信息与软件工程学院

企业实习初期报告

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

目 录

1 企业实习目标任务

本部分将从企业公司实习情况和岗位职责、实习目标和具体任务出发，阐述公司的背景、具体业务以及发展趋势等。同时结合自身岗位和业务的要求，明确自己的实习目标，并按照时间线罗列具体任务，做阶段性总结。

1.1 实习公司情况和岗位职责

1.1.1 公司情况

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，见图 1-1：

图 1-1

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 1-2：

图 1-2

我目前参与在合迅智灵的产品研发中，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。

以下是合迅智灵产品的功能架构，见图 1-3。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具体阐述。

图 1-3

1.2 实习目标和具体任务

1.2.1 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，全身心投入完成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。

1.2.2 具体任务

1.2.2.1 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（后面用 SDK 简写）。SDK 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 QtSDK。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 SDK 的部分作为基础平台，又是重中之重。

SDK 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：

1. LarkSDK-DB (larkdb)：数据库支持模块。

2. LarkSDK-Network (larknetwork)：网络编程支持模块。

3. 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 SDK 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。

以下列出到目前为止的代码走查表 1-1：（注：公司两周为一个迭代，因此任务都是按迭代划分的）

表 1-1

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

既然做了代码走查，那么必然需要记录，后续修改。目前公司已经完成对三大核心模块的代码走查，并慢慢开始优化重构。在上述时间期间，我的第二份任务就是优化重构部分代码，包括重构 LStack、LQueue 等；优化修改线程部分代码、LPen、LBrush 等。但是由于公司其他项目比较紧急，该部分的优化重构并未作为主要迭代任务。

在 SDK 中，除了源代码，还有针对源代码的测试代码。以下是项目的文件架构图，其中 src 是源代码部分，snippet 是研发人员存放自己测试代码的地方，test 是单元测试部分，具体见图 1-4：

图 1-4

单元测试 test，其中的测试代码采用 googleTest 架构编写，这部分的测试代码会在 gitlab 后台每次 commit 的时候自动通过 CI 调用脚本运行，称之为 scan。scan 之后会通过某种技术分析出本次运行之后的结果，包括编译、运行是否报错，跑通的代码占整个代码的比例等，其中跑通的代码占整个代码的比例称之为代码覆盖率。代码覆盖率有效的反映了源代码当中实际起作用的代码比例，有助于对冗杂代码的优化修改，这也是验收的一个重要标准。因此我的第三个任务就是负责 util 部分和 network 部分的单元测试，编写测试样例，维持流程，并且提高覆盖率。以下是代码覆盖率的变化趋势图 1-5：

图 1-5

可以看出在一二月的时间覆盖行的个数到达了顶峰，在这期间我针对 util 和 network 部分进行了单元测试代码的编写，分为将覆盖率提升到了 75%和 90%，基本达到了 80%的标准。但是由于后面领导对代码结构的调整和修复紧急 bug 等，加上我本身有其他工作，因此后续的单元测试覆盖率有所下降，但是在整个过程中，我一直维持单元测试的流程，让其不会在流程上编译或者运行失败。

1.2.2.2 LarkTestKit

LarkSDK 作为一个跨平台的 C++基础开发库，前面的单元测试提到，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，并在 3 月 19 日提交了初版，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，具体如图 1-6：

图 1-6

当我接手的时候，LarkTestKit 代码库还是初始化的状态。这意味着我需要编写配置文件将本地编译运行流程跑通。同时由于需要同 LarkSDK 一样，需要走自动化打包和静态扫描的流程，因此我还需要编写自动化脚本实现这部分工作。具体的细节第二部分会提到。

1.2.2.3 总结

总结一下，我将截止到目前的任务按照迭代周期时间线汇总到表格当中，见表 1-2：

表 1-2

迭代周期	具体完成工作
1.15 - 1.26	1. 走查代码 LStack、LQueue、LByteArray 2. 编写单元测试内容
1.29 - 2.8	1. 重构代码 LStack、LQueue 2. 走查代码 LObject，LApplication，LSignal 3. 编写单元测试内容
2.19 - 3.1	1. 走查线程与同步相关部分代码 2. 优化修改线程部分代码 3. 编写单元测试内容
3.4 - 3.15	1. 走查代码 LPen、LBrush、LLinearGradient、LMenu、LMenuItem、LMenuSeperator 2. 优化修改画笔、画刷、菜单部分代码
3.18 - 3.29	1. 审核 LarkTestKit 代码 2. 构建代码库流程，包括本地跑通、CI 自动化打包等 3. 与测试进行沟通，编写部分测试样例

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。

2.1 LByteArray：内存管理问题

这部分是在走查代码 LByteArray 的时候遇到的工程上的严重问题。

先简单介绍下 LByteArray，顾名思意，这是一个字节数组的工具类，可以将对二进制数据进行编码或者解码之后以字节为单位进行输出，提升了操作的便携性和安全性，尤其与字符串 LString 的联系非常紧密，可以与字符串之间进行相互转换。可以见得字节数组是一个非常底层的基础和重要的工具类，很多上层的类都需要依托于该类。因此这个类的设计和管理就非常重要。

因此，在走查的时候，旧版本的内存管理一塌糊涂，以 insert()函数为例，代码如下：

```
1.  LByteArray& LByteArray::insert(const char* str, int len, int pos)
2.  {
3.    // pos 越界则直接返回
4.    if (pos < 0 || pos > m_size)
5.    {
6.      throw LException("Invalid param:Pos index out of bounds!");
7.      return *this;
8.    }
9.    if (len < 0 || len > strlen(str) + 1)
10.   {
11.     throw LException("Invalid param:Length must be smaller than the st
    ring length and not be nagetive!");
12.     return *this;
13.   }
14.   // 更新 size 的信息，开辟新内存
15.   m_size = m_size + len;
16.   unsigned char* lc = new unsigned char[m_size + 1];
17.   // 将插入点以前的内容复制到新内存中
18.   memcpy(lc, m_pByte, (pos) * sizeof(unsigned char));
19.   // 复制要插入的内容到新内存中
20.   for (int i = pos, j = 0; i < (pos + len); i++, j++)
21.   {
22.     unsigned char tempByte = 0;
23.     tempByte = (unsigned char)*(str + j);
24.     *(lc + i) = tempByte;
25.   }
26.   // 将插入点以后的内容复制到新内存中
27.   memcpy(lc + pos + len, m_pByte + pos, (m_size - pos - len) * sizeof(unsi
    gned char));
```

```
28.    // 将字节数组存放的数据指向新内存
29.    delete[] m_pByte;
30.    m_pByte = lc;
31.    return *this;
32. }
```

而它的数据区是这样存储的：

```
1.  private:
2.    unsigned char *m_pByte = nullptr; ///< 数据存放
3.    std::size_t m_size = 0;           ///< 字节数
```

通过上面，我们可以发现，该类的数据存储是基于栈区指针指向堆内存的模型的。这是非常直观的想法，当要存储的空间很大的时候开辟在堆内存是没毛病的，例如 C++标准库的 std::vector 就是这种模型，问题就在于开辟在堆内存以后的 insert()操作，从代码中我们可以很明显的看出，假设我要插入几个字节的数据，插到原字节数组的某位置，先开辟一段新空间用于存放新内存，在把原字节数组分半，然后在调用三次 memcpy 把三段字节拷贝到新内存当中，最后删除掉原内存，返回新数据指针。这样的确能达到目的，测试也确实测不出任何问题。但是问题就在于这样做的效率太慢了。在堆上操作空间就会消耗大量的性能，何况是这里，查一次就开辟新空间，还进行三次的 memcpy，这样得到的结果是绝对不可以接受的，因此经过评估，本类最后需要代码重构。

解决方案定下来了，下一步就是如何处理了。由于字节数组 LByteArray 和字符串 LString 有很大的联系，本质上来讲字节数组就是一个一个的字节，只不过有一些编码和解码的算法而已。由于 char 是一个字节，因此字节数组完全可以考虑继承或者复合 LVector<char>（在我进公司的时候，LVector 已经完全重构了，功能上也比较完善了）。[1]简单来讲 LVector 和标准库的 std::vector 没有太大区别，我的前辈做了一个两端有预留空间的优化，但是最核心的的内存模型是一样的，都是栈区的一根指针指向堆区的数据区。和上面有什么不一样呢？显而易见，vector 具有二倍扩容机制，并且采用更安全的分配器 allocator 来管理内存的声明和释放，这样就更安全和高效了，完全不需要每次 insert 都 new 新空间，再到处 memcpy 了。

其次，除了工程上问题的解决，这一部分我学到了更重要的一点：代码复用。代码复用在工程上是非常重要的一点。我从两个方面解读。第一，减少代码工程量。从数据结构举例，任何的数据结构都是基于数组和链表延伸出来的，例如二叉树是链表，哈希表是数组加链表等。如果把这两个最基本的类实现好了，理论上来说其他类就在这些类的基础上实现就可以了。[2]当然实际过程中肯定有更多的细节需要讨论，这里只是阐述大体思路。并且在上层的类当中就不需要考虑类似内存管理这种偏底层的问题了，后续的开发也会轻松许多。第二，方便 bug 调试。很多功能之间是具有复用的可能的。例如 append 和 prepend 函数就完全可以直接调用 insert 函数，当我把最基本的那个函数功能实现没有问题的时候，其他的功能自然而然就好了。[3]如果每个功能都从头手写，那么当代码出了问题的时候，去进行调试是一件非常麻烦和棘手的事情，同时这也是为了开发效率做考虑。

2.2 LarktestKit：代码流程构建

这部分是在审核 LarkTestKit 自动化测试框架的时候遇到的问题。

首先简单说明一下项目的背景。自动化测试框架的需求是为了完善合迅智灵产品的功能，同时也是顺应国产化的趋势。框架参考了开源的 googleTest，在基础上加入了其他的需求，例如模拟 UI 测试，最终得到的产品。而 LarkTestKit 代码库就是将源代码编译成为对应的静态库并做打包。目前在 Linux 环境下构建，后续功能齐全以后做 windows 适配。

而如何构建项目并完成自动化打包和静态扫描流程就成了我的工作，也是我遇到的问题。这是源代码的 src 架构，见图 2-1：

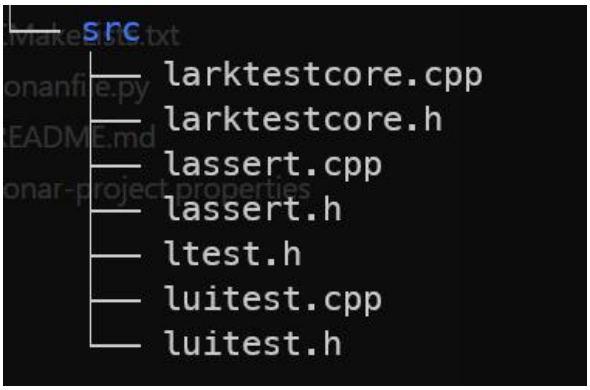

图 2-1

首先容易想到的就是，这不就是几个.h 和.cpp 文件嘛，既然目的是静态库，那我先用 g++ -c 编译成指定的.o 文件但不链接，然后在使用 Linux 自带的 ar 命令不就行了嘛。[4]的确，最底层的做法确实是这样的，但是显然直接这样干是行不通的，因为还有其他的流程内容，例如写 snippet 测试，使用 CI 自动化打包等，因此需要一个工具来统筹这些内容。

在 C++的工程当中，尤其是在 Linux 下，CMake 就是一个非常好的工具了。它提供了丰富的语法配置（windows 下使用 VS 全家桶比较好，当然 CMake 也是可以的），包括但不限于构建可执行文件，构建静态库、动态库等。[5]做好 CMake 配置以后，会自动生成 Makefile 文件，那就很显而易见了，使用 make 相关命令就能达到预期的效果了。关于这部分提到的工具的具体如何使用见第三部分知识技能学习情况。

有了思路以后，查找相关文档，编写了如下的 CMakeLists.txt：

```
1.  cmake_minimum_required (VERSION 3.12)
2.
3.  set (CMAKE_CXX_STANDARD 11)
4.  set (CMAKE_CXX_STANDARD_REQUIRED true)
5.
6.  if (NOT DEFINED PACKAGE_VERSION)
7.    message ("-- [LarkTestKit] WARNING: Package version is not defined, s
    et as 0.0.0")
8.    set (PACKAGE_VERSION "0.0.0")
9.  endif ()
```

```
10.
11. project("LarkTestKit"
12.   VERSION ${PACKAGE_VERSION}
13.   DESCRIPTION "合迅智灵 LarkStudio5 自动化测试框架"
14. )
15.
16. # import conan libs
17. message("-- [LarkTestKit] Start Configuring Conan ... ")
18. include(${PROJECT_BINARY_DIR}/conanbuildinfo.cmake)
19. conan_basic_setup(NO_OUTPUT_DIRS)
20. message("-- [LarkTestKit] Done configuring Conan.")
21.
22. set(CMAKE_C_COMPILER "gcc")
23. set(CMAKE_CXX_COMPILER "g++")
24.
25. message("-- [LarkTestKit] Using C Compiler: ${CMAKE_C_COMPILER} (${C
    MAKE_C_COMPILER_VERSION})")
26. message("-- [LarkTestKit] Using C++ Compiler: ${CMAKE_CXX_COMPILER}
    (${CMAKE_CXX_COMPILER_VERSION})")
27.
28. set(SOURCES
29.   src/larktestcore.cpp
30.   src/lassert.cpp
31.   src/luitest.cpp
32. )
33.
34. # import header files
35. include_directories(src)
36.
37. # build src into STATIC lib
38. add_library(larktestkit STATIC ${SOURCES})
39. set_target_properties(larktestkit PROPERTIES ARCHIVE_OUTPUT_DIRECT
    ORY ${CMAKE_BINARY_DIR}/lib)
40.
41. # build snippet
42. add_subdirectory(snippet)
43.
44. # make install
45. set(CMAKE_INSTALL_PREFIX "${PROJECT_BINARY_DIR}/install")
46. set(CMAKE_INSTALL_LIBDIR "lib")
47. set(CMAKE_INSTALL_INCLUDEDIR "include")
48.
49. # larktestkit
```

```
install(TARGETS larktestkit ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR})

# headers
install(DIRECTORY ${CMAKE_SOURCE_DIR}/src/ DESTINATION ${CMAKE_INSTALL_INCLUDEDIR} FILES_MATCHING PATTERN "*.h")
```

同时为了跑自己写的 snippet 测试代码，同步编写了 snippet/CMakeLists.txt：

```
# Get a list of directories with a CMakeLists.txt inside

file(GLOB ITEMS RELATIVE ${CMAKE_CURRENT_LIST_DIR} ${CMAKE_CURRENT_LIST_DIR}/*)
set(SUB_DIRS "")
foreach(ITEM ${ITEMS})
  if(
    IS_DIRECTORY ${CMAKE_CURRENT_LIST_DIR}/${ITEM}
    AND EXISTS ${CMAKE_CURRENT_LIST_DIR}/${ITEM}/CMakeLists.txt
    AND EXISTS ${CMAKE_CURRENT_LIST_DIR}/${ITEM}/.buildme
  )
    list(APPEND SUB_DIRS ${ITEM})
  endif()
endforeach()

get_property(PROP_GENERATOR_IS_MULTI_CONFIG GLOBAL PROPERTY GENERATOR_IS_MULTI_CONFIG)
message("-- [LarkTestKit] Multi-config generator flag: ${PROP_GENERATOR_IS_MULTI_CONFIG}")


# Run add_subdirectory() for each

foreach(SUB_DIR ${SUB_DIRS})
  add_subdirectory(${SUB_DIR})
  file(GENERATE OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/${SUB_DIR}/$<$<BOOL:${PROP_GENERATOR_IS_MULTI_CONFIG}>:$<CONFIG>/>lark.config" CONTENT "SdkPath=${PROJECT_BINARY_DIR}")
endforeach()

unset(PROP_GENERATOR_IS_MULTI_CONFIG)

message("-- [LarkTestKit] Done configuring snippets")
```

最后得到如下的执行效果，可以看到静态库成功构建，用于测试的可执行文件成功编译，见图 2-2：

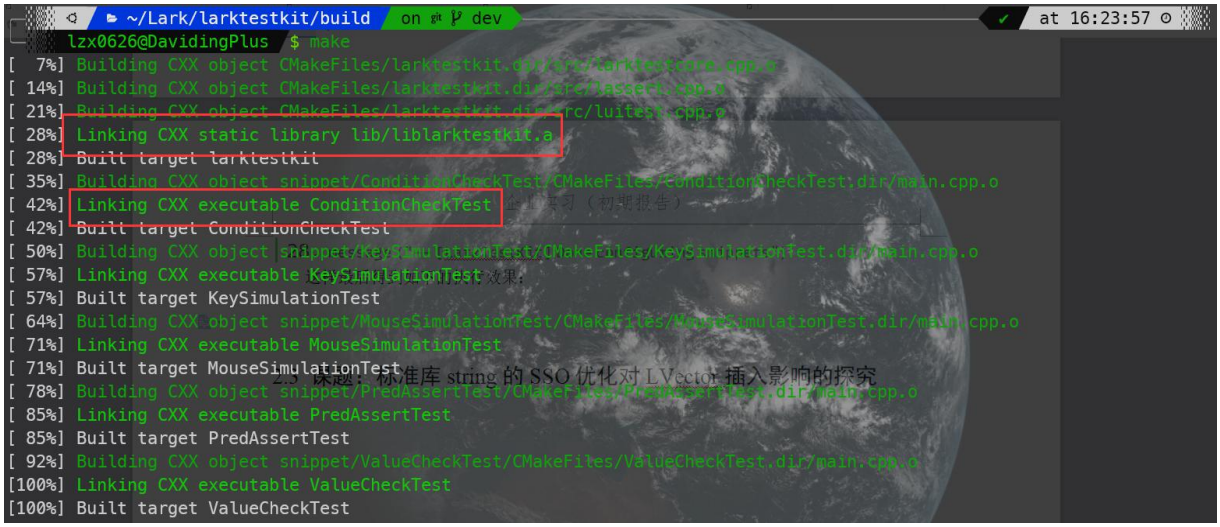

图 2-2

解决了本地编译的流程问题，就剩下自动化打包和静态扫描的问题了。公司的 git 代码库使用的是 gitlab，而 gitlab 每次的 commit 可以带有一个 CI 自动化脚本的功能，可以执行脚本命令，在 Linux 下面就是 shell 命令。因此对于自动化的问题，通过 CI 就解决了。而关于静态扫描，在公司的服务器上有扫描的可执行文件 cppcheck，因此只用编写静态扫描的启动脚本即可。

现在就只剩下打包的问题了。在实际开发的过程中，调用第三方库是一件非常普遍的事情，在前后端当中的开发当中尤其多。对于 LarkTestKit 和 LarkSDK 的研发，虽然使用的不多，仍然是需要一些开源的有协议的可以使用的第三方库的，例如 LarkSDK 中需要处理图片的 libpng 库，LarkTestKit 中需要依赖 LarkSDK 的部分功能等。因此如何管理这些第三方库就是一个问题了。这部分公司做了统一，公司使用的是 conan 包管理器，conan 是一个跨平台的可以管理 c++包的工具，具体使用见第三部分。因此问题就解决了，把 CMake 构建出来的静态库文件 liblarktestkit.a 和头文件一起通过 conan 打包，再上传到公司的 conan 服务器上，就完成整个流程了。同时测试那边由于没有权限直接处理研发的代码库，因此只能通过拉取 conan 包的方式，下载头文件和静态库做相关测试，这也同时也统一了研发和测试的工作流程。[6]

以下是 conan 的项目配置文件 conanfile.py：

```python
from conans import ConanFile, CMake, tools

class LarkTestKitConan(ConanFile):
    name = "LarkTestKit"
    version = "1.0.0"
    author = "liuzx01@sinux.com.cn"
    description = "合迅智灵国产 C++ 软件开发平台·自动化测试工具"
    url = "http://192.168.1.248/larkstudio/larktestkit"
    homepage = "https://www.sinux.com.cn"
```

```
    license = "Sinux License"
    requires = "LarkSDK/5.2.0@sinux/dev"
    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = {"shared": False, "fPIC": True}
    generators = "cmake"
    exports_sources = "CMakeLists.txt", "src/*", "snippet/CMakeLists.t
xt"

    def get_defs(self):
        defs = { "PACKAGE_VERSION": self.version }
        return defs

    def config_options(self):
        if self.settings.os == "Windows":
            del self.options.fPIC

    def configure(self):
        self.settings.compiler.libcxx = "libstdc++11"
        self.settings.compiler.cppstd = "11"

    def build(self):
        cmake = CMake(self)
        cmake.configure(build_folder = "build", cache_build_folder = ".",
 defs = self.get_defs())
        cmake.build()

    def package(self):
        # TODO use cmake install
        self.copy("*.h", dst = "include", src = "src")
        self.copy("liblarktestkit.a", dst = "lib", src="lib")

        # cmake = CMake(self)
        # cmake.configure(build_folder = "build", cache_build_folder = ".
", defs = self.get_defs())
        # cmake.install()


    def deploy(self):
        self.copy("*", dst = "LarkTestKit", excludes = "*.txt")
```

以下是 gitlab 的 CI 配置文件：

```
variables:
  yml_project_dir: ${CI_PROJECT_DIR}
  yml_commit_branch: ${CI_COMMIT_BRANCH}
```

```
    yml_is_daily: ${IS_DAILY}
    yml_short_sha: ${CI_COMMIT_SHORT_SHA}

  include:
    - project: 'LarkStudio/LarkPri'
      ref: main
      file: '/CI/yml/lark5.yml'


# Build

build_x86:
  variables:
    script_file_name: ./CI/build.sh
  extends:
    - .linux_gcc_x86_build_template
  rules:
    - if: $CI_COMMIT_BRANCH == "dev" || $CI_COMMIT_BRANCH == "main" || $CI_COMMIT_BRANCH =~ /^trial-.+/

build_arm:
  variables:
    script_file_name: ./CI/build.sh
  extends:
    - .linux_gcc_arm_build_template
  rules:
    - if: $CI_COMMIT_BRANCH == "dev" || $CI_COMMIT_BRANCH == "main" || $CI_COMMIT_BRANCH =~ /^trial-.+/

# Scan

scan:
  variables:
    script_file_name: ./CI/scan.sh
  extends:
    - .linux_sonarqube_template
  rules:
    - if: $CI_COMMIT_BRANCH == "dev"
```

以下是自动化脚本 build.sh 和 scan.sh：

build.sh:

```
#!/bin/bash

project_dir=$yml_project_dir
```

```
cd $project_dir


if [ $yml_commit_branch == "dev" ]
then

  if [[ -n $yml_is_daily ]]
  then
    build_config_name="daily"

  else
    build_config_name="dev"

  fi

elif [ $yml_commit_branch == "main" ]
then
  build_config_name="stable"

elif [ ${yml_commit_branch:0:6} == 'trial-' ]
then
  build_config_name="$yml_commit_branch"

else
  echo "Cannot proceed: undefined build config."
  exit 1

fi

echo "****** Build config is: $build_config_name ******"


echo "****** Build and Package with Conan ******"

conan user lark5 -p Sinux123_ -r larkstudio
conan user lark5 -p Sinux123_ -r lark3rdparty

conan create . sinux/$build_config_name -s compiler.libcxx="libstdc++11"
conan upload LarkTestKit/*@sinux/$build_config_name --all -r=larkstudio -c

echo "****** End of Build and Package ******"
```

scan.sh:

```
#!/bin/bash

project_dir=$yml_project_dir
cd $project_dir


echo "****** Run Cppcheck ******"
cppcheck -v --enable=all --platform=unix64 --xml ./src 2> cppcheck-report.xml
echo "****** Done Cppcheck ******"

echo "Debug: XML files in $project_dir:"
ls *.xml

echo "****** Run SonarScanner ******"
/opt/sonar-scanner/bin/sonar-scanner
echo "****** Done SonarScanner ******"
```

这部分主要完成了整个代码库从本地构建到自动化 CI 的流程。虽然每个部分都有相关的工具，但是如何使用这部分工具，并合理的与当前项目结合起来，培养了我将大问题化解为小问题，逐一解决的能力。LarkTestKit 的真正完成有待于后续进一步的工作。

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，做一个阶段性收获的总结记录。

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。目前我主要在 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。

关于 LarkTestKit 项目，实际上大同小异。由于目前是初版，很多功能等待后续实现。因此目前只在 Linux 系统上编译运行。后续功能完善以后再做 windows 等平台的适配，以达到跨平台的目的。

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

由于我们项目使用的是 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。另外由于 SDK 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，都会在项目中无时不刻不被运用。[7]

其次，由于 SDK 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 新知识点的学习

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。[8]

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等问题。Conan 包的安装也非常简单，只需要通过 pip 即可。[9]

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，并在需要时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。

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。

4 前期任务完成度与后续实施计划

本部分将总结到目前为止的任务完成情况，具体任务参见前面。并对未完成的任务做出反思，从而有助于对后期任务做安排和统筹，包括后期任务、后期复杂工程问题的思路方案、后期需要学习的知识技能等。

4.1 前期任务完成度

根据前面的任务，可以将前期任务分为三个部分，代码走查、代码优化重构、审核 LarkTestKit，具体见表 1-2。

第一，代码走查。目前已完成三大核心模块的走查工作，并作了文档的记录，见图 4-1。该部分的完成度为 100%。

图 4-1

第二，补充单元测试。目前已完成 util 和 network 部分的单元测试，使其从原有的 40%代码覆盖率分别上升到 75%和 90%。但是由于后续代码的修改，覆盖率目前有所下降，但是我一直维持单元测试的流程，一直保持数据更新。该部分的完成度 80%。

第三，代码优化重构。针对上述代码走查的问题，目前已经完成重构代码 LStack、LQueue，并已经与测试对接回归测试。同时优化 LPen、LBrush、LMenu、LMenuItem、LThread 等代码的部分问题，其他更大一点的问题已在文档中记录，等待领导后续统筹安排，具体可见图 4-2。该部分的完成度为 70%。

图 4-2

第四，审核 LarkTestKit。目前已完成代码的本地编译运行，Conan 打包和自动化 CI 流程，同时也完成和测试、校方同学的对接，自己也完成了部分测试代码的编写。后续等待继续沟通，尽早完成该部分的工作。该部分的完成度为 30%。

4.2 后续实施计划

针对上述的未完成任务，下表作了对于后续任务的规划，见表 4-2：

表 4-2 后期任务计划表

序号	工作内容	工作开始时间	持续时间
1	补充单元测试	2023.4.1	持续进行
2	代码优化重构	2023.4.1	大概一个月
3	审核 LarkTestKit	2023.3.25	大概一个月

5 参考文献

[1]：Drozd, Y.. "Rejection Lemma and Almost Split Sequences." Ukrainian Mathematical Journal (2020).

[2]： 杨浩,杨陟卓.基于面向对象编程的代码复用技术[J].信息技术,2017(06):52-57+61.DOI:10.13274/j.cnki.hdzj.2017.06.012.

[3]： Ullrich, Philip. "The Bug (Up Close and Personal)." APRIA Journal (2022).

[4]：Greenberg, M., Konstantinos Kallas, and N. Vasilakis. "Unix shell programming: the next 50 years." Proceedings of the Workshop on Hot Topics in Operating Systems (2021).

[5]： Hegde, Shrinidhi G, and G. Ranjani. "Package Management System in Linux." 2021 Asian Conference on Innovation in Technology (ASIANCON) (2021).

[6]： Butler, Branden A., and Ryan M. Richard. "CMinx: A CMake Documentation Generator." J. Open Source Softw. (2022).

[7]： Watson, R., et al. "CHERI C/C++ Programming Guide." (2020).

[8]： Martin, Kent, and B. Hoffman. "Mastering CMake : a cross-platform build system : version 3.1." (2015).

[9]：Tourani, Parastou, Bram Adams, and Alexander Serebrenik. "Code of conduct in open source projects." 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER) (2017).