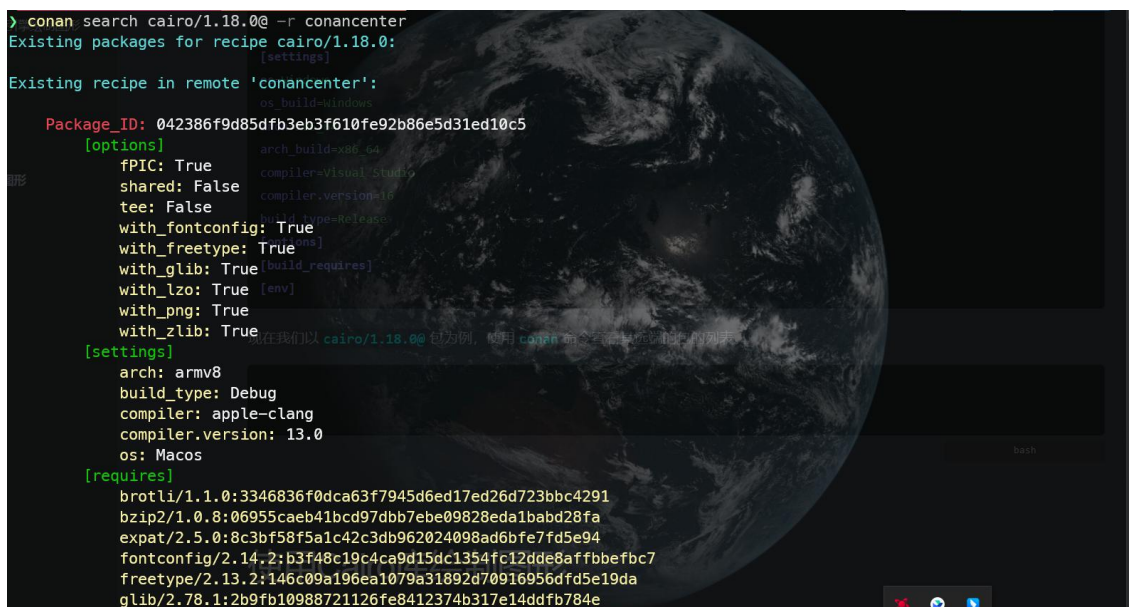
3.  os_build=Windows
4.  arch=x86_64
5.  arch_build=x86_64
6.  compiler=Visual 聽 Studio
7.  compiler.version=16
8.  build_type=Release
9.  [options]
10. [build_requires]
11. [env]

```

现在我们以 `cairo/1.18.0@`包为例，使用 `conan` 命令查看其远端的包的列表。

```
1.  conan search cairo/1.18.0@ -r conancenter
```

得到的结果大致是这样：



```

> conan search cairo/1.18.0@ -r conancenter
Existing packages for recipe cairo/1.18.0:
[settings]
Existing recipe in remote 'conancenter':
Package_ID: 042386f9d85dfb3eb3f610fe92b86e5d31ed10c5
[options]
  arch_build=x86_64
  fpic: True
  shared: False
  tee: False
  with_fontconfig: True
  with_freetype: True
  with_glib: True
  with_lzo: True
  with_png: True
  with_zlib: True
[settings]
  arch: armv8
  build_type: Debug
  compiler: apple-clang
  compiler.version: 13.0
  os: MacOS
[requires]
  brotli/1.1.0:3346836f0dca63f7945d6ed17ed26d723bbc4291
  bzip2/1.0.8:06955caeb41bcd97dbb7ebe09828eda1babb28fa
  expat/2.5.0:8c3bf58f5a1c42c3db962024098ad6bfe7fd5e94
  fontconfig/2.14.2:b3f48c19c4ca9d15dc1354fc12dde8affbbefbc7
  freetype/2.13.2:146c09a196ea1079a31892d70916956dfd5e19da
  glib/2.78.1:2b9fb10988721126fe8412374b317e14ddfb784e

```

图 2-11 cairo 的 conan 远端仓库的包列表

我们发现 settings 中的内容和我们的 profiles/default 中的内容对应，这就是前面提到的匹配。例如图中是一个 Mac 下的 apple-clang 的 13.0 版本的静态库的 Debug 包。每个包的 options, settings 和 requires 都会对最前面的 Package\_ID 产生影响，这是一个哈希计算值，具体如何影响和生成请参考 [https://docs.conan.io/2/reference/binary\\_model/package\\_id.html](https://docs.conan.io/2/reference/binary_model/package_id.html)。当然这其中也有令人费解的地方，后面会提到。

前面提到，conan 官方的包有问题，会导致在安装的时候出现错误。例如，我在安装的出现错误如下：



```
192.168.237.128
Decompressing conan_export.tgz completed [0.00k]
libpng/1.6.43: Downloaded recipe revision 0
WARN: libpng/1.6.43: requirement zlib/[>=1.2.11 <2] overridden by freetype/2.13.2 to zlib/1.3.1
bzip2/1.0.8: Retrieving from server 'conancenter'
bzip2/1.0.8: Trying with 'conancenter'...
Downloading conanmanifest.txt completed [0.17k]
Downloading conanfile.py completed [4.00k]
Downloading conan_export.tgz completed [0.31k]
Decompressing conan_export.tgz completed [0.00k]
bzip2/1.0.8: Downloaded recipe revision 0
brotli/1.1.0: Retrieving from server 'conancenter'
brotli/1.1.0: Trying with 'conancenter'...
Downloading conanmanifest.txt completed [0.18k]
Downloading conanfile.py completed [5.64k]
Downloading conan_export.tgz completed [0.40k]
Decompressing conan_export.tgz completed [0.00k]
brotli/1.1.0: Downloaded recipe revision 0
fontconfig/2.15.0: Retrieving from server 'conancenter'
fontconfig/2.15.0: Trying with 'conancenter'...
Downloading conanmanifest.txt completed [0.10k]
Downloading conanfile.py completed [4.80k]
Downloading conan_export.tgz completed [0.33k]
Decompressing conan_export.tgz completed [0.00k]
fontconfig/2.15.0: Downloaded recipe revision 0
ERROR: fontconfig/2.15.0: Cannot load recipe.
Error loading conanfile at '/home/lzx0626/.conan/data/fontconfig/2.15.0/_/_/export/conanfile.py': Current Conan version (1.60.1) does not satisfy the defined one (>=1.64.0 <2 || >=2.2.0).
```

图 2-12 conan 安装报错结果 1

错误信息告诉我 fontconfig 的 2.15.0 的版本需要 conan 1.60.4 以上才能安装。由于公司使用的是 conan 1.60.1，首先我想到的是 conan 版本不正确。深入研究后，我发现的问题出乎我的意料。

由于公司实际开发的 conan 版本不可能从 1.60.1 升到 1.60.4，要升肯定一步到位到 conan 2.0 了。首先查看 conancenter 中 cairo/1.18.0@远端的包，检索符合当前系统和编译环境的。我发现了三个长的几乎一样的包：

```
1.  ; package 1
2.  Package_ID: 703bcc640002869a53960c4449d3825ff8a103e6
3.  [options]
4.    fPIC: True
5.    shared: False
6.    tee: False
7.    with_fontconfig: True
8.    with_freetype: True
9.    with_glib: True
10.   with_lzo: True
11.   with_png: True
12.   with_symbol_lookup: False
13.   with_xcb: True
14.   with_xlib: True
15.   with_xlib_xrender: True
16.   with_zlib: True
17.  [settings]
```



```
18.     arch: x86_64
19.     build_type: Release
20.     compiler: gcc
21.     compiler.version: 9
22.     os: Linux
23.     [requires]
24.     brotli/1.1.0:b21556a366bf52552d3a00ce381b508d0563e081
25.     bzip2/1.0.8:da606cf731e334010b0bf6e85a2a6f891b9f36b0
26.     expat/2.6.0:c215f67ac7fc6a34d9d0fb90b0450016be569d86
27.     fontconfig/2.15.0:b172ac37518ca059ccac0be9c3eb29e5179ecf1
    e
28.     freetype/2.13.2:f1014dc4f9380132c471ceb778980949abf136d3
29.     glib/2.78.3:06c63123a2bb8b6d3ea5dcae501525df32efb7b5
30.     libelf/0.8.13:6af9cc7cb931c5ad942174fd7838eb655717c709
31.     libffi/3.4.4:6af9cc7cb931c5ad942174fd7838eb655717c709
32.     libmount/2.39:6af9cc7cb931c5ad942174fd7838eb655717c709
33.     libpng/1.6.43:7929d8ecf29c60d74fd3c1f6cb78bbb3cb49c0c7
34.     libselinux/3.5:6b0384e3aaa343ede5d2bd125e37a0198206de42
35.     lzo/2.10:6af9cc7cb931c5ad942174fd7838eb655717c709
36.     pcre2/10.42:647f8233073b10c84d51b1833c74f5a1cb8e8604
37.     pixman/0.43.4:6af9cc7cb931c5ad942174fd7838eb655717c709
38.     util-linux-libuuid/2.39.2:6af9cc7cb931c5ad942174fd7838eb6
    55717c709
39.     xorg/system:5ab84d6acfe1f23c4fae0ab88f26e3a396351ac9
40.     zlib/1.3.1:6af9cc7cb931c5ad942174fd7838eb655717c709
41.     Outdated from recipe: True
42.
43. ; package 2
44. Package_ID: 8098347825649d9fd3e21c49992446a2a2193ad4
45.     [options]
46.     fPIC: True
47.     shared: False
48.     tee: False
49.     with_fontconfig: True
50.     with_freetype: True
51.     with_glib: True
52.     with_lzo: True
53.     with_png: True
54.     with_symbol_lookup: False
55.     with_xcb: True
56.     with_xlib: True
```

---

```
57.     with_xlib_xrender: True
58.     with_zlib: True
59. [settings]
60.     arch: x86_64
61.     build_type: Release
62.     compiler: gcc
63.     compiler.version: 9
64.     os: Linux
65. [requires]
66.     brotli/1.1.0:b21556a366bf52552d3a00ce381b508d0563e081
67.     bzip2/1.0.8:da606cf731e334010b0bf6e85a2a6f891b9f36b0
68.     expat/2.5.0:c215f67ac7fc6a34d9d0fb90b0450016be569d86
69.     fontconfig/2.14.2:b172ac37518ca059ccac0be9c3eb29e5179ecf1
70.     freetype/2.13.0:f1014dc4f9380132c471ceb778980949abf136d3
71.     glib/2.78.0:06c63123a2bb8b6d3ea5dcae501525df32efb7b5
72.     libelf/0.8.13:6af9cc7cb931c5ad942174fd7838eb655717c709
73.     libffi/3.4.4:6af9cc7cb931c5ad942174fd7838eb655717c709
74.     libmount/2.39:6af9cc7cb931c5ad942174fd7838eb655717c709
75.     libpng/1.6.40:7929d8ecf29c60d74fd3c1f6cb78bbb3cb49c0c7
76.     libselinux/3.5:6b0384e3aaa343ede5d2bd125e37a0198206de42
77.     lzo/2.10:6af9cc7cb931c5ad942174fd7838eb655717c709
78.     pcre2/10.42:647f8233073b10c84d51b1833c74f5a1cb8e8604
79.     pixman/0.40.0:6af9cc7cb931c5ad942174fd7838eb655717c709
80.     util-linux-libuuid/2.39:6af9cc7cb931c5ad942174fd7838eb655
81.     xorg/system:5ab84d6acfe1f23c4fae0ab88f26e3a396351ac9
82.     zlib/1.3:6af9cc7cb931c5ad942174fd7838eb655717c709
83. Outdated from recipe: True
84.
85. ; package 3
86. Package_ID: a336bac291d8ec6a55c6257f3266f9a8760c7403
87. [options]
88.     fPIC: True
89.     shared: False
90.     tee: False
91.     with_fontconfig: True
92.     with_freetype: True
93.     with_glib: True
94.     with_lzo: True
95.     with_png: True
```

```
96.     with_symbol_lookup: False
97.     with_xcb: True
98.     with_xlib: True
99.     with_xlib_xrender: True
100.    with_zlib: True
101.    [settings]
102.    arch: x86_64
103.    build_type: Release
104.    compiler: gcc
105.    compiler.version: 9
106.    os: Linux
107.    [requires]
108.    brotli/1.1.0:b21556a366bf52552d3a00ce381b508d0563e081
109.    bzip2/1.0.8:da606cf731e334010b0bf6e85a2a6f891b9f36b0
110.    expat/2.5.0:c215f67ac7fc6a34d9d0fb90b0450016be569d86
111.    fontconfig/2.14.2:b172ac37518ca059ccac0be9c3eb29e5179ecf
    1e
112.    freetype/2.13.2:f1014dc4f9380132c471ceb778980949abf136d3
113.    glib/2.78.1:06c63123a2bb8b6d3ea5dcae501525df32efb7b5
114.    libelf/0.8.13:6af9cc7cb931c5ad942174fd7838eb655717c709
115.    libffi/3.4.4:6af9cc7cb931c5ad942174fd7838eb655717c709
116.    libmount/2.39:6af9cc7cb931c5ad942174fd7838eb655717c709
117.    libpng/1.6.40:7929d8ecf29c60d74fd3c1f6cb78bbb3cb49c0c7
118.    libselinux/3.5:6b0384e3aaa343ede5d2bd125e37a0198206de42
119.    lzo/2.10:6af9cc7cb931c5ad942174fd7838eb655717c709
120.    pcre2/10.42:647f8233073b10c84d51b1833c74f5a1cb8e8604
121.    pixman/0.42.2:6af9cc7cb931c5ad942174fd7838eb655717c709
122.    util-linux-libuuid/2.39.2:6af9cc7cb931c5ad942174fd7838eb
    655717c709
123.    xorg/system:5ab84d6acfe1f23c4fae0ab88f26e3a396351ac9
124.    zlib/1.3:6af9cc7cb931c5ad942174fd7838eb655717c709
125.    Outdated from recipe: True
```

这三个包的唯一区别在于 `requires` 不同，第一个包需要 `expat/2.6.0`，第二个和第三个包需要 `expat/2.5.0`。第二个包需要 `freetype/2.13.0`，第三个包 `freetype/2.13.2`。这就是唯一区别，然后生成了三个不同的哈希值，对应不同的 `package`。

那么问题来了，`conan install` 默认会读取本地的配置，但是本地配置不可能指定 `requires` 啊。不同的项目可能用的版本不同，这是很正常的事情，也不应该是由用户承担责任的地方。其次，`conan install` 不能指定 `package_id` 下载，因此下载的是哪一个包，完全就看他怎么想了。非常不幸的是，我需要第二个包，但是下

载只能下载到第一个包。

那么如何解决这个问题呢？幸运的是，conan 提供了 conan download 的方法，这个东西可以指定 package\_id 进行下载。那就简单了，先把 cairo 本包下载到本地，然后再指定这一套流程，由于本地有缓存，会跳过去远端拉取 cairo 包的这一步，这样所有本包和所有依赖不就顺利拉取下来了吗？说干就干。

```
1. conan download cairo/1.18.0@:8098347825649d9fd3e21c49992446a2a2193ad4 -r conancenter
```

成功下载下来以后，再次执行 conan install ..，又出问题了。

```
Decompressing conan_export.tgz completed [0.00k]
libpng/1.6.43: Downloaded recipe revision 0
WARN: libpng/1.6.43: requirement zlib/[>=1.2.11 <2] overridden by freetype/2.13.2 to zlib/1.3.1
bzip2/1.0.8: Retrieving from server 'conancenter'
bzip2/1.0.8: Trying with 'conancenter'...
Downloading conanmanifest.txt completed [0.17k]
Downloading conanfile.py completed [4.00k]
Downloading conan_export.tgz completed [0.31k]
Decompressing conan_export.tgz completed [0.00k]
bzip2/1.0.8: Downloaded recipe revision 0
brotli/1.1.0: Retrieving from server 'conancenter'
brotli/1.1.0: Trying with 'conancenter'...
Downloading conanmanifest.txt completed [0.18k]
Downloading conanfile.py completed [5.64k]
Downloading conan_export.tgz completed [0.40k]
Decompressing conan_export.tgz completed [0.00k]
brotli/1.1.0: Downloaded recipe revision 0
fontconfig/2.15.0: Retrieving from server 'conancenter'
fontconfig/2.15.0: Trying with 'conancenter'...
Downloading conanmanifest.txt completed [0.10k]
Downloading conanfile.py completed [4.80k]
Downloading conan_export.tgz completed [0.33k]
Decompressing conan_export.tgz completed [0.00k]
fontconfig/2.15.0: Downloaded recipe revision 0
ERROR: fontconfig/2.15.0: Cannot load recipe.
Error loading conanfile at '/home/lzx0626/.conan/data/fontconfig/2.15.0/_/_/export/conanfile.py': Current Conan version (1.60.1) does not satisfy the defined one (>=1.64.0 <2 || >=2.2.0).
```

图 2-12 conan 安装报错结果 2

不对啊，这个包依赖的 fontconfig 的版本是 2.14.2 啊，为什么这里还是下载的 2.15.0 啊？为了解决这个问题，我打开了对应的 conanfile.py，阅读到 requirements() 函数的时候，豁然开朗。

```
1. def requirements(self):
2.     self.requires("pixman/0.43.4")
3.     if self.options.with_zlib and self.options.with_png:
4.         self.requires("expat/[>=2.6.2 <3]")
5.     if self.options.with_lzo:
6.         self.requires("lzo/2.10")
7.     if self.options.with_zlib:
8.         self.requires("zlib/[>=1.2.11 <2]")
9.     if self.options.with_freetype:
10.        self.requires("freetype/2.13.2", transitive_headers=True,
            transitive_libs=True)
11.    if self.options.with_fontconfig:
```

```
12.     self.requires("fontconfig/2.15.0", transitive_headers=True, transitive_libs=True)
13.     if self.options.with_png:
14.         self.requires("libpng/[>=1.6 <2]")
15.     if self.options.with_glib:
16.         self.requires("glib/2.78.3")
17.     if self.settings.os in ["Linux", "FreeBSD"]:
18.         if self.options.with_xlib or self.options.with_xlib_xrender or self.options.with_xcb:
19.             self.requires("xorg/system", transitive_headers=True, transitive_libs=True)
20.         if self.options.get_safe("with_opengl") == "desktop":
21.             self.requires("opengl/system", transitive_headers=True, transitive_libs=True)
22.         if self.settings.os == "Windows":
23.             self.requires("glxext/cci.20210420")
24.             self.requires("wglxext/cci.20200813")
25.             self.requires("khrplatform/cci.20200529")
26.         if self.options.get_safe("with_opengl") and self.settings.os in ["Linux", "FreeBSD"]:
27.             self.requires("egl/system", transitive_headers=True, transitive_libs=True)
```

问题就出在这里, `expat` 需要的是 2.5.0, 但是规定依赖的包是 `>=2.6.2 and <3`。这不前后矛盾吗? 对于 `freetype` 和 `fontconfig` 也是相同的问题。同时, 我们再浏览一下 `cairo` 包拉下来以后的整体结构。噢, 原来同一个版本的所有包的 `conanfile.py` 都是同一个文件。好, 这没有任何问题, 这是 `conan` 的设计。但是不更新依赖版本的限制就很令人费解了, 如果硬要不改的话, 发布不同的版本也是 `ok` 的啊。这样一套流程下来, 导致 `conan` 拉包就出现了问题。

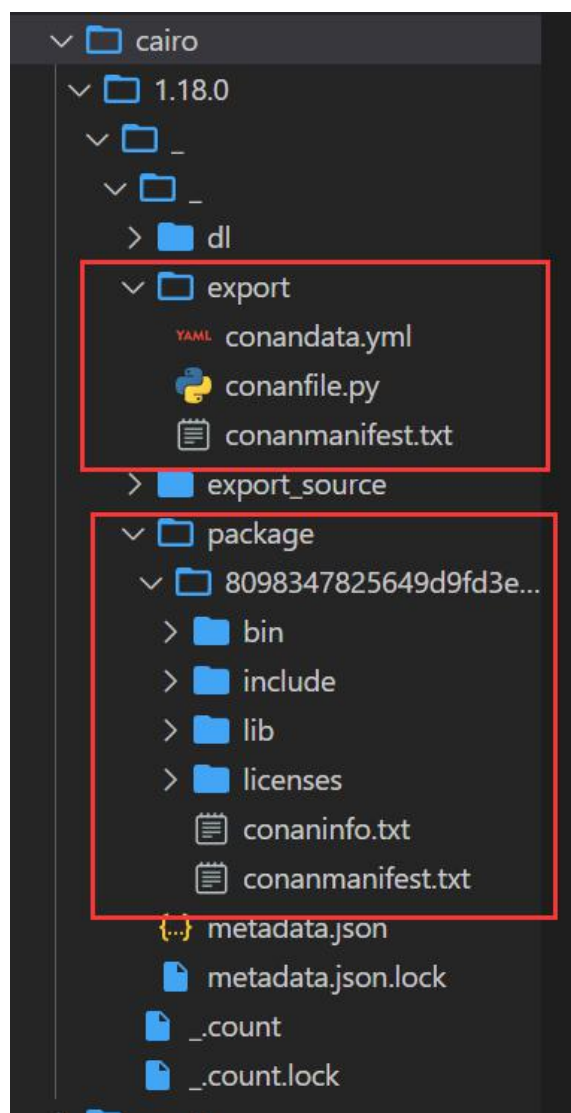


图 2-13 conan 包结构

那么如何解决呢？公司这边的解决方式是将该版本的 **cairo** 包上传到公司的 **conan** 服务器上，并手动修改 **conanfile.py** 使其版本匹配，并将所有的依赖以及穿透依赖全部拷贝到公司服务器上。至此，**cairo** 终于成功通过 **conan** 安装。

**Windows** 那边也是同样的问题，同样操作即可。

另外，在 **Linux** 下的 **Cairo** 包中，有一个依赖项值得看一下，即 **xorg/system**。这个包的版本是 **system**，意思是和系统相关的包，这里是 **Linux** 下 **X** 图形协议相关的依赖。众所周知，系统包是通过 **apt** 或者 **yum** 进行安装的，意思是 **conan** 能够调用这些包命令来帮我们自动安装对应的包吗？

答案是肯定的，参见文档 [https://docs.conan.io/2/reference/tools/system/package\\_manager.html](https://docs.conan.io/2/reference/tools/system/package_manager.html)。留意这段代码：

```
1. def system_requirements(self):
2.     apt = package_manager.Apt(self)
3.     apt.install(["libx11-dev", "libx11-xcb-dev", "libfontenc-dev",
4.                 "libice-dev", "libsm-dev", "libxau-dev", "libxaw7-dev",
5.                 "libxcomposite-dev", "libxcursor-dev", "libxdamage-dev",
6.                 "libxdmcp-dev", "libxext-dev", "libxfixedev",
7.                 "libxi-dev", "libxinerama-dev", "libxkbfile-dev", "libxmu-dev", "libxmuu-dev",
8.                 "libxpm-dev", "libxrandr-dev", "libxrender-dev", "libxres-dev", "libxss-dev", "libxt-dev", "libxtst-dev",
9.                 "libxv-dev", "libxxf86vm-dev", "libxcb-glx0-dev", "libxcb-render0-dev",
10.                "libxcb-render-util0-dev", "libxcb-xkb-dev", "libxcb-icccm4-dev", "libxcb-image0-dev",
11.                "libxcb-keysyms1-dev", "libxcb-randr0-dev", "libxcb-shape0-dev", "libxcb-sync-dev", "libxcb-xfixedev",
12.                "libxcb-xinerama0-dev", "libxcb-dri3-dev", "uuid-dev", "libxcb-cursor-dev", "libxcb-dri2-0-dev",
13.                "libxcb-dri3-dev", "libxcb-present-dev", "libxcb-composite0-dev", "libxcb-ewmh-dev",
14.                "libxcb-res0-dev"], update=True, check=True)
15.     apt.install_substitutes(
16.         ["libxcb-util-dev"], ["libxcb-util0-dev"], update=True, check=True)
17.     yum = package_manager.Yum(self)
18.     yum.install(["libxcb-devel", "libfontenc-devel", "libXaw-devel", "libXcomposite-devel",
19.                 "libXcursor-devel", "libXdmcp-devel", "libXtst-devel", "libXinerama-devel",
20.                 "libXkbfile-devel", "libXrandr-devel", "libXres-devel", "libXScrnSaver-devel",
21.                 "xcb-util-wm-devel", "xcb-util-image-devel", "xcb-util-keysyms-devel",
22.                 "xcb-util-renderutil-devel", "libXdamage-devel", "libXxf86vm-devel", "libXv-devel",
23.                 "xcb-util-devel", "libuuid-devel", "xcb-util-cursor-devel"], update=True, check=True)
24.     dnf = package_manager.Dnf(self)
```



---

```
25. dnf.install(["libxcb-devel", "libfontenc-devel", "libXaw-devel", "libXcomposite-devel",
26.             "libXcursor-devel", "libXdmcp-devel", "libXtst-devel", "libXinerama-devel",
27.             "libxkbfile-devel", "libXrandr-devel", "libXres-devel", "libXScrnSaver-devel",
28.             "xcb-util-wm-devel", "xcb-util-image-devel", "xcb-util-keysyms-devel",
29.             "xcb-util-renderutil-devel", "libXdamage-devel", "libXxf86vm-devel", "libXv-devel",
30.             "xcb-util-devel", "libuuid-devel", "xcb-util-cursor-devel"], update=True, check=True)
31.
32. zypper = package_manager.Zypper(self)
33. zypper.install(["libxcb-devel", "libfontenc-devel", "libXaw-devel", "libXcomposite-devel",
34.                "libXcursor-devel", "libXdmcp-devel", "libXtst-devel", "libXinerama-devel",
35.                "libxkbfile-devel", "libXrandr-devel", "libXres-devel", "libXss-devel",
36.                "xcb-util-wm-devel", "xcb-util-image-devel", "xcb-util-keysyms-devel",
37.                "xcb-util-renderutil-devel", "libXdamage-devel", "libXxf86vm-devel", "libXv-devel",
38.                "xcb-util-devel", "libuuid-devel", "xcb-util-cursor-devel"], update=True, check=True)
39.
40. pacman = package_manager.PacMan(self)
41. pacman.install(["libxcb", "libfontenc", "libice", "libsm", "libxaw", "libxcomposite", "libxcursor",
42.                "libxdamage", "libxdmcp", "libxtst", "libxinerama", "libxkbfile", "libxrandr", "libxres",
43.                "libxss", "xcb-util-wm", "xcb-util-image", "xcb-util-keysyms", "xcb-util-renderutil",
44.                "libxxf86vm", "libxv", "xcb-util", "util-linux-libs", "xcb-util-cursor"], update=True, check=True)
45.
46. package_manager.Pkg(self).install(["libX11", "libfontenc", "libice", "libsm", "libxaw", "libxcomposite", "libxcursor",
47.                                     "libxdamage", "libxdmcp", "libxtst", "libxinerama", "libxkbfile", "libxrandr", "libxres",
```



```

48.         "libXScrnSaver", "xcb-util-wm", "xcb-util-image", "x
    cb-util-keysyms", "xcb-util-renderutil",
49.         "libxxf86vm", "libxv", "xkeyboard-config", "xcb-util
    ", "xcb-util-cursor"], update=True, check=True)

```

对于 Ubuntu 来讲, conan 能手动帮我们调用 apt 安装所需要的系统依赖。需要注意一点, 需要在配置文件中加上两句, 来指明开启这个功能和使用 sudo。所以最终的配置文件类似如下:

```

1. [settings]
2. os=Linux
3. os_build=Linux
4. arch=x86_64
5. arch_build=x86_64
6. compiler=gcc
7. compiler.version=9
8. compiler.libcxx=libstdc++11
9. build_type=Release
10. [options]
11. [build_requires]
12. [env]
13. [conf]
14. tools.system.package_manager:mode=install
15. tools.system.package_manager:sudo=True

```

至此, 我们成功通过 conan 安装下来了 cairo, 看到成功的结果, 我的内心无比兴奋。

```

Installing (downloading, building) binaries...
brotli/1.1.0: Already installed!
bzip2/1.0.8: Already installed!
expat/2.5.0: Already installed!
libelf/0.8.13: Already installed!
libffi/3.4.4: Already installed!
libmount/2.39: Already installed!
lzo/2.10: Already installed!
pike/0.40.0: Already installed!
util-linux-libuuid/2.39: Already installed!
xorg/system: Already installed!
zlib/1.3: Already installed!
libpng/1.6.40: Already installed!
pcre2/10.42: Already installed!
pcre2/10.42: Appending PATH environment variable: /home/lzx0626/.conan/data/pcre2/10.42/_/_/package/b359196ca5d3d5c89f5f661c25b5a6d70eb2c315/bin
freetype/2.13.0: Already installed!
libselinux/3.5: Already installed!
fontconfig/2.14.2: Already installed!
glib/2.78.0: Already installed!
cairo/1.18.0: Already installed!
conanfile.txt: Generator txt created conanbuildinfo.txt
conanfile.txt: Generator cmake created conanbuildinfo.cmake
conanfile.txt: Aggregating env generators
conanfile.txt: Generated conaninfo.txt
conanfile.txt: Generated graphinfo
使用cairo库绘制图形
~/Lark5/cairo-research/build on P main
lzx0626@DavidingPlus $
took 5s at 17:58:28

```

图 2-14 安装成果结果

---

### 2.3.2 使用 Cairo 库绘制图形

安装完 cairo 库，是时候使用它绘制一些简单的图形了。

#### 1、Linux GUI 背景简述

Linux 本身是不带有图形界面的，真正原生的 Linux 系统只是一个基于命令行的操作系统。但是我们目前所使用的 Linux 发行版，例如 Ubuntu、Centos 等，都是有图形界面的啊。这是因为这些图形界面是 Linux 下的一个应用程序而已，是通过程序和协议模拟和实现出来的，或者说图形界面并不属于 Linux 内核的一部分。这一点和 Windows 系统完全不一样，Windows 的命令行在我看来完全不如 Linux 这么好用，甚至有时候我会爆粗口喷他，但是 Windows 的用户依然最多。为什么呢？就是因为 GUI 好看。Windows 的图形界面是操作系统的一部分，并且做的确实好看和丝滑。这样变相的降低了用户的学习和使用成本，对大多数的人而言是件好事。但是对于程序员特别是偏向底层的程序员来讲，却真不一定。

在 Linux 下，需要通过应用程序实现图形界面，那就需要设计一个合适的协议。目前市面上比较流行的有两种协议，X 协议和 Wayland 协议。这两种协议都是基于网络通信的思想，将图形显示分为了客户端（即你的应用程序）和服务端通信的过程。输入设备和显示设备不是同一个设备，而且他们需要相互配合，进行画面显示，所以需要一个交互协议，建立他们直接的沟通桥梁。当然 X 协议和 Wayland 协议的细节有所区别，粗略的讲是 server 和 compositor 的设计不同，具体可见 <https://www.secjuice.com/wayland-vs-xorg/>。

本文以及后续以 X 协议为例展开，并以 XClient 和 XServer 分别代指客户端和服务端。例如现在我需要画一个圆，XClient 需要告诉 XServer 在屏幕的什么地方，使用什么颜色，画多大的一个圆。至于这个圆如何生成，如何使用硬件真正绘制图形等等这些操作，都是由 XServer 完成的。当然更进一步的，XServer 还可以捕捉鼠标和键盘的动作，会触发相应的事件。XClient 可以接受相应的事件并且完成相应的逻辑。这就是整个 X 协议以及绘制逻辑的简要概括。

X 协议有很多实现。目前用的最多的是 XOrg，对应的 XClient 有 Xlib 和 XCB 的两种实现，提供了和 XServer 对接的 API。（At the bottom level of the X client library stack are Xlib and XCB, two helper libraries (really sets of libraries) that provide API for talking to the X server.）本文的背景是 X11，也是 Xlib 库的一个特殊版本。

#### 2、安装 X11 并编写 GUI 程序

强烈建议在虚拟机下进行，因为 Wsl 需要内核更新到 2.2.4 以后才能使用最新

---

功能的 GUI，而且体验还不是很很好。

以 Ubuntu 为例，系统默认是未安装 X11 库的（XServer 肯定是有，不然你怎么看得到图形界面呢？安装 X11 只是提供了窗口系统的开发支持）。因此我们使用如下命令手动安装。值得一提的是，由于 X11 是系统库，因此使用 apt 包安装即可。同理对于 Wayland 也是一样。

```
1. sudo apt install libx11-dev
```

安装完成以后，使用 CMake 就能很方便的引入 X11 支持了。

```
1. add_executable (xxx
2. ...
3. )
4. target_link_libraries (xxx X11)
```

现在我们编写一个样例程序，用于在 X11 下绘制一个图形窗口。这里面涉及到很多的 X11 的 API，本文只做简单介绍，具体请参考文档 <https://www.x.org/releases/X11R7.7/doc/libX11/libX11/libX11.html>

```
1. #include <X11/Xlib.h>
2.
3. #include <iostream>
4.
5.
6. int main()
7. {
8.     // 用于和 XServer 建立连接, dpy 指针全局只有一份
9.     Display *dpy = XOpenDisplay(nullptr);
10.    if (!dpy)
11.    {
12.        std::cerr << "Unable to open X display!" << std::endl;
13.        throw;
14.    }
15.
16.    int screen = DefaultScreen(dpy);
17.    // 创建一个顶层窗口
18.    Window w = XCreateSimpleWindow(
19.        dpy,
20.        RootWindow(dpy, DefaultScreen(dpy)),
21.        100, 100, 400, 300, 0,
22.        BlackPixel(dpy, screen),
23.        BlackPixel(dpy, screen)
24.    );
```

```
25. // 将需要检测的事件绑定在窗口上
26. XSelectInput(dpy, w, ExposureMask);
27. // 展示这个窗口
28. XMapWindow(dpy, w);
29.
30. std::cout << "Entering loop ..." << std::endl;
31.
32. // 进入事件循环
33. XEvent e;
34. while (1)
35. {
36.     XNextEvent(dpy, &e);
37.     switch (e.type)
38.     {
39.         case Expose:
40.             std::cout << "event: Expose" << std::endl;
41.             break;
42.         default:
43.             std::cout << "event: " << e.type << std::endl;
44.             break;
45.     }
46. }
47.
48. return 0;
49. }
```

接口的大致功能以在代码注释中体现。注意到整个程序最重要的架构是最后的这个事件循环,当 XClient 窗口成功创建出来以后,XServer 需要不断监听 XClient 发送的事件并予以处理,这样才能实现 GUI 程序的功能。因此事件循环的存在就自然而然了。有点类似于 IO 多路复用中 epoll 技术的框架。在这里监听的是 Expose 事件,不用具体关心这个语义是什么意思,在窗口创建和窗口大小发生改变的时候会触发 Expose 事件,这里也是用作信息打印测试。

程序的运行结果如下,可以发现窗口创建和改变窗口大小的时候在不断打印 Expose 的信息。

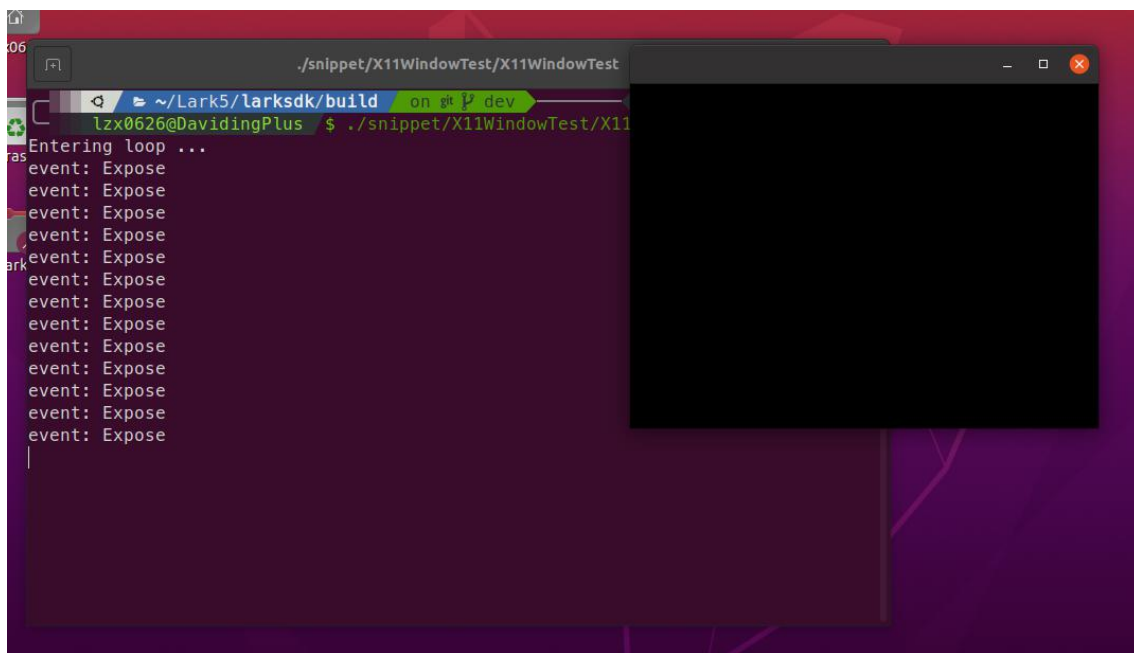


图 2-15 样例程序运行结果

对于事件循环这个概念，不管是 Qt，还是 LarkSDK，还是对于一个跨平台的 GUI 框架，事件循环显然是必不可少的。问题在于跨平台需要统一不同平台下的窗体系统和事件处理等逻辑，最终抽象出跨平台的接口，这就是这些框架正在做最重要的一件事情。以 LarkSDK 为例，虽然最简单的跨平台的程序是四行就能搞定，但是其中涉及到的知识和背景是非常庞大的。

```
1. #include <lwindowapplication.h>
2. #include <lwindow.h>
3.
4. int main()
5. {
6.     LWindowApplication app; // 创建窗体程序主框架实例
7.     LWindow w; // 创建顶层窗体
8.     w.show(); // 让窗体可见
9.     return app.exec(); // 进入主事件循环
10. }
```

### 3、尝试引入 Cairo

cairo 将输出和绘制的概念做了严格区分。cairo surface 是一个抽象出来的概念，与其对接的是多种输出方式，例如 PDF、PNG、SVG、Win32、XLib、XCB 等，如图所示。

---

## Surfaces

- `cairo_device_t` — interface to underlying rendering system
- `cairo_surface_t` — Base class for surfaces
- Image Surfaces — Rendering to memory buffers
- PDF Surfaces — Rendering PDF documents
- PNG Support — Reading and writing PNG images
- PostScript Surfaces — Rendering PostScript documents
- Recording Surfaces — Records all drawing operations
- Win32 Surfaces — Microsoft Windows surface support
- SVG Surfaces — Rendering SVG documents
- Quartz Surfaces — Rendering to Quartz surfaces
- XCB Surfaces — X Window System rendering using the XCB library
- XLib Surfaces — X Window System rendering using XLib
- XLib-XRender Backend — X Window System rendering using XLib and the X Render extension
- Script Surfaces — Rendering to replayable scripts
- Surface Observer — Observing other surfaces
- Tee surface — Redirect input to multiple surfaces

图 2-16 cairo surface 概览

接着我们查看 `cairo` 官方提供的 `samples`，可以发现，官方提供的样例好像完全和 `surface` 没有关系，换句话说没有反映输出的方式。例如这段代码：

```
1. double xc = 128.0;
2. double yc = 128.0;
3. double radius = 100.0;
4. double angle1 = 45.0 * (M_PI/180.0); /* angles are specified */
5. double angle2 = 180.0 * (M_PI/180.0); /* in radians */
6.
7. cairo_set_line_width (cr, 10.0);
8. cairo_arc (cr, xc, yc, radius, angle1, angle2);
9. cairo_stroke (cr);
10.
11. /* draw helping lines */
12. cairo_set_source_rgba (cr, 1, 0.2, 0.2, 0.6);
13. cairo_set_line_width (cr, 6.0);
14.
15. cairo_arc (cr, xc, yc, 10.0, 0, 2*M_PI);
16. cairo_fill (cr);
17.
18. cairo_arc (cr, xc, yc, radius, angle1, angle1);
19. cairo_line_to (cr, xc, yc);
20. cairo_arc (cr, xc, yc, radius, angle2, angle2);
21. cairo_line_to (cr, xc, yc);
```

---

22. `cairo_stroke(cr);`

它的绘制结果是这样的:

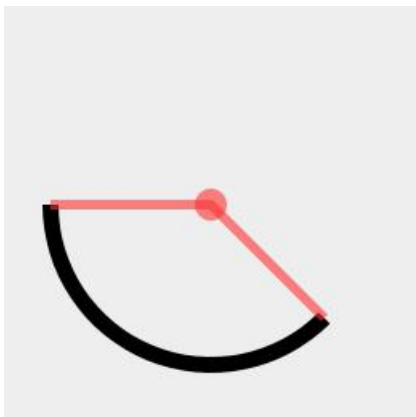


图 2-17 样例绘制效果图

这个图样可以被输出到前面提到的任意一种 `surface` 中。同时这也是我想说的, `cairo` 将输出和绘制的概念做了完整区分, 同样这也是我们容易想到和愿意看到的。所以, 如果想要创建一个输出到 PNG 中的实例, 代码应该类似如下:

```
1. // 创建 surface
2. cairo_surface_t *surface = cairo_image_surface_create(CAIRO_FORMAT_ARGB32, 640, 480);
3. // 创建绘图上下文 context
4. cairo_t *cr = cairo_create(surface);
5.
6. // 绘制逻辑, 只和 context 相关, 与 surface 无关
7. ...
8.
9. // 输出到 png 格式中
10. cairo_surface_write_to_png(surface, "xxx.png");
```

至此, 我们用 `surface` 和 `context` 来代指输出和绘制的概念, 也明白了 `cairo` 是如何区分这两个概念的了。 `surface` 用作指明绘制的图形输出到哪里, `context` 则用于进行绘制。读者可以尝试编写一个完整的程序输出上面的图案到一张图片文件中。

#### 4、Cairo 与 X11 相结合

使用 `cairo` 成功输出到图片文件中以后, 试着想想如何与 X11 窗口系统结合了。提前声明, 用到的 `surface` 只有 `XLib Surface` 和 `Image Surface`, 其他的 API 请自行查询文档。

首先想到的肯定是 `XLib Surface`。这代表 `cairo` 帮我做好了与 X11 平台的对接

---

工作，我只需要按部就班地使用 `cairo` 的 API 即可。所有的 `surface` 基本上只有在创建的时候会有区别，在 `context` 那一层的绘制几乎没有区别。例如下面就是 `XLib Surface` 的创建 API，参数具体含义请参考官方文档。

```
1.  cairo_surface_t *
2.  cairo_xlib_surface_create (Display *dpy,
3.                             Drawable drawable,
4.                             Visual *visual,
5.                             int width,
6.                             int height);
```

查询该方法需要的接口如何获取以后，编写出如下的代码：

```
1.  #include <iostream>
2.  #include <exception>
3.
4.  #include <X11/Xlib.h>
5.
6.  #include "cairo.h"
7.  #include "cairo/cairo-xlib.h"
8.
9.
10. int main()
11. {
12.     Display *dpy = XOpenDisplay(nullptr);
13.     if (!dpy)
14.     {
15.         throw std::runtime_error("Failed to open X display");
16.     }
17.
18.     int screen = DefaultScreen(dpy);
19.     Window w = XCreateSimpleWindow(
20.         dpy,
21.         RootWindow(dpy, DefaultScreen(dpy)),
22.         100, 100, 640, 480, 0,
23.         BlackPixel(dpy, screen),
24.         BlackPixel(dpy, screen));
25.     XSelectInput(dpy, w, ExposureMask);
26.
27.     XMapWindow(dpy, w);
28.
29.     std::cout << "Entering loop ..." << std::endl;
```



```
30.
31. // 根据 window id 获取该窗口的信息
32. XWindowAttributes attr;
33. XGetWindowAttributes(dpy, w, &attr);
34.
35. // 创建 XLib Surface
36. cairo_surface_t *surface = cairo_xlib_surface_create(dpy, w,
    attr.visual, attr.width, attr.height);
37. cairo_t *cr = cairo_create(surface);
38.
39. XEvent e;
40. while (true)
41. {
42.     XNextEvent(dpy, &e);
43.
44.     switch (e.type)
45.     {
46.         case Expose:
47.         {
48.             std::cout << "event: Expose" << std::endl;
49.
50.             // 绘制操作
51.             cairo_set_source_rgb(cr, 1.0, 1.0, 0.5);
52.             cairo_paint(cr);
53.
54.             cairo_set_source_rgb(cr, 1.0, 0.0, 1.0);
55.             cairo_move_to(cr, 100, 100);
56.             cairo_line_to(cr, 200, 200);
57.             cairo_stroke(cr);
58.
59.             break;
60.         }
61.         default:
62.             std::cout << "event: " << e.type << std::endl;
63.             break;
64.     }
65. }
66.
67. cairo_destroy(cr);
68. cairo_surface_destroy(surface);
69.
```

```
70.  XDestroyWindow(dpy, w);
71.  XCloseDisplay(dpy);
72.
73.
74.  return 0;
75. }
```

这样能在 `Expose` 事件触发的时候成功绘制出文章开始时候展示的图样。

`XLib Surface` 的代码结构看着很像自动挡的感觉，创建 `surface`，在事件循环中用 `context` 进行绘制，最终得到想要的图案。

我们深入思考一下，图形是如何被绘制到屏幕上的呢？前面举了个例子，画一个圆，告诉 `XServer` 在哪里，用什么颜色，画多大、多宽的圆。至于如何用硬件画不是 `XClient` 关心的事情，但是如何表示这些信息呢？显然需要用合适的 `data` 进行存储。进一步讲，`context` 调用各种方法的实际过程，其实就是往数据缓冲区 `data` 中写数据的过程。当类似 `flush` 操作的被调用以后，这些数据才会真正反映在屏幕上，形成我们观看的效果。

在这样的语义下，我们进一步思考 `surface` 的概念，其实用可绘制表面的概念好像更加贴切（本概念借鉴于 `LarkSDK` 的 `LSurface`）。绘图的数据缓冲区记录了图形的数据，类型是 `unsigned char *`，这些数据和不同的输出方式对接就能达到不同的输出效果。至于为什么是 `unsigned char *`（我猜测大概是字节流）以及如何对接，这不是重点，有兴趣请自己查询资料。

知道这个过程以后，回到 `Image Surface` 本身，为什么要使用这个东西，是因为它为我们提供了获取数据的接口。也就是当我用 `context` 绘制以后，调用这个方法就能立刻拿到缓冲区的数据。

```
1.  unsigned char *
2.  cairo_image_surface_get_data (cairo_surface_t *surface);
```

然后让我们思考一下绘制效率。不同引擎的效率的区别根本上就是在于如何快速的把这些数据计算出来，或者换句话讲，如何快速地把缓冲区的内存填充为指定的数据。比如对于最基本的暴力软渲染和 `cairo` 引擎，他们的效率差距显然是非常大的。这里有一个例子可以参考，是 `LarkSDK` 原生软渲染和 `cairo` 引擎同样绘制 10000 条斜线的效率差距，以下是结果，保守估计至少差了几百到一千倍。

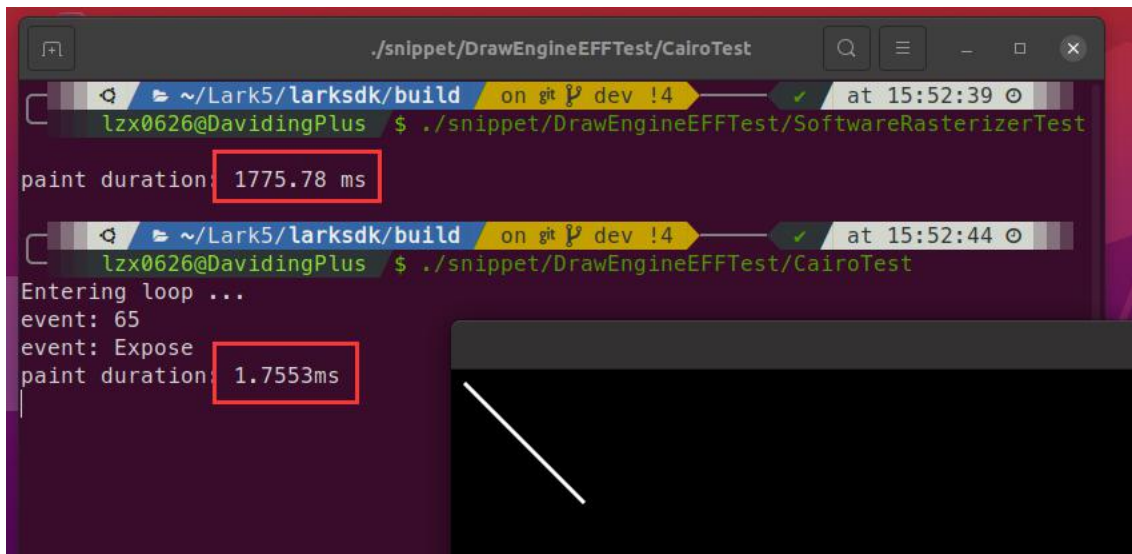


图 2-18 绘制效率对比图

回到正题，再结合 X11 的 API，我们可以给出使用 Image Surface 的代码：

```
1. #include <iostream>
2. #include <exception>
3.
4. #include <X11/Xlib.h>
5.
6. #include "cairo.h"
7.
8.
9. int main()
10. {
11.     Display *dpy = XOpenDisplay(nullptr);
12.     if (!dpy)
13.     {
14.         throw std::runtime_error("Failed to open X display");
15.     }
16.
17.     int screen = DefaultScreen(dpy);
18.     Window w = XCreateSimpleWindow(
19.         dpy,
20.         RootWindow(dpy, DefaultScreen(dpy)),
21.         100, 100, 640, 480, 0,
22.         BlackPixel(dpy, screen),
23.         BlackPixel(dpy, screen));
24.     XSelectInput(dpy, w, ExposureMask);
```

```
25.
26.     unsigned long mask = 0;
27.     XGCValues values;
28.     GC gc = XCreateGC(dpy, w, mask, &values);
29.
30.     XMapWindow(dpy, w);
31.
32.     // 创建 Image Surface
33.     cairo_surface_t *surface = cairo_image_surface_create(CAIRO_
        FORMAT_ARGB32, 640, 480);
34.     cairo_t *cr = cairo_create(surface);
35.
36.     // 获得 Cairo 管理的绘图数据的指针
37.     unsigned char *pData = cairo_image_surface_get_data(surface);
38.     // 创建 X11 下的 Image Buffer, 将其中的数据替换为 Cairo 的数据指针
39.     XImage *pBackBuffer = XCreateImage(
40.         dpy,
41.         DefaultVisual(dpy, screen),
42.         DefaultDepth(dpy, screen),
43.         ZPixmap,
44.         0,
45.         (char *)pData,
46.         640, 480,
47.         8,
48.         0);
49.
50.     std::cout << "Entering loop ..." << std::endl;
51.
52.     XEvent e;
53.     while (true)
54.     {
55.         XNextEvent(dpy, &e);
56.
57.         switch (e.type)
58.         {
59.             case Expose:
60.             {
61.                 std::cout << "event: Expose" << std::endl;
62.
63.                 cairo_set_source_rgb(cr, 1.0, 1.0, 0.5);
64.                 cairo_paint(cr);
```

```
65.
66.     cairo_set_source_rgb(cr, 1.0, 0.0, 1.0);
67.     cairo_move_to(cr, 100, 100);
68.     cairo_line_to(cr, 200, 200);
69.     cairo_stroke(cr);
70.
71.     // flush 操作, 刷新缓冲区, 更新数据
72.     cairo_surface_flush(surface);
73.
74.     // X11 下真正绘制图形的方法, 用到了外面定义的 X11 Image Buffer, 而
       其内部的数据就是 Cairo 管理的缓冲区数据
75.     XPutImage(
76.         dpy,
77.         w,
78.         gc,
79.         pBackBuffer,
80.         0, 0,
81.         0, 0,
82.         640, 480);
83.
84.     break;
85. }
86. default:
87.     std::cout << "event: " << e.type << std::endl;
88.     break;
89. }
90. }
91.
92. cairo_destroy(cr);
93. cairo_surface_destroy(surface);
94.
95. XDestroyWindow(dpy, w);
96. XCloseDisplay(dpy);
97.
98.
99. return 0;
100. }
```

至此，我们完成了在 X11 下使用 Cairo 引擎绘制图形的全部过程。Windows 的程序架构和事件循环有所区别，但思路是相同的。当然，这仅仅是阐述了基本过程，还有更多的细节值得研究和探讨。

---

## 2.4 使用 Woboq CodeBrowser 搭建源代码网站

### 2.4.1 背景

在日常学习工作中，我们不免需要浏览一些库的源码。在本地浏览源代码，例如使用 **Source Insight**，当然是可以的，但问题是一是不方便，二是很多库下载下来是以头文件配合静态库或动态库的形式存在的，看不到 **c++** 代码，因此阅读会受限。现在 **Web** 技术高速发展，有没有办法用网页直接查看源代码，并且还有类似于 **Code** 的代码跳转功能呢？换句话说，如何把 **C/C++** 代码转化为前端页面，并且最好是静态的前端页面，就是一个难题了。

幸运的是，**github** 上有人提前考虑到了这件事情，并且有了具体的项目，已经有 1k 多的 **star**。并且原作者还基于这个框架弄出了一个在线的源代码网站，方便开发者查看各个 **C/C++** 库的源代码，例如 **Qt**、**GCC**、**Linux Kernel**、**GNU C Library** 等。

那么问题来了，都有在线的网站了为什么还要自己搭建一个呢？原因就是这个网站在国内被墙了。有人会说？开梯子啊，不是每个节点都能上，我嫌他烦，因此决定自己搭建一个源代码网站一劳永逸。

### 2.4.2 工作原理简介

**Woboq CodeBrowser** 是基于 **LLVM/Clang** 实现的一个命令行工具。它通过深度解析 **C/C++** 源码生成最终我们需要的静态 **HTML** 文件。**Woboq CodeBrowser** 包含 **codebrowser\_generator** 和 **codebrowser\_indexgenerator** 两个子命令。生成 **HTML** 文件到最终可以通过浏览器阅读代码，整体分三个步骤：

1. 先通过 **codebrowser\_generator** 解析 **.h** 和 **.cpp** 生成对应的 **.h.html** 和 **.cpp.html**。
2. 然后通过 **codebrowser\_indexgenerator** 为所有目录生成 **index.html**。
3. 最后我们可以把这些 **HTML** 文件拷贝到某个 **Web** 服务器上，就可以在浏览器里愉快地浏览 **C/C++** 项目的源码了。

### 2.4.3 安装 Woboq CodeBrowser 工具

我们通过源码编译安装这个工具。

首先将该项目克隆到本地，由于需要用 **LLVM/Clang** 工具链，强烈建议在 **Linux** 下操作。

```
1. git clone https://github.com/KDAB/codebrowser.git
```

克隆下的项目中有三个目录比较重要，一是 **data** 目录，这是一些前端的资源

文件，例如 png、css、js 等；二是 generator 目录，这是可执行文件 codebrowser\_generator 的源码目录；三是 indexgenerator，这是可执行文件 codebrowser\_indexgenerator 的源码目录。

前面提到，编译这个工具需要用到 LLVM/Clang 工具链，结合 github 上的 issue#119，使用 llvm-14 和 clang-14 的版本比较合适。以 Ubuntu 为例：

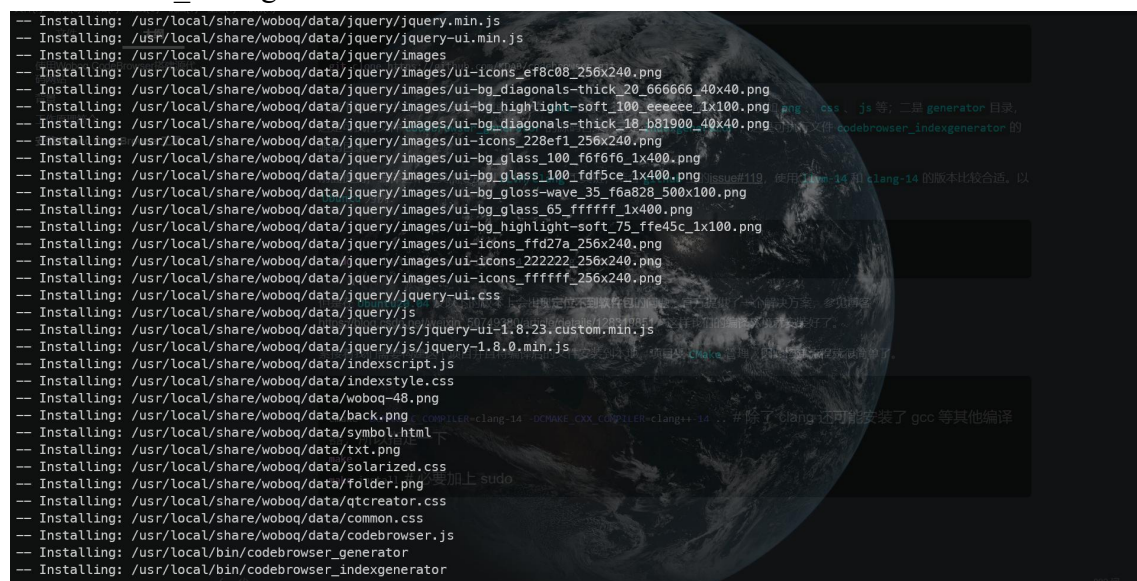
```
1. sudo apt install llvm-14 clang-14 libclang-14-dev
```

但是在 Ubuntu20.04 及以下的版本上会出现定位不到软件包的问题，官方提供了一个解决方案，参见博客 [https://blog.csdn.net/weixin\\_50749380/article/details/128319851](https://blog.csdn.net/weixin_50749380/article/details/128319851)

这样我们的编译环境就安装好了，紧接着我们需要构建这个项目并且将编译后的文件安装到本地。项目受 CMake 管理，同时参考官方文档，构建流程就很简单了。

```
1. mkdir -p build
2. cd build
3.
4. cmake -DCMAKE_C_COMPILER=clang-14 -DCMAKE_CXX_COMPILER=clang++-14 -DCMAKE_BUILD_TYPE=Release .. # 除了 clang 还可能安装了 gcc 等其他
   的编译器，所以指定一下；同时指定编译选项为 Release，这个在 Linux 下没有影响
5. make
6. sudo make install # 必要加上 sudo
```

安装完毕以后 data 目录，两个可执行文件 codebrowser\_generator 和 codebrowser\_indexgenerator 就成功安装到本地了。



```
-- Installing: /usr/local/share/woboq/data/jquery/jquery.min.js
-- Installing: /usr/local/share/woboq/data/jquery/jquery-ui.min.js
-- Installing: /usr/local/share/woboq/data/jquery/images
-- Installing: /usr/local/share/woboq/data/jquery/images/ui-icons_ef8c08_256x240.png
-- Installing: /usr/local/share/woboq/data/jquery/images/ui-bg_diagonals-thick_20_666666_40x40.png
-- Installing: /usr/local/share/woboq/data/jquery/images/ui-bg_highlight-soft_100_eeeeee_1x100.png
-- Installing: /usr/local/share/woboq/data/jquery/images/ui-bg_diagonals-thick_16_b81900_40x40.png
-- Installing: /usr/local/share/woboq/data/jquery/images/ui-icons_228ef1_256x240.png
-- Installing: /usr/local/share/woboq/data/jquery/images/ui-bg_glass_100_f6f6f6_1x400.png
-- Installing: /usr/local/share/woboq/data/jquery/images/ui-bg_glass_100_fdf5ce_1x400.png
-- Installing: /usr/local/share/woboq/data/jquery/images/ui-bg_gloss-wave_35_f6a828_500x100.png
-- Installing: /usr/local/share/woboq/data/jquery/images/ui-bg_glass_65_ffffff_1x400.png
-- Installing: /usr/local/share/woboq/data/jquery/images/ui-bg_highlight-soft_75_ffe45c_1x100.png
-- Installing: /usr/local/share/woboq/data/jquery/images/ui-icons_ffd27a_256x240.png
-- Installing: /usr/local/share/woboq/data/jquery/images/ui-icons_222222_256x240.png
-- Installing: /usr/local/share/woboq/data/jquery/images/ui-icons_ffffff_256x240.png
-- Installing: /usr/local/share/woboq/data/jquery/jquery-ui.css
-- Installing: /usr/local/share/woboq/data/jquery/js
-- Installing: /usr/local/share/woboq/data/jquery/js/jquery-ui-1.8.23.custom.min.js
-- Installing: /usr/local/share/woboq/data/jquery/js/jquery-1.8.0.min.js
-- Installing: /usr/local/share/woboq/data/indexstyle.js
-- Installing: /usr/local/share/woboq/data/indexstyle.css
-- Installing: /usr/local/share/woboq/data/woboq-48.png
-- Installing: /usr/local/share/woboq/data/back.png
-- Installing: /usr/local/share/woboq/data/symbol.html
-- Installing: /usr/local/share/woboq/data/txt.png
-- Installing: /usr/local/share/woboq/data/solarized.css
-- Installing: /usr/local/share/woboq/data/folder.png
-- Installing: /usr/local/share/woboq/data/qtcreator.css
-- Installing: /usr/local/share/woboq/data/common.css
-- Installing: /usr/local/share/woboq/data/codebrowser.js
-- Installing: /usr/local/bin/codebrowser_generator
-- Installing: /usr/local/bin/codebrowser_indexgenerator
```

图 2-19 woboq 工具安装结果



---

## 2.4.4 如何使用

接下来以 googletest-1.12.1 为例，展示如何使用这个工具构建静态的源代码网站。

### 1、生成 compile\_commands.json

compile\_commands.json 文件是一种特定格式的 compilation database 文件，而所谓 compilation database 其实很简单，它里面记录的是每一个源码文件在编译时详细的信息（包括文件路径，文件名，编译选项等等）。而 compile\_commands.json 文件是 LibTooling 需要的以 json 格式呈现的 compilation database 文件，以下截取的是 compile\_commands.json 中的一个 entry：

```
1.  [
2.  ...
3.  {
4.      "arguments": [
5.          "c++",
6.          "-c",
7.          "-g",
8.          "-O2",
9.          "-Werror",
10.         "-std=c++0x",
11.         "-Wall",
12.         "-fPIC",
13.         "-o",
14.         "attrs.o",
15.         "attrs.cc"
16.     ],
17.     "directory": "/home/astrol/libelfin/dwarf",
18.     "file": "attrs.cc"
19. },
20. ...
21. ]
```

Woboq Codebrowser 是基于 LLVM/Clang 实现的工具，也是基于 compile\_commands.json 来分析源码关系的。

换句话说，要想使用 Woboq Codebrowser，必须首先生成 compile\_commands.json 文件。如果项目是由 cmake 构建的，那么恭喜你，只需加上 -DCMAKE\_EXPORT\_COMPILE\_COMMANDS=ON 即可。如果是传统的 make build system 也不要担心，Bear 和 compdb 工具可以帮我们生成

---

compile\_commands.json 文件。

现在让我们一起生成 googletest-1.12.1 项目对应的 compile\_commands.json。首先我们拉取 googletest 的源码到本地。源码采用 CMake 工具的管理，这样正好很方便的能帮我们生成 compile\_commands.json。之所以这么推荐 CMake 是因为使用其他的工具可能需要编译整个工程才能生成该文件，费事费力。

因为我是 C++11 标准的环境，因此需要使用的版本是 1.12.1，切换到对应的 tag。当然直接下载对应的 release 源码也行。

```
1. git checkout release-1.12.1 # target tag
```

然后我们使用 CMake 工具生成 compile\_commands.json 文件，个人建议使用 gcc/g++ 编译器。

```
1. mkdir -p build
2. cd build
3.
4. cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_C_COMPILER="gcc" -DCMAKE_CXX_COMPILER="g++" -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ..
```

在 build 目录下看到 compile\_commands.json 文件正确生成即为成功。

## 2、使用 codebrowser\_generator 生成 html

codebrowser\_generator 是一个可执行文件，有各种执行参数，说明如下：

- -a: 处理 compile\_commands.json 中的所有文件。如果没有传递这个参数，那么就需要传递要处理的文件列表
- -o: 指定文件输出目录
- -b: 是指包含 compile\_commands.json 的构建目录。如果没有传递这个参数，编译参数可以在 -- 后通过命令行传递
- -p: （一个或多个）用于项目规范。即项目的名称、源代码的绝对路径和用冒号分隔的版本信息。示例：-p projectname:/path/to/source/code:0.3beta
- -d: 指定包含所有 JavaScript 和 CSS 文件的数据 URL。默认为相对于输出目录的 ../data。示例：-d https://codebrowser.dev/data/
- -e: 是对外部项目的引用。示例：-e clang/include/clang:/opt/llvm/include/clang/:https://codebrowser.dev/llvm

例如，对于当前 googletest-1.12.1 项目，一条合适的命令可能是这样的：

```
1. codebrowser_generator -b ./compile_commands.json -a -p googletest-1.12.1:"${PWD}/.." :1.12.1 -o ./docs -d ../data
```

执行完的结果是这样的：

```
lzx0626@DavidingPlus $ codebrowser_generator -b ./compile_commands.json -a -p googletest-1.12.1:"${PWD}/..":1.12.1
1 -o ./docs -d ../data
[25%] Processing /home/lzx0626/DavidingPlus/googletest/googletest/src/gmock-all.cc
[50%] Processing /home/lzx0626/DavidingPlus/googletest/googletest/src/gmock_main.cc
[75%] Processing /home/lzx0626/DavidingPlus/googletest/googletest/src/gtest-all.cc
[100%] Processing /home/lzx0626/DavidingPlus/googletest/googletest/src/gtest_main.cc
```

图 2-20 codebrowser\_generator 命令执行结果

需要注意的是，上面的操作都是在根目录的 build 构建目录执行的，这也是 CMake 用户的构建习惯。

同时，-o ./docs 代表文件输出在 build/docs 中，-d ../data 是指定资源文件的相对路径，代表资源文件位于生成目录 docs 的父级目录。至于为什么是../data，后面会详细解释。安装好 Woboq CodeBrowser 工具以后 data 目录会被安装在 /usr/local/share/woboq/data，将其拷贝为 build/data 即可。

### 3、使用 codebrowser\_indexgenerator 为每个目录生成 index.html

codebrowser\_indexgenerator 同样有很多执行参数，说明如下：

- -p: （一个或多个）用于项目规范。即项目的名称、源代码的绝对路径和用冒号分隔的版本信息。示例：-p projectname:/path/to/source/code:0.3beta
- -d: 指定包含所有 JavaScript 和 CSS 文件的数据 URL。默认为相对于输出目录的 ../data。示例：-d https://codebrowser.dev/data/

同样执行类似命令：

```
1. codebrowser_indexgenerator ./docs -d ../data
```

类似执行结果如下，可以看到每个目录对应都有了 index.html。

```
lzx0626@DavidingPlus $ codebrowser_indexgenerator ./docs -d ../data
Generating ./docs/index.html
Generating ./docs/googletest-1.12.1/index.html
Generating ./docs/googletest-1.12.1/googletest/index.html
Generating ./docs/googletest-1.12.1/googletest/include/index.html
Generating ./docs/googletest-1.12.1/googletest/include/gmock/index.html
Generating ./docs/googletest-1.12.1/googletest/include/gmock/internal/index.html
Generating ./docs/googletest-1.12.1/googletest/include/gmock/internal/custom/index.html
Generating ./docs/googletest-1.12.1/googletest/src/index.html
Generating ./docs/googletest-1.12.1/googletest/index.html
Generating ./docs/googletest-1.12.1/googletest/include/index.html
Generating ./docs/googletest-1.12.1/googletest/include/gtest/index.html
Generating ./docs/googletest-1.12.1/googletest/include/gtest/internal/index.html
Generating ./docs/googletest-1.12.1/googletest/include/gtest/internal/custom/index.html
Generating ./docs/googletest-1.12.1/googletest/src/index.html
```

图 2-21 codebrowser\_indexgenerator 命令执行结果

### 4、关于 -d 参数以及资源文件

上面提到，为什么我使用的是 -d ../data 呢？我们先弄清楚 data 目录是干嘛的。

data 目录存放着前端代码需要的资源文件，例如 CSS、JS 和图片资源等。因此需要指明这个目录的路径，绝对路径或者相对路径。由于我的 code-browser 项目需要存放很多库的源代码，因此我将 data 目录统一在根目录中，每个库的源代

码对应一个自己的子目录，因此最终的结果是这样的：

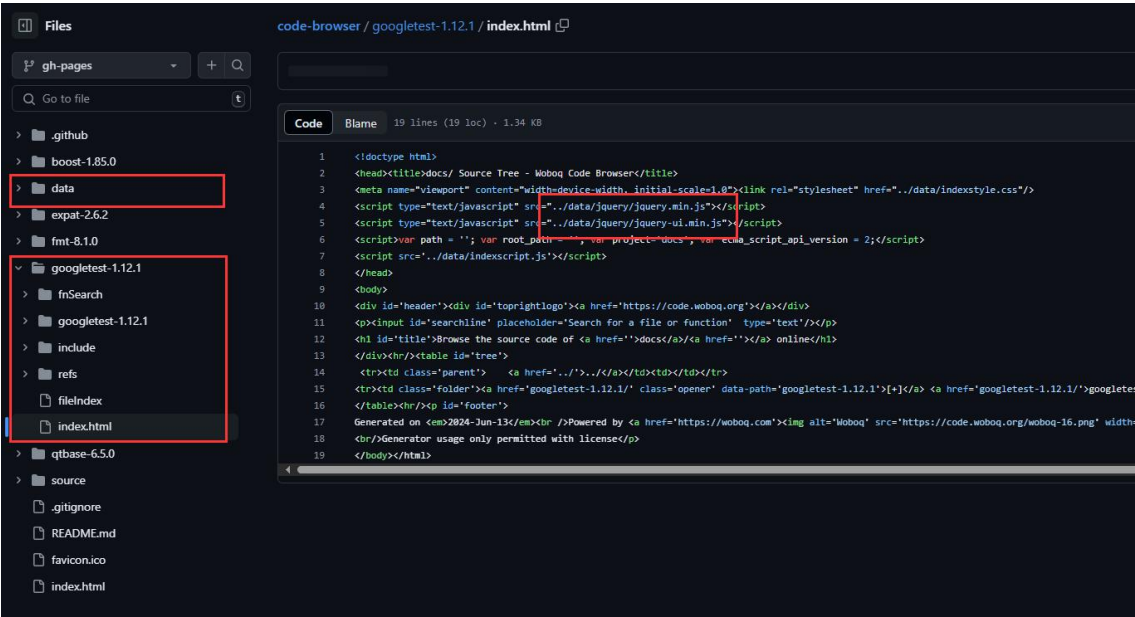


图 2-22 code-browser 项目结构图

这样就能正确定位到资源文件的路径了，codebrowser\_generator 生成的 html 文件同理。用户使用自己的习惯和方式即可。

---

## 3 知识技能学习情况

本部分将从知识技能学习的主题出发，从开发环境和工具、预备知识、新知识点的学习出发，阐述实际工作过程中所涉及到的技术栈、开发环境、工程思想等，做一个总结性收获的记录。

### 3.1 开发环境和工具

#### 3.1.1 开发环境

LarkSDK 是跨平台的，在很多操作系统上都能构建，主要是 windows 和 Linux。在实习的初期阶段，我主要在 Linux 操作系统上进行研发，使用的 Linux 系统发行版是 Ubuntu 20.04，由于项目的语言是 c++，语言标准规定为 c++11，同时需要使用编译器 gcc，版本是 gcc 9.4.0，同时构建需要的 CMake 版本不小于 3.12。但是在中期以后的工作中，由于部分功能涉及到跨平台的校验，例如文件系统 filesystem 的相关处理，需要依赖 windows 的环境，从而兼容跨平台了。目前编译 LarkSDK 框架的 windows 系统推荐使用 win10，使用的编译器是微软官方提供的 MSVC14+，对应 VS 的版本是 VS2019。

由于 LarkSDK 是一个跨平台的底层 c++ 框架，因此兼容多个操作系统，例如 linux 和 windows，是必然需要考虑的问题，在 LarkSDK 中体现的最突出的就是不同平台下的窗口逻辑处理的相关机制，包括但不限于事件循环，绘图机制等。目前 LarkSDK 中考虑了 win32、x11 和 wayland 的平台差异并分别处理，同时采用抽象父类指针多态的模式进行统一管理。同时，在其他模块的工作中或多或少都涉及到了跨平台的相关需求。

#### 3.1.2 开发工具

关于代码 IDE，windows 下推荐使用 VS，Linux 下使用 VS Code 即可。VS 完整的工具链为 windows 环境下的构建、测试等提供了非常便捷的操作，缺点则是工具太大，且功能过于复杂，甚至有些繁琐；而 VS Code，由于具有众多插件，并且 IDE 非常轻量，因此非常适合在 Linux 下进行使用。

关于代码托管工具，毫无疑问是 git。至于平台，我们是在公司服务器上部署 gitlab。gitlab 开源，并且可以通过脚本或者 docker 轻松的部署在内网当中，对于任何企业，特别是像我们这样的军工企业，就非常方便了。同时同前面提到，gitlab 的 CI 等功能，可以方便实现自动化打包、扫描等需求，开发流程一目了然。

---

## 3.2 预备知识

由于项目中使用的是 `c++` 语言，因此一定的 `c++` 语言基础是必不可少的。这其中包括但不限于基础语法、关键字、命名空间、输入和输出、函数重载、引用等等。另外 `c++` 作为一门面向对象的语言，理解类和对象也是必不可少的，其中包括但不限于面向对象和面向过程的区别、类的定义、类的封装、类的作用域、初始化列表、内部类等等知识的熟悉和掌握。另外由于 LarkSDK 是一个偏底层的基础平台库，其中很多的基础类例如 `LVector`、`LString` 等都和 `std` 的标准容器有着很大的联系，因此对 `c++` 的 STL 有一定的了解也是很重要的，每种容器的内存模型和管理、设计思路等，都会在实际工作中无时无刻不被运用。

其次，由于 LarkSDK 主要在 Linux 上进行研发，因此 Linux 基本的命令是非常必要的，包括但不限于 `ls`、`rm`、`cd`、`mkdir` 等。同时需要理解 Linux 系统和 windows 系统的区别，包括但不限于文件系统、命令系统等。

同时由于项目托管使用了 `git`，因此需要会使用 `git` 的基本命令，例如 `git pull`、`git merge`、`git push` 等等。这些都会在实际工作当中时时刻刻被运用。

## 3.3 新知识点学习和掌握情况

### 3.3.1 CMake

在使用 IDE 开发软件的过程中，代码的编译和构建一般是使用 IDE 自带的编译工具和环境进行编译，例如 VS，JetBrains 全家桶等，开发者参与的并不算多。但是如果想要控制构建的细节，则需要开发者自己定义构建的过程。而 CMake 工具是一个开源、跨平台的编译、测试和打包工具，它使用比较简单的语言描述编译、安装的过程，输出 Makefile 或者 project 文件，再去执行构建。

CMake 工具具有很多优点。第一，CMake 是一个跨平台的构建工具，可以在各种操作系统上使用，包括 Windows、Linux、macOS 等。第二，CMake 允许开发者使用简单的语法编写项目的构建规则，然后根据这些规则自动生成特定平台下的构建系统文件，如 Makefile、Visual Studio 解决方案、Xcode 项目等。第三，CMake 使用 CMakeLists.txt 文件来描述项目的构建规则和依赖关系。这些文件可以分成多个模块，使得项目的组织结构更加清晰，并且易于维护和管理。第四，CMake 支持各种主流的编译器和构建工具，包括 GCC、Clang、Visual Studio、Xcode 等，以及各种构建系统，如 Make、Ninja 等。第五，CMake 支持生成多种不同配置的构建系统，例如 Debug、Release、MinSizeRel 等，这使得开发者可

---

以针对不同的需求生成不同的构建版本。第六，CMake 允许开发者方便地管理项目所依赖的外部库和模块，可以通过 `find_package()` 函数来查找已安装的库，并自动配置相关的构建规则。最后，CMake 拥有庞大的用户社区和活跃的开发团队，提供了丰富的文档、示例和支持资源，使得开发者能够快速上手并解决遇到的问题。

关于多个 c/cpp 文件的编译，用 MakeFile 自然是没有问题的，但是一旦文件架构复杂起来，MakeFile 文件的编写就比较困难。CMake 就是在 MakeFile 的基础上，提供了全自动化的构建方案，只需要做好配置，就能自定义构建的过程，这也是 CMake 最方便的地方。

### 3.3.2 Conan

在任何项目的当中，除非是从头造轮子，引入一些第三方库是必不可少的。在前后端当中有比较完善的包管理器例如前端的 `node`，后端的 `maven`；在 `python` 当中有 `pip/pip3`；在 `Ubuntu` 当中有 `apt`，在 `centos` 中有 `yum` 等等。但是对于 `c++`，我以前从来没有听过任何的第三方库或者包管理器，这也和 `c++` 是偏底层的语言，习惯造轮子比较相关。但是实际项目里面确实需要依赖一些第三方开源库，例如 LarkSDK 的 `libpng` 库，这个时候 Conan 包管理器就发挥它的作用了。

Conan 是一个开源的 C/C++ 软件包管理器，用于帮助开发人员管理项目中的依赖项。它的主要目标是简化 C/C++ 项目中的依赖项管理过程，提供了一种方便的方式来引入、构建和共享依赖项，同时解决了版本冲突和平台兼容性问题。Conan 包的安装也非常简单，只需要通过 `pip` 即可。

Conan 有很多优点。第一，Conan 可以在各种操作系统上运行，包括 Windows、Linux 和 macOS 等。第二，Conan 允许用户从中央仓库或自定义仓库中获取预编译的软件包，并将它们安装到本地环境中。第三，Conan 能够解析项目依赖项的依赖关系，并自动下载和安装所需的依赖项，包括库文件、头文件和其他构建工具等。第四，Conan 允许用户指定所需依赖项的版本，并在需要进行版本切换或升级。第五，Conan 允许用户根据项目需求自定义构建选项，并将这些选项传递给依赖项的构建过程。第六，Conan 可以与各种常用的构建系统集成，包括 CMake、Makefile、Visual Studio 等，以便将依赖项集成到项目的构建过程中。最后，Conan 支持与其他包管理系统（如 CMake、vcpkg 等）集成，以便在不同的项目和环境中共享和重用依赖项。

在 LarkSDK 中需要引入很多第三方开源库，在 LarkTestKit 中则需要引入



---

LarkSDK 库。将 LarkSDK 和 LarkTestKit 的源文件编译成为对应的库文件，再和头文件打包在一起，通过 Conan 包的相关命令，就能很方便的将结果打成 Conan 包，这不仅对于后续测试人员做测试、版本发布等，都起着非常重要的作用。至于 Conan 具体的使用细节，前面第二部分已经阐述过了，这里不再赘述。

### 3.3.3 对实用工具的深入理解

在以前，关于 STL，也就是 c++ 的标准模板库，我只是会用，稍微了解内部原理的状态。但是由于代码走查的工作，我接触到了不同的底层设计，也借机会对比了我们的实用工具和 STL 的设计思路。

以 `std::string` 和我们的 `LString` 为例对比。`std::string` 对应 c++ 的标准字符串，使用的单字节编码，并且只支持 ASCII 或者 ISO-8859-1 编码；`LString` 使用的是双字节编码，因此可以支持 unicode 中的字符，应用空间更广。其次，`std::string` 的内存结构是栈区存放堆区的指针，而堆区存放 C 风格的字符串（当然 `std::string` 还有其他的优化，这里仅声明绝大多数情况）；而 `LString` 继承 `LVector<unsigned short>`，字符串中的每个元素都是一个 unicode 字符，这样大大复用 `LVector` 的功能，对于底层的内存管理也交给 `LVector` 处理。同时 `LString` 提供了更多的功能，例如 `split`、`trim`、`format` 等等。

以上的例子还有很多，通过代码走查的工作，我对底层实用工具的理解更深了，虽然可能达不到自己动手实现的水平，但是能够大致说出不同的设计思路，实现方案，也算是达到了目的。

### 3.3.4 C++ Boost 后备标准库的知识

Boost 是为 C++ 语言标准库提供扩展的一些 C++ 程序库的总称。Boost 库是一个可移植、提供源代码的 C++ 库。作为标准库的后备，是 C++ 标准化进程的开发引擎之一，是为 C++ 语言标准库提供扩展的一些 C++ 程序库的总称。

Boost 社区建立的初衷之一就是为 C++ 的标准化工作提供可供参考的实现，Boost 社区的发起人 Dawes 本人就是 C++ 标准委员会的成员之一。在 Boost 库的开发中，Boost 社区也在这个方向上取得了丰硕的成果。在送审的 C++ 标准库 TR1 中，有十个 Boost 库成为标准库的候选方案。在更新的 TR2 中，有更多的 Boost 库被加入到其中。从某种意义上讲，Boost 库成为具有实践意义的准标准库。

具体到本项目的任务上，本项目原则上不应直接引用 Boost 库，然而，由于 Boost 库中包含了很多已经达到标准库水平但标准库尚未实现的功能模块，因此可

---

以将其作为参考，挖掘原理以后采取自行编写的方式。例如 `util` 模块中的 `LUuid`、`LMd5`，`xml` 模块的 `LXmlStream` 类等都在 `Boost` 库中都有相应的体现。

### 3.3.5 Xml 基础知识

可扩展标记语言(Extensible Markup Language, XML)，可以用来标记数据、定义数据类型，是一种允许用户对自己的标记语言进行定义的源语言。XML 是一种独立于软件和硬件的工具，用于存储和传输数据。在电子计算机中，标记指计算机所能理解的信息符号，通过此种标记，计算机之间可以处理包含各种的信息比如文章等。它可以用来标记数据、定义数据类型，是一种允许用户对自己的标记语言进行定义的源语言。它非常适合万维网传输，提供统一的方法来描述和交换独立于应用程序或供应商的结构化数据。是 Internet 环境中跨平台的、依赖于内容的技术，也是当今处理分布式结构信息的有效工具。

具体到本项目的任务上，我在进行 Xml 模块的代码走查的时候，首先需要了解 Xml 的一些基础知识，比如语法定义、语句含义等，这样才能为我们自己的解析、读取和写入 Xml 流打下基础。虽然没有深入挖掘 Xml 的更多特性或者高级用法，但是 Xml 确实是一种高度自定义化的存储和传输数据的语言，非常方便好用。

### 3.3.6 对 STL 的进一步理解

在重构关联性容器 `LMap`、`LHash`、`LSet` 的时候，初版的代码是采用从头手写造轮子的方式，虽然功能可用，但是难以达到商用的需求。因此经过代码走查以后，决定采用封装 STL 标准库容器的方式实现。我负责这部分的工作，由此进一步加深了对底层红黑树、哈希表的理解。

以标准库的 `map` 和 `unordered_map` 为例，`map` 的底层是红黑树，`unordered_map` 的底层是哈希表。二者提供的功能类似，都是通过键来快速查找值，但是实现方式却完全不同，导致使用的时候需要注意的细节也不一样。

红黑树 (Red-Black Tree) 是一种自平衡二叉搜索树，通过一组严格的规则来保持树的平衡，确保插入、删除和查找操作的时间复杂度为  $O(\log n)$ 。其主要特性包括：每个节点是红色或黑色，根节点是黑色，所有叶子节点 (NIL 节点) 是黑色，红色节点的子节点必须是黑色，从任一节点到其每个叶子的所有路径包含相同数量的黑色节点。为了维护这些特性，红黑树通过左旋和右旋操作调整节点的位置。红黑树的这种高度平衡特性，使其在插入和删除操作后，树的高度始终保持在较低水平，从而保证操作的效率。红黑树广泛应用于计算机系统和标准库容器中，

---

如 C++ 的 `std::map` 和 `std::set`，在实际应用中，它提供了高效的动态数据集管理，使得各种操作在大多数情况下都能在对数时间内完成。这种数据结构的稳定性和高效性，使其成为许多算法和系统实现中不可或缺的一部分。

哈希表（Hash Table）是一种高效的数据结构，用于实现平均情况下常数时间复杂度的插入、删除和查找操作。其基本原理是通过哈希函数将数据的关键码映射到表中的一个位置（即哈希值），然后将数据存储在这个位置。哈希表主要由两个部分组成：哈希函数和存储数组。哈希函数的设计至关重要，决定了数据在表中的分布情况，理想的哈希函数能将数据均匀地分布在整个数组中，从而减少冲突（多个关键码映射到同一位置的情况）。为了处理冲突，常用的方法包括链地址法和开放地址法。链地址法在每个数组元素中维护一个链表来存储所有哈希值相同的元素，而开放地址法则在冲突位置寻找下一个可用位置。在 C++ 标准库中，`std::unordered_map` 就是一个基于哈希表实现的容器，通过哈希表提供快速的键值对存储和查找功能。虽然哈希表在最坏情况下的时间复杂度为  $O(n)$ ，但通过设计良好的哈希函数和适当的负载因子，哈希表在大多数实际应用中的性能非常优越。

即便是封装标准库已有的容器，但在实际处理的时候，我仍然遇到了一些问题。其中一个不小的问题就是关于迭代器。为了兼容 QT 用户的使用习惯，迭代器的 `*` 运算符重载需要返回元素值 `data`，因此不能直接简单的直接使用标准库对应的迭代器，而需要再做一层封装。这个时候问题来了，如果在使用迭代器的时候越界了怎么办？LHash 是单向迭代器，LMap 是双向迭代器，而标准库对于越界的处理是很模糊的，尤其这两个东西都涉及到指针，因此稍不注意就会发生内存泄漏的问题，紧接着就是一系列的段错误 `core dump`，结果可以称之为灾难性的打击。因此我不得不深挖标准库的源码，找出迭代器 `++` 或者 `--` 到底发生了什么操作，这样才能知道原因，并且给出合理的解决方案。经过调研，目前在迭代器中引入容器的 `size` 以及当前访问的下标 `index` 辅助判断迭代器的操作是否越界，方便抛出异常，这才是一个完整的问题的解决过程。

---

## 4 结束语

### 4.1 实习工作完成情况总结

结合前面的内容，我的实习工作总结为以下几点：

(1) 负责 LarkSDK 的部分代码走查工作。审核走查现有的老版本的代码，总结归纳相关问题，形成文档，并于会议讨论，形成解决方案。

(2) 负责 LarkSDK 的部分代码优化重构工作。针对代码走查中出现的问题，根据解决方案实行修改。

(3) 负责 LarkSDK 的部分新功能的开发。根据实际应用过程的产生的新需求，进行调研工作，总结文档，形成设计方案，并予以执行。

(4) 负责 LarkTestKit 的协助工作。审核该部分源代码，与校方同学和测试沟通，并设计编写测试用例。同时协助校方同学完成该部分的工作，讨论部分设计思路、实现方案等。

各个部分在实习期间具体完成的内容已经以图表的形式罗列在前面了。总体而言，我对自己的实习目标完成的还是相当不错的。对于公司目标，我严格按照公司的安排，全身心投入合迅智灵基础平台的研发，完成了很多有意义的工作。对于个人目标，我在实习期间跟随前辈的脚步，保质保量完成好安排到的工作，同时在实习期间学到了很多知识和技术，对以后的深造和就业有着非常重要的帮助。这是一个双向互利、共赢的过程。

### 4.2 对于企业实习的收获及体会

---

## 参考文献

- [1] 王浩刚, 聂在平. 三维矢量散射积分方程中奇异性分析[J]. 电子学报, 1999, 27(12): 68-71
- [2] X. F. Liu, B. Z. Wang, W. Shao. A marching-on-in-order scheme for exact attenuation constant extraction of lossy transmission lines[C]. China-Japan Joint Microwave Conference Proceedings, Chengdu, 2006, 527-529
- [3] 竺可桢. 物理学[M]. 北京: 科学出版社, 1973, 56-60

---

## 致谢

总结报告最基本要求是 15000 字以上。