



信息与软件工程学院

企业实习中期报告

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 企业实习的进展情况.....	1
1.1 实习工作完成情况.....	2
1.2 职业素养学习培养.....	4
1.3 总结.....	5
2 复杂工程问题和解决方案.....	6
2.1 标准库 string 的 sso 优化对 LVector 插入影响的探究.....	6
2.1.1 std::string 的优化.....	6
2.1.2 在 LVector 中插入 std::string.....	9
2.2 LDir 和 LFileInfo 的语义和设计.....	16
3 知识技能学习情况.....	24
3.1 开发环境和工具.....	24
3.1.1 开发环境.....	24
3.1.2 开发工具.....	24
3.2 新知识点的学习.....	24
3.2.1 C++ Boost 后备标准库的知识.....	24
3.2.2 Xml 基础知识.....	25
3.2.3 对 STL 的进一步理解.....	25
4 中期任务完成度与后续实施计划.....	27
4.1 中期任务完成度.....	27
4.2 后续实施计划.....	28
5 参考文献.....	29

1 企业实习的进展情况

本部分将从企业实习的进展情况出发，从实习工作完成情况和职业素养学习培养两个方面出发，阐述在中期的工作中，我在企业实习的大体情况，做一个阶段性的总结。

1.1 实习工作完成情况

在初期的工作中，我完成了三个主要模块中的部分代码走查，部分代码优化重构，以及完成 LarkTestKit 的审核。在中期的工作中，同理我的工作分为了代码走查、代码优化重构、课题调研以及协助 LarkTestKit 这四个部分，下面我将一一阐述。

第一，代码走查。在初期，我和项目组的成员们一起完成了三个主要模块 larkcore、larkutil 和 larkgui 的代码走查工作，而剩余三个扩展模块 larkdb、larknetwork 和 larkxml 暂未走查。因此在中期的工作中，首先的工作就是继续走查代码。但由于组内人员架构的调整以及公司工作重点的迁移，LarkSDK 的研发暂缓，故我暂时仅走查了 larkxml 模块，并将发现的问题和自己的思考记录于文档中，如图 1-1。



图 1-1 LXmlStream 走查文档

第二，代码优化重构。代码走查的目的是为了发现问题，记录问题，而最终的解决则需要后续的代码优化重构。在中期的工作中，我目前已完成 util 部分字符串列表（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，具体见图 1-2。另外，在组内成员整体中期的工作下，util 部分目前已经

	A	B	C	D		K	L	M	N	O	P	Q
--	---	---	---	---	--	---	---	---	---	---	---	---

模块	问题	C++ 类	兼容性评估	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-UI	字符串列表	LStringList	2件才能通过编译 -std=c++11	勉强可用	代偿重构	1. 直接继承Vector容器再实现相关功能 2. 直接继承String类，不实现数据类 3. 继承Vector再调用C++方法实现相关操作 4. 继承Vector再实现Sort排序 5. A3.0 更新：删除所有无用的接口和继承，等以外的运算符重载，只保留构造数、析构函数、= 运算符、以及 join 函数 6. 直接实现 LVariant 数据类实现	中	刘治宇	已完结	2024Q2
	哈希容器	LHash<Key, T>	不完全适用 (参考 QSet)， 到 LSet。	不可用	重新实现	以 std::unordered_map 重构	中	刘治宇	已完结	2024Q2
	集合容器	LSet<T>	不完全适用 (参考 QSet)， 到 LSet。	不可用	重新实现	以 std::unordered_set 重构	中	刘治宇	已完结	2024Q2
	字典容器	LMap<Key, T>	blue_type 是 T, key-blue 组成的pair,	不可用	重新实现	以 std::map 重构	中	刘治宇	已完结	2024Q2
	日期时间数据类型	LDateTime LTime	的又内联 和QIsValid()多余 冗余 使用int Time一般即初始化为	勉强可用	代偿修改	1. 简化代码，合理使用内联和宏 2. 目前存在循环修正，无需对无穷时地异常 3. 多个接口中多次使用了将DateTime转换为std::tm且代码较冗余，提供私有接口直接返回即可 4. 删除IsValid()接口 5. 操作运算符重载 <, <=, ==, >, >= 的实现直接对私有数据进行比较即可，且在实现 <, >= 时使用操作运算符重载 <, <=, == 即可 6. time_t 表示的时间戳值与参数传递和函数返回值一致，部分特殊的时间戳值保留 time_t 7. LDateTime 构造函数和析构函数修改为0 8. 设置时间的相关函数/时函数为三目运算符实现 9. LDateTimeFromString/GetLString 接口需要传入的格式进行识别 10. LDateTime的操作运算符不使用时，直接LDateTime即可 11. 直接使用循环修正的时间设置即可，不需要IsValid()且不需要返回bool	中	刘治宇	已完结	2024Q2
	入的待读字符串									

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。在中期的工作中，我很幸运完整负责了一些课题的完整调研，例如标准库 `string` 的 `sso` 优化，`LDir` 和 `LFileInfo` 的设计架构，以及目前正准备开展的 `QGraphicsScene` 调研等。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，具体见图 1-3。

刘治学 - LDir/LFileInfo	刘治学	4月19日 17:53	...
刘治学 - LMap/LMultiMap	刘治学	4月25日 11:36	...
刘治学 - LMutex/LReadWriteLock/LPlatformThreadInterface/LThread/LThreadData	刘治学	3月4日 11:00	...
刘治学 - LObject/LApplication/LSignal	刘治学	2月23日 11:06	...
刘治学 - LPen/LBrush/LLinearGradient/LMenu/LMenuItem	刘治学	3月18日 13:21	...
刘治学 - LStack/LQueue/LByteArray	刘治学	2月23日 15:22	...
刘治学 - LXmlStream	刘治学	今天 10:37	...
刘治学 - 标准库string的sso优化对LVector插入影响的探究	刘治学	今天 10:54	...

第四，协助 LarkTestKit。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。

1.2 职业素养学习培养

作为一名软件工程师，尤其是一个从事军工行业的企业的软件工程师，我更应该提升自己的思维意识，自身素养，保质保量完成公司指派给我的任务，同时做好保密工作，严格维护公司和自身的权益，努力在工作的同时提升自己的技术水平，进入一个正向反馈的过程。在实习的过程中，我逐渐学到了技术层面以外，更多是程序员职业素养的知识，下面将进行阐述。

首先是重视设计和思考。由于 LarkSDK 产品是完全从头开始，独立自研的，我们的目标是对标 QT，甚至是替代乃至超越 QT。不可否认我们参照了行业巨头 QT 的很多思路，但是人无完人，强如 QT 的设计也存在很多不合理，有缺漏的地方。[1]当然 QT 自己也在后续的版本中自己在努力修改，但是 QT 毕竟是一个完整的产业链产品，有些东西久了是改不掉的。而 LarkSDK 作为一个新鲜血液的产品，并且目前还非常轻量。我们有必要，并且是必须在搭建底层地基的时候就考虑好整体的架构和设计，例如如何设计窗口和组件的关系，如何设计跨平台的统一管理调度等。这些功能不仅需要稳定可用，还需要为后续研发的上层功能留出口子。[2]当然这个过程是一个非常复杂、繁琐并且困难的过程。在这个过程中，设计花费的时间远大于真正写代码的时间。事实证明，这样的策略是行得通的，目前 LarkSDK 的架构和语义非常明确，功能也完全符合预期。重设设计和思考这一点在后续我自己负责的 LDir 和 LFileInfo 中体现的尤为突出。

其次是多阅读，多记录。我的老板曾经讲过，提升代码水平的最好办法就是阅读代码。在我个人看来，不仅是多阅读代码，更需要多阅读文档，同时多做好记录。单打独斗是成不了气候的，每个人都存在知识的盲区，因此需要不断学习，不断提升自己，当然在当今社会也是为了不被淘汰。同时，最好做好知识的记录，就像初高中在课本上做笔记一样，知识太多，难免会忘记，因此需要做好记录，后续方便复习回顾。我个人一直有写博客的习惯，后续回顾以前知识的时候能很快回想起来，同时这样也能增加心中的成就感，为自己的前进提供动力。在我调研标准库 `std::string` 的 sso 优化的时候，查询了很多博客和技术文档，最终才把具体 sso 优化的细节，它具体的内存模型是怎样的等问题研究明白。这样清楚以后应用到我们的 LarkSDK 中，问题自然就迎刃而解了。后续第二部分会具体阐述这部分的具体技术细节。

最后，也是我觉得最重要的一点，端正态度，脚踏实地。没有解不出来的问题，有的只是半途而废的人。我的组长是一位技术非常厉害，同时也非常谦逊的行业前辈。他对组员很有信心，愿意给我们安排各种各样的需求工作，培养我们的技术和能力，当然，他自己也以身作则，带头完成项目中最困难的部分。我觉得有这样的领导和前辈，是我的幸运，甚至我认为以后真正工作以后都很难再遇

到这样的领导了。同时，从他身上我也看出了，问题并不是不可战胜的，只要不断探索，肯下功夫，并且团队协作，一起努力，问题就一定能迎刃而解。在这个过程中，技术和能力也就自然而然的提升了。

1.3 总结

截止到目前的任务，按照迭代周期时间线，见如下汇总表 1-1。

表 1-1 目前完成的工作

迭代周期	具体完成工作
4.1 - 4.3	1. 完成实习初期阶段的首尾工作
4.7 - 4.19	1. 走查 Xml 模块代码 2. 继续维护单元测试 3. 协助 LarkTestKit 完成初步验收
4.22 - 4.30	1. 调研标准库 <code>std::string</code> 的 sso 优化对 <code>LVector</code> 的插入影响 2. 完成关联容器 <code>LHash</code> 、 <code>LMap</code> 、 <code>LSet</code> 的重构
5.6 - 5.18	1. 调研 <code>LDir</code> 和 <code>LFileInfo</code> 的语义和设计，进而完成 <code>LFilePath</code> 和 <code>LFileSystemEntry</code> 类 2. 完成字符串列表 <code>LStringList</code> 、三元组容器 <code>LTrio</code> 的优化修改
5.19 - 5.31	1. 完成日期时间数据类型 <code>LDateTime</code> 、 <code>LDate</code> 、 <code>LTime</code> 的重构 2. 继续维护单元测试

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 标准库 string 的 sso 优化对 LVector 插入影响的探究

LVector 在插入 `std::string` 的时候遇到了问题，我的组长研究了一段时间，我接着他的成果继续探讨，将学到的内容总结在这里。

2.1.1 `std::string` 的优化

标准库的 `std::string` 其实是做了优化的，不同的编译器实现的细节可能不同，但是基本的大思路框架都是一样的。

注：以下都是理论上的思路分析，具体的底层代码请自行查阅资料。

1、基本内存模型

我们熟知的 `std::string` 的内存模型大致是这样的，具体见图 2-1：

- (1) 栈区当中存放容量 `capacity`，大小 `size` 和一根指向堆区数据区域的指针，三个分别占据 8 字节，总共 24 字节。
- (2) 堆区当中 `data` 是实际存放数据的地方，通过分配器分配出来的（默认使用 `std::allocator`，实际就是 `new` 出来的），`std::string` 中的 `c_str()` 方法获取的就是堆区这个数据区首地址。[3]

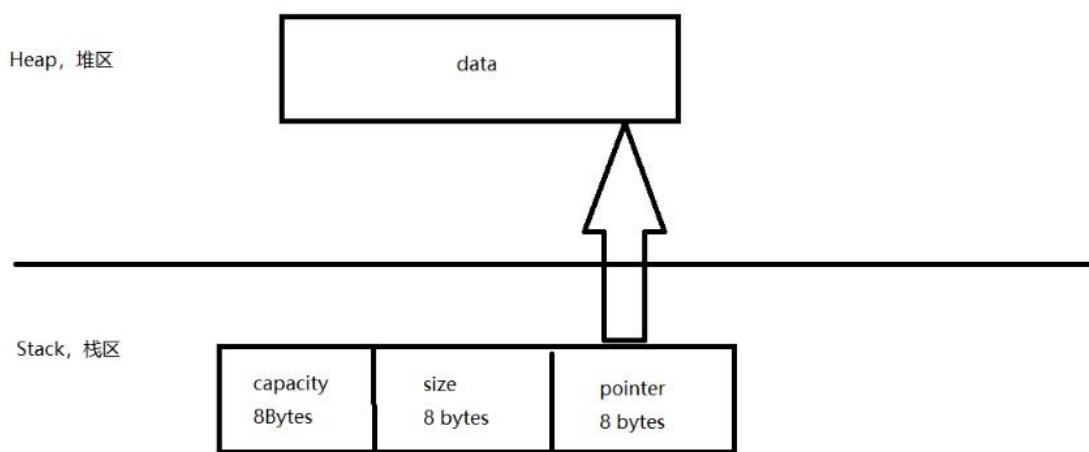


图 2-1 `std::string` 基本的内存模型图

2、COW 优化（现在一般不用）

COW，即为 Copy-On-Write，写时拷贝。

提到这里，我首先想到了 Linux 当中父子进程的"读时共享，写时拷贝"，父子进程在读的时候共享用户区的数据，例如先 `open()` 一个文件，在 `fork()`，父子进程的文件描述符是同一个，具体可以表现为父进程读 2 个字节，对于子进程的文件偏移指针也向后偏移 2 个字节；这就是因为用户区的文件描述符表是读时共享的；当需要修改用户区的数据，比如一个变量，就会做拷贝操作了，这点毋庸置疑，这也是优化性能的一种策略。[4]

参上，`std::string` 的 COW 优化也是一个道理，具体图 2-2：

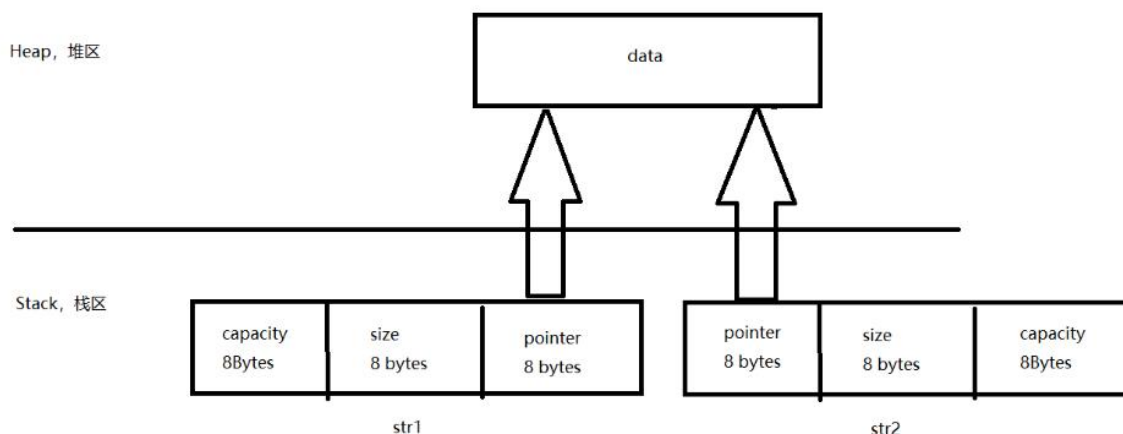


图 2-2 COW 优化的内存模型图

当然，COW 也会存在一定的问题。看到上面，可能会想，COW 这么好，为啥标题还是现在一般不用呢？之所以不用，是因为这种机制在多线程当中可能会出现不可预期的乱七八糟的问题，具体自行查阅资料，这里不作阐述。

3、SSO 优化

所以就有了 SSO 优化，即 Small String Optimization，翻译过来就是短字符串优化。

那么为什么需要短字符串优化呢？看基本的内存模型，如图 2-3：

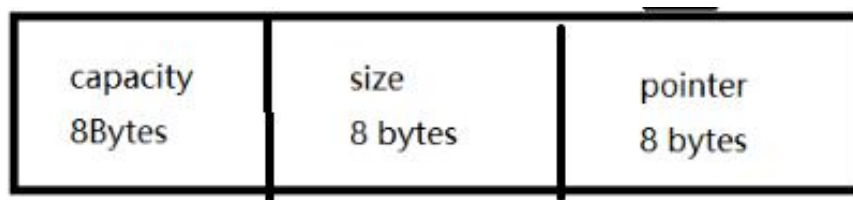


图 2-3 SSO 优化的内存模型图

想象一下，如果我的字符串比较短，举个例子，小到 8 个字节就能存下，那么是不是就不用存一个指针了，直接在栈区存储即可，还不用去堆区开辟空间，还不用考虑堆区内存释放的问题，岂不美哉？

我们再考虑一下，8 个字节的 `capacity` 和 8 个字节的 `size` 最大能表示多少的数？ $2^{63}-1$ ，这也太大了吧，完全没必要，因此 `capacity` 和 `size` 也可以

做进一步优化，注意不同编译器的实现不同，但是思路都是这样，能砍的就砍。当然不管如何，里面存放的 `pointer` 是不会变的，因为 `std::string` 中还有 `c_str()` 接口，不能让功能变了。大概的优化模型如下，可以看到数据在栈区，这个时候指针指向自身内部的 `data`，合理，非常合理。[5]

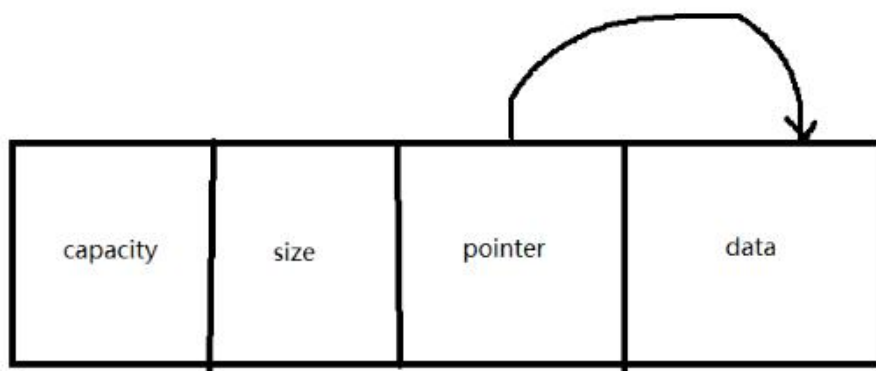


图 2-4 SSO 优化的指针指向

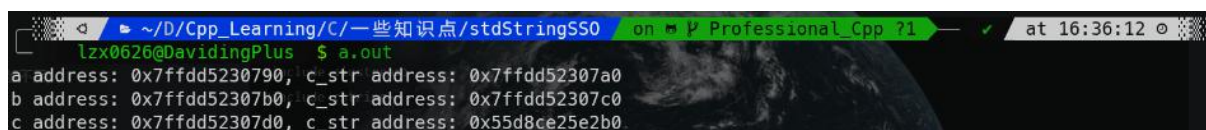
当字符串的长度变长，达到长字符串的标准（不同编译器的规定不一样），就会恢复一般的内存模型。

关于 SSO 优化的下的字符串的拷贝，由于内存模型中仍然存在指针，显然是一个深拷贝，可以写一个程序测试一下，顺便看一下 `std::string` 的 SSO 优化的表现。

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string a{"one"};
7.     auto b = a;
8.
9.     std::string c{"twooooooooooooooooooooooooooooooooooooooooooooo
    oooo"};
10.
11.     printf("a address: %p, c_str address: %p\n", &a, a.c_str());
12.     printf("b address: %p, c_str address: %p\n", &b, b.c_str());
13.     printf("c address: %p, c_str address: %p\n", &c, c.c_str());
14.
15.     return 0;
16. }
```

执行结果，可以发现完美验证了我们的分析，具体如图 2-5:

- (1) a 到 b 经过了一次深拷贝，他们两个的数据区地址不同。
- (2) a、b 的本类地址和数据区地址非常相近，而 c 离的很远。



```

~D/Cpp_Learning/C/一些知识点/stdStringSSO on P Professional_Cpp ?1 at 16:36:12
lzx0626@DavidingPlus $ a.out
a address: 0x7ffdd5230790, c_str address: 0x7ffdd52307a0
b address: 0x7ffdd52307b0, c_str address: 0x7ffdd52307c0
c address: 0x7ffdd52307d0, c_str address: 0x55d8ce25e2b0

```

图 2-5 sso 优化的测试结果

当然 SSO 优化也存在一定问题，这就不是这个课题的重点了，后续自行查阅资料。

2.1.2 在 LVector 中插入 std::string

测试 LVector 的时候，发现 `prepend std::string` 的时候程序崩溃，返回的值也不符合预期。

我们先不管这个 LVector 是如何实现的，我们知道 `prepend` 函数，肯定是调用 `insert` 方法，所以去查 `insert` 函数，以下给出该部分的相关代码。

真正做插入的函数叫 `insertMultiple()`，里面有一些算法，不用管他，我们考虑 `insert` 函数的逻辑，在目标处进行插入，然后需要把其他的数据前移或者后移，也就是说，需要做内存的移动或者拷贝，问题就处在这里，也就是代码中的 `moveMemory()` 中。[6]

```

1. template <typename T>
2. inline void LPaddedVector<T>::moveMemory(T *pTargetAddress, T
   *pSourceAddress, std::size_t count)
3. {
4.     std::memmove(pTargetAddress, pSourceAddress, count * sizeof
      (T));
5. }

```

可以发现，`moveMemory` 是直接调用的 `std::memmove`，直接把原内存给移动过去了，乍一看好像没什么毛病，搬运就搬运呗，但是注意，使用了 `std::memmove`，对象仅仅是换了一个位置，里面的数据什么都没变，现在把这个同 `std::string` 的 SSO 优化结合起来。

从左边移动到右边，`capacity`、`size`、`data` 都没有问题，关键在于这个指针，前面说过，里面的数据仅仅是换了一个位置，那指针指向的还是原先的地址啊，而原先的地址现在如何？不知道，可能被覆盖，可能被释放了，因此就会出现上面的问题。

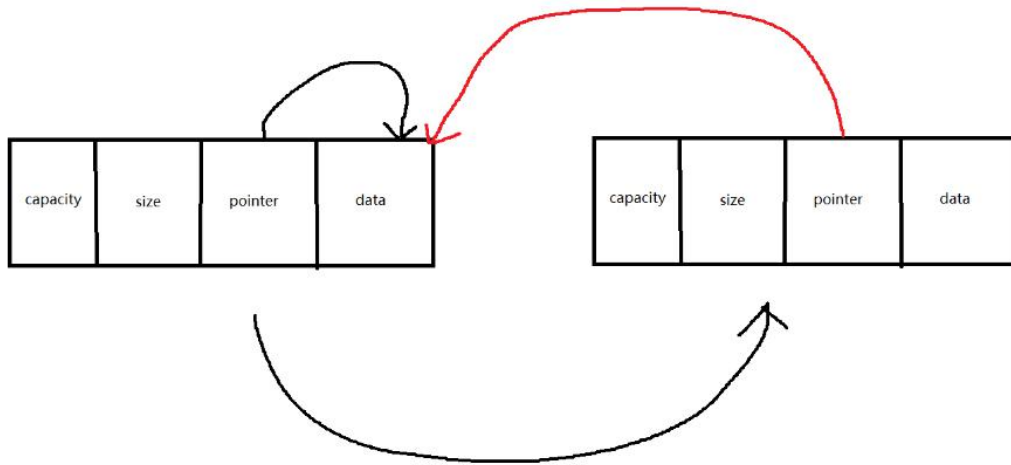


图 2-6 memmove 的内存模型图

分析了问题的来源，那解决问题就好办了，比如可以为 `std::string` 做特化，让他在这里使用拷贝的策略，这样能解决问题。

注意：这里的拷贝和前面移动导致的类似浅拷贝不一样，这里的拷贝是通过分配器构造，实际上调用的是 `std::string` 的拷贝构造函数，不管 `std::string` 是哪种优化方式，深拷贝他是必然做的，也就是说那根 `pointer` 就指向的是自身的 `data` 而不是之前的了，这样就是对的。

```

1.  template <>
2.  inline void LPaddedVector<std::string>::copyConstruct(std::string *pTargetAddress, const std::string &itemToCopy)
3.  {
4.      sm_allocator.construct(pTargetAddress, itemToCopy.data(),
5.                             itemToCopy.size());
6.
7.  template <>
8.  inline void LPaddedVector<std::string>::moveMemory(std::string *pTargetAddress, std::string *pSourceAddress, std::size_t count)
9.  {
10.     if (pTargetAddress < pSourceAddress)
11.     {
12.         for (int i = 0; i < count; i++)
13.         {
14.             sm_allocator.construct(pTargetAddress + i, pSourceAddress[i].data(), pSourceAddress[i].size());
15.             sm_allocator.destroy(pSourceAddress + i);
16.         }
17.     }

```

```

18.  else if (pTargetAddress > pSourceAddress)
19.  {
20.      for (int i = count - 1; i >= 0; i--)
21.      {
22.          sm_allocator.construct(pTargetAddress + i, pSourceAddress[i].data(), pSourceAddress[i].size());
23.          sm_allocator.destroy(pSourceAddress + i);
24.      }
25.  }
26.  }

```

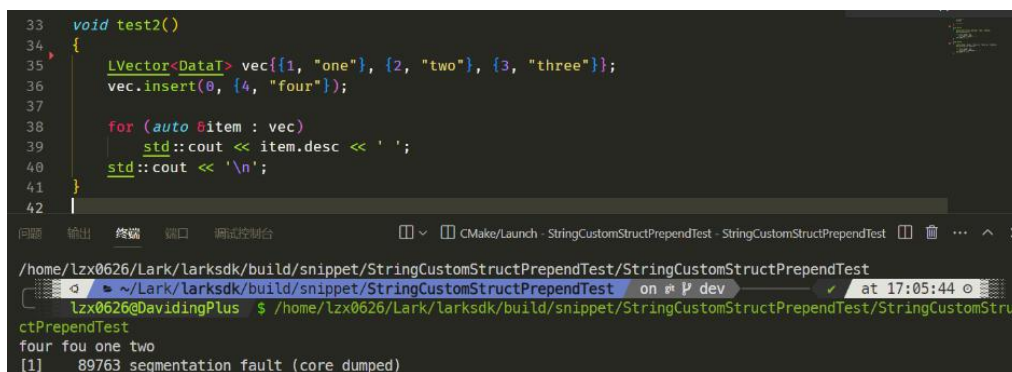
那么问题来了，如果有一个自定义类型，里面含有 `std::string`，那又该怎么办呢？例如下面的测试，结构如下：

```

1.  struct DataT
2.  {
3.      int x;
4.      std::string desc;
5.  };

```

测试结果如图 2-7：



```

33 void test2()
34 {
35     LVector<DataT> vec{{1, "one"}, {2, "two"}, {3, "three"}};
36     vec.insert(0, {4, "four"});
37
38     for (auto &item : vec)
39         std::cout << item.desc << ' ';
40     std::cout << '\n';
41 }
42
/home/lzx0626/Lark/larksdk/build/snippet/StringCustomStructPrependTest/StringCustomStructPrependTest
lzx0626@DaVidingPlus: $ /home/lzx0626/Lark/larksdk/build/snippet/StringCustomStructPrependTest/StringCustomStru
ctPrependTest
four fou one two
[1] 89763 segmentation fault (core dumped)

```

图 2-7 自定义类型的测试结果

我第一反应想到的就是能不能用 c++ 通过某种手段判断一个类当中是否含有指定类型例如 `std::string` 的成员变量，但由于水平不够，或者因为本来就不太好使，这条路走不通。

所以就只能从刚才提到的 `moveMemory()` 入手了，既然直接移动不好，那我干脆改成拷贝不行吗？当然不好，白白多了很多次拷贝，这是不可接受的，那有没有办法将二者结合起来呢？你别说，还真有。

参考了 Qt 的部分实现，Qt 中封装了一个叫 `QTypeInfoQuery` 的类，里面有一个变量 `isRelocatable`，这个东西可以用来判断类能否平凡可复制，顾名思义，像 `std::string` 显然不能平凡可复制，因为 SSO 的优化，平凡复制的话指针指向的地方是原来的，显然不行，说白了就是类似浅拷贝，因此这里做了判断，如果不

行就拷贝，可以就移动[7]。具体如图 2-8：

```

typename QVector<T>::iterator QVector<T>::insert(iterator before, size_type n, const T &t)
{
    Q_ASSERT_X(isValidIterator(before), "QVector::insert", "The specified iterator argument 'before' is invalid");

    const auto offset = std::distance(d->begin(), before);
    if (n != 0) {
        const T copy(t);
        if (!isDetached() || d->size + n > int(d->alloc))
            realloc(&alloc, d->size + n, options: QArrayData::Grow);
        if (!QTypeInfoQuery<T>::isRelocatable) {
            T *const e = d->end();
            T *const b = d->begin() + offset;

            T *i = e;
            T *j = i + n;

            // move old elements into the uninitialized space
            while (i != b && j > e)
                new (--j) T(std::move(*--i));
            // move the rest of old elements into the tail using assignment
            while (i != b)
                *--j = std::move(*--i);

            // construct copies of t inside the uninitialized space
            while (j != b && j > e)
                new (--j) T(copy);
            // use assignment to fill the recently-moved-from space
            while (j != b)
                *--j = copy;
        } else {
            T *b = d->begin() + offset;
            T *i = b + n;
            memmove(static_cast<void*>(i), static_cast<const void*>(b), (d->size - offset) * sizeof(T));
            while (i != b)
                new (--i) T(copy);
        }
        d->size += n;
    }
}

```

图 2-8 QTypeInfoQuery 的部分源码

关于 `std::string` 堆内存那个模型，是满足平凡可复制条件的，画个图如下理解。

`std::memmove` 不会触发类的析构函数，因此堆内存还在，不会被释放，因此就做到了完美迁移，同时避免了不必要的栈内存和堆内存的拷贝，提升了效率，具体如图 2.9：

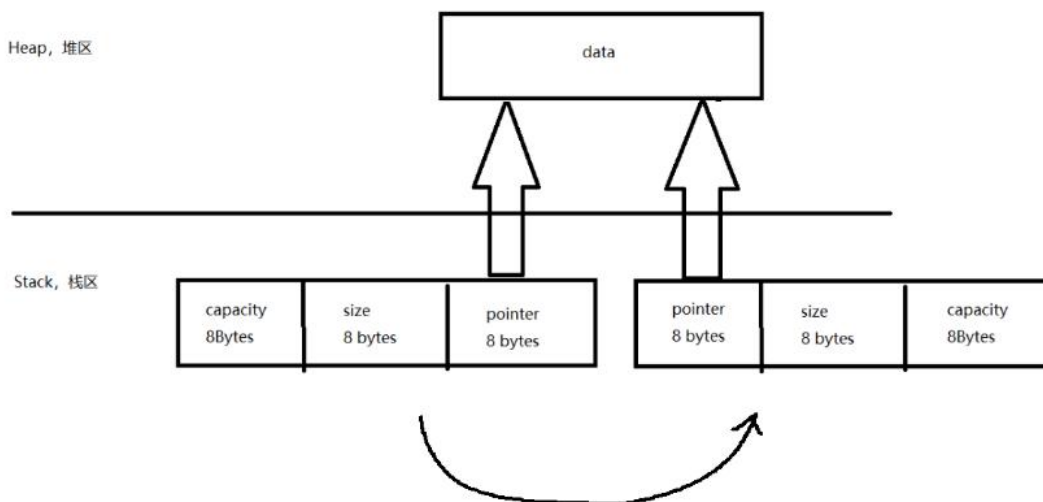


图 2-9 完美迁移的示意图

在查看了 `QTypeInfoQuery` 的 `isRelocatable` 的来源之后，我发现调用的是标准库的一个 `type_traits`，叫 `is_trivially_copyable`，可以判断是否可以平凡可复制，平凡可复制的含义见上。

这里写了一个程序测试：

```
1. #include <iostream>
2. #include <type_traits>
3.
4. struct DataT1
5. {
6.     int x;
7.     std::string str;
8. };
9.
10. struct DataT2
11. {
12.     int x;
13.     DataT1 t;
14. };
15.
16. struct DataT3
17. {
18.     int a;
19.     int b;
20.     int c;
21.     int d;
22.     int e;
23.     int f;
24.     int g;
25. };
26.
27. class DataT4
28. {
29.
30. public:
31.     DataT4() { p = new int(0); }
32.
33.     ~DataT4()
34.     {
35.         if (p)
36.         {
37.             delete p;
38.             p = nullptr;
39.         }
```

```

40. }
41.
42. int *p = nullptr;
43. };
44.
45. int main()
46. {
47.     std::cout << std::is_trivially_copyable<DataT1>::value << '
        \n';
48.     std::cout << std::is_trivially_copyable<DataT2>::value << '
        \n';
49.     std::cout << std::is_trivially_copyable<DataT3>::value << '
        \n';
50.     std::cout << std::is_trivially_copyable<DataT4>::value << '
        \n';
51.
52.     return 0;
53. }

```

执行结果如下：

(1) 前两个类由于具有 `std::string`（第二个类套娃，也算），返回 0，不可平凡可复制。

(2) 第三个类，全是一些 `int`，显然可以。

(3) 第四个类，堆内存，这个设计和一般的 `std::string` 是一样的，返回的是 0。

好，现在问题来了，我们刚才说借助 `type_traits` 来进行判断，好决定是通过移动还是通过拷贝，代码甚至我都写好了。

```

1. template <typename T>
2. inline void LPaddedVector<T>::moveMemory(T *pTargetAddress, T
    *pSourceAddress, std::size_t count)
3. {
4.     if (std::is_trivially_copyable<T>::value)
5.     {
6.         // 如果是平凡可复制类型, 可以直接使用 std::memmove
7.         std::memmove(pTargetAddress, pSourceAddress, count * sizeof
            f(T));
8.     }
9.     else
10.    {
11.        // 如果不是, 逐个元素的拷贝
12.        // 注意源地址和目标地址的位置不同, 拷贝的顺序也不同
13.        if (pTargetAddress < pSourceAddress)
14.        {
15.            for (int i = 0; i < count; ++i)

```



```
16.    {
17.        sm_allocator.construct(pTargetAddress + i, *(pSourceAd
    dress + i));
18.        sm_allocator.destroy(pSourceAddress + i);
19.    }
20. }
21. else if (pTargetAddress > pSourceAddress)
22. {
23.     for (int i = count - 1; i >= 0; --i)
24.     {
25.         sm_allocator.construct(pTargetAddress + i, *(pSourceAd
    dress + i));
26.         sm_allocator.destroy(pSourceAddress + i);
27.     }
28. }
29. }
30. }
```

我们还是考虑刚才的堆内存模型，经过判断之后是走拷贝这一条路，但是没必要啊，`std::memmove` 不会激活对象的析构函数，我把所有数据移动到另一个地方，这个指针还是指向堆区的这一块内存，也就是说，实际上这个内存模型是平凡可复制的，这也是组长的最初的想法（我研究了这么久才到大佬的初步想法，差距不是一点半点），这一点优化，就导致了最开始的问题，但是我们必须要有这种思维。

那么就没有解决方案了吗？

其实是有的，可以在这个判断之前再加上一层判断，例如某个类就是这种堆内存模型，经过分析他其实是可以平凡可复制的，那么我通过某种手段，例如宏，在那个类当中提供一种注册的方式，我保证这个类的行为是平凡可复制的，执行到这里之后先执行这个判断，如果 ok，那直接走移动的道路，目前这样来看比较合理。这也是我和组长目前总结之后的研发需求。

2.2 LDir 和 LFileInfo 的语义和设计

QDir 和 QFileInfo 的语义一直以来都比较令人费解。我们知道文件和目录的关系是：目录是一种特殊的文件。按照 QDir 和 QFileInfo 的命名来讲，应该是 QDir 管理目录，QFileInfo 管理文件，但是实际上这两个类的功能是非常混乱的，QDir 可以操作文件，QFileInfo 也可以操作目录。而初版的 LDir 和 LFileInfo 也是完全按照 QT 的思维走的，因此导致该部分的语义非常混乱，让我们和用户感到非常费解。[8]

注意，该任务不包括 LFile 的部分，只关心 LDir 和 LFileInfo 的语义和设计问题。LFile 是一类，需要通过 IO 进行文件操作，类似于 IODevice。而 LDir 和 LFileInfo 是一类。

现在的设计方式是直接存储一个 LString 类型的路径，但是这样对于 windows 平台非常不友好。QT 也是这样做的，不过可能是因为历史原因，未能按照更好的方式修改。但是 QT 的功能经过庞大的迭代是很稳定的，但是这部分初版的功能一言难尽。

同时，目前 LDir 和 LFileInfo 的语义非常不明确。为了避免混淆，将二者重新命名为 LFilePath 和 LFileEntry，后续 LDir 作为 LFilePath 的别名，LFileInfo 作为 LFileEntry 的别名。经过我的思考和组内集体的讨论，有了如下的设计。

1、LFilePath：存储规范化后的路径的结构。

LFilePath 讨论以后的设计是能够将用户传入的路径进行合理并且严格的规范化，本类只负责这个功能。至于该路径指向的具体是文件还是目录，该路径指向的文件或目录是否存在，有什么权限，本类不关心。按照此语义，本类应当不会涉及与平台相关的具体接口。

2、LFileEntry：内部存储一个路径结构 LFilePath，真正与系统 API 打交道的类。

LFilePath 对路径做了严格规定，因此在 LFileEntry 中会存储这个结构用于处理系统中的路径。在本类当中就会提供平台相关的借口了，比如创建目录 mkdir，进入目录 cd，文件权限 permission 等等操作，当然，这些操作都离不开路径结构 LFilePath。

与 Qt 不同的是，LFileEntry 中存储的路径指向的文件或目录都必须是在系统中实际存在的，不允许存储不存在的路径，因为这样没有意义。如果非要存储，请使用单纯的路径结构 LFilePath。

3、二者的命名规定。

对于大多数用户而言，可能并不知道目录是一种特殊的文件（目录和文件都

具有 `rwX` 权限，只是表现方式不同），因此本类的名称有待商榷，目前的商讨结果是将 `LDir` 作为另一个类（例如 `LFileSystemPath` 这种）的别名，类似 `LVector` 和 `L PaddedVector` 的关系。保留 `LDir` 是考虑了 Qt 的缘故。`LFileSystemEntry` 与 `LFileInfo` 的关系同理。

下面会有更多的细节值得商讨研究。

1、`LFileSystemPath` 如何存储路径。

经过对比 QT 和与组长的讨论，决定使用 `StringList` 存储各级目录名和文件名的方式。

windows 和 linux 文件系统最大的区别就在于 windows 使用反斜杠 `\`，linux 使用正斜杠 `/`，但是这些在存储的时候根本没有必要存储，因此直接使用 `StringList` 存储文件名即可。下面会涉及到更多的细节。

第一，如何判断绝对路径还是相对路径。

在 Linux 下，绝对路径以 `/` 开始。在 windows 下同理，绝对路径是盘符加上 `\`，例如 `c:\`，用户传递进来的路径经过我们内部的 `split` 处理存入到一个一个的 `StringList` 元素中，很容易联想到，在开头作标识即可。如果是绝对路径，那么 `StringList` 开头会留出一个空元素，相对路径则不会。

例如，对路径 `/path/to/./to/local`，就是这样的结构，如图 2-10：

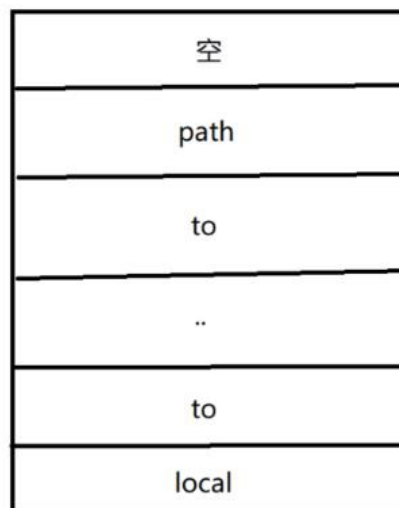


图 2-10 `/path/to/./to/local` 的结构

第二，如何判断末尾是文件还是目录。

文件和目录很有可能出现同名的情况，这是必须要考虑的事情。（这和文件是否具有后缀没有关系，目录也可以写成带后缀的形式）我们需要想办法在 `LFileSystemPath` 的结构里面表现出来。结合上面的例子，如果我的路径是

/path/to/./to/local/的话，就一定代表 local/是一个目录了，具体在系统中到底存不存在本类不关心，可以效仿刚才的做法，在末尾加上一个空元素，即可区分。[9]

对于文件还是目录，在本类中做了严格的规定，local 是文件，local/是目录。之所以要这样做，请看下面 LFileSystemEntry 提出的联想问题。

对于路径/path/to/./to/local/，就是这样的结构，如图 2-11：

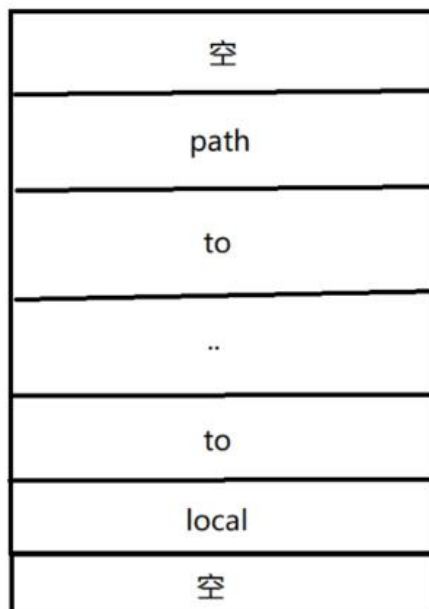


图 2-11 /path/to/./to/local/的结构

至此，我们就在 LFileSystemPath 的层面对绝对路径和相对路径，文件和目录进行了严格的规定。

2、LFileSystemEntry 构造时关于文件和目录的修正。

在构造的时候会涉及到一个问题，对于 LFileSystemPath 而言，我们硬性对目录和文件做了规定，目录结尾必须有/，文件没有。但是对用户而言，可能并不知道，因此用户可能想要目录，也可能写入/path/to/./to/local，如果系统中这个 local 的确是一个目录，并且没有要给同名的 local 文件存在，这是没有毛病的。这个问题需要进行处理。

比较合理的解决方式是，首先由于无论是在 linux 还是在 windows 下，文件和目录不能同名，也就是不能同时出现 test 和 test/。对于 LFileSystemEntry 而言，对于文件类型的路径，既能匹配到文件也能匹配到目录，因此如果匹配成功，需要做二次匹配，匹配对应的目录，如果匹配失败，代表是一个文件路径；如果匹配成功，需要对此时的 LFileSystemPath 存储的内容做了修正，变成了/path/to/./to/local/。当然用户如果传入/path/to/./to/local/，那一定匹配的是目录，这一点毋庸置疑。

3、考虑盘符。

windows 下存在盘符，例如路径 `d:\a\b\c`，其中 `d:` 就代表盘符，`\a\b\c` 就是从 `d:` 盘符下的根目录开始的依次的 `a`，`b` 目录和 `c` 文件，那么对应到 linux 下面呢？例如 `d:/a/b/c`，这就是一个正确的相对路径了，分别对应 `d:`，`a`，`b` 目录和 `c` 文件，可见如何处理盘符是一个非常重要的问题。

其次，考虑了盘符以后，我们需要考虑 `LFilePath` 和 `LFileSystemEntry` 的对接问题。例如我给出路径 `d:a\b\c`，对应盘符 `d:` 和相对路径 `a\b\c`。我们上面专门谈到过 `LFilePath` 不关心这个路径是否合法，指向的东西是否存在，只是做一个规范化的存储的数据结构。因此在这里而言，理论上讲，这个路径下的盘符是没有意义的，因为给出的是一个相对路径，而 `LFilePath` 的结构为 `LFileSystemEntry` 服务的时候，真正的工作路径的盘符与用户给出的盘符可能并没有联系，因此经过讨论，这里的盘符会被忽略掉。也就是说如果调用 `path()` 方法导出路径，会得到 `a\b\c`，盘符就没有了，这也是比较符合逻辑的。[10]

当然。上面只是考虑了一种情况，实际的情况可能是有无盘符和绝对相对路径的综合情况，故作下图进行总结：

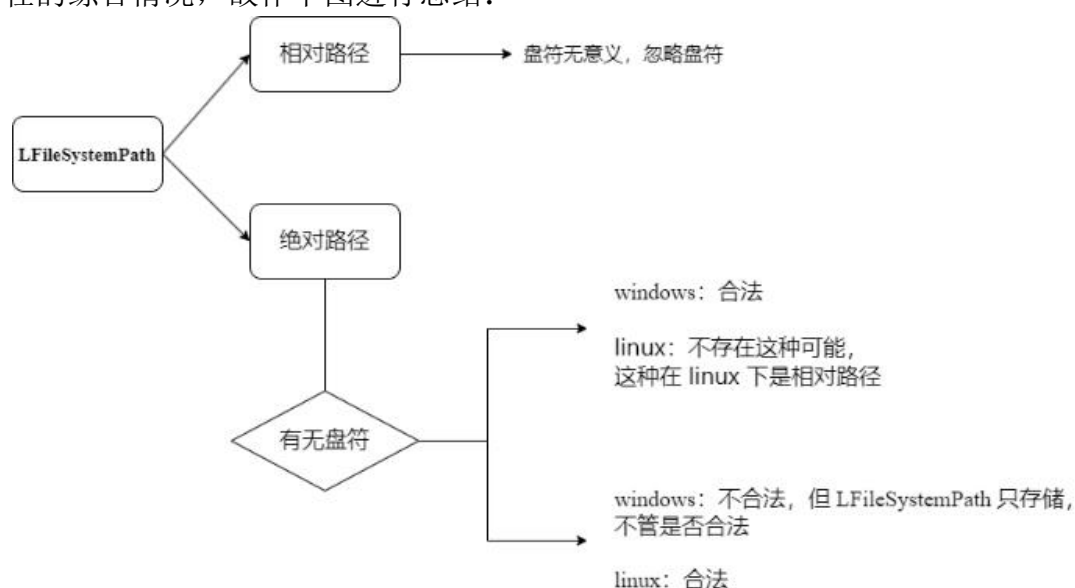


图 2-12 考虑盘符以后的综合情况

因此，经过如上考虑，最新的算法流程是一个 `path` 进来以后，先考虑盘符，如果是 windows 尝试提取盘符，如果是 linux 不管；后面再进行反斜杠\转化为正斜杠/，然后 `split`，再存储的过程，当然其中会有更多需要注意的小细节。

综上所述，我们最终的 `LFileSystemEntry` 和 `LFilePath` 的语义就完全区分开了。`LFilePath` 只是一个跨平台的路径存储结构，负责将传入的可能不太规范的路径字符串进行内部处理，导出为不同平台下的合法格式的绝对或

者相对路径的字符串,其内部存储一个 LStringList 的路径结构和 LString 的盘符。LFileSystemEntry 在 LFileSystemPath 基础上真正和文件系统打交道,在这里,将文件和目录视为一个东西(目录本来就是特殊的文件),从而提供各种各样的功能。

下面展示部分代码:

```
1. // lfilesystempath.cpp
2. void LFileSystemPath::setPath(const LString &path)
3. {
4.     // 传入路径为空, 不做处理, 直接返回
5.     if (path.isEmpty())
6.     {
7.         // TODO 这里传入为空是否需要打印一个 warning 信息
8.         clear();
9.         return;
10.    }
11.
12.    // 首先去掉首尾的空格
13.    LString copyPath = path.trimmed();
14.
15.    #if _WIN32
16.        // windows 下 / 和 \ 都可以作为分隔符, 但是 linux 下 \ 可以作为文件名, 因此
        // 在 windows 下将 path 中的所有反斜杠 \ 替换为正斜杠 /
17.        copyPath.replace(LChar("\\"), LChar("/"));
18.
19.    #endif
20.
21.    // 将多个 / 进行合并, 变成一个 /
22.    for (int i = 0; i < copyPath.size(); ++i)
23.    {
24.        if (i != copyPath.size() - 1 && LChar('/') == copyPath.at(i)
            && LChar('/') == copyPath.at(1 + i))
25.        {
26.            copyPath.erase(1 + i);
27.            --i;
28.        }
29.    }
30.
31.    #if _WIN32
32.        // 在 windows 下, 先尝试匹配盘符, 然后将盘符部分从 copyPath 中剔除
33.        int pos = copyPath.find(LChar(':'));
34.        if (-1 != pos)
35.        {
36.            m_drive = copyPath.substr(0, pos);
```

```
37.     copyPath.erase(0, 1 + pos);
38. }
39.
40. #endif
41.
42. // 由于我们进行了统一, 因此对 / 进行 split
43. // 第二个参数表示是否保留分割出来的空字符串, 因为我们把中间连续的 / 合并
   了, 因此只用管首尾即可
44.     m_pathList = copyPath.split(LChar('/'), false);
45.
46. // 没有以 .. 和 . 命名的文件, 用户可能输入 ./a/b/../../, 这个最后的 .. 是
   没有 / 的, 因此这里需要规范化一下
47.     if (LString(".") == m_pathList.last() || LString("..") == m_p
        athList.last())
48.     {
49.         m_pathList.append(LString());
50.     }
51.
52. #if _WIN32
53. // 如果是相对路径, 那么盘符就没有意义了, 直接将其置空即可
54.     if (!m_pathList.first().isEmpty())
55.     {
56.         m_drive.clear();
57.     }
58.
59. #endif
60. }
61.
62. // lfilesystementry.cpp
63. void LFileSystemEntry::setPath(const LFileSystemPath &fileSy
    stemPath)
64. {
65.     // 处理传入 filePath 为空的特殊情况给, 否则 windows 下, 下面会异
        常
66.     if (filePath.isEmpty())
67.     {
68.         // TODO 同 LFileSystemPath, 是否需要打印一个 warning 信息
69.         clear();
70.         return;
71.     }
72.
73.     m_fileSystemPath = filePath;
74.
75. // windows 下, 不含有盘符的绝对路径是不合法的, 这里抛出异常
```

```
76. #if_WIN32
77.   if (m_fileSystemPath.m_pathList.first().isEmpty() && m_file
       SystemPath.m_drive.isEmpty())
78.   {
79.       throw LException("路径不合法, 绝对路径未带盘符!");
80.   }
81.
82. #endif
83.
84.   // 处理目录或者文件名称是否包含不合法的字符
85.   // Linux 下不能使用带 / 的字符串作为名字, 但是带有 / 被认为是分隔符, 因此不
       能出现在名字中, 同时不能使用 . 和 .. 作为名字, 这也被 LFileSystemPath 处理
       了, 因此在这里是 ok 的
86.   // windows 下不能使用 \ / : * ? " < > | , 其中 \ 和 / 已处理
87. #if_WIN32
88.   for (auto &e : fileSystemPath.m_pathList)
89.   {
90.       if (e.contains(LChar(':')) || e.contains(LChar('*')) || e.
           contains(LChar('?')) || e.contains(LChar('"')) || e.contains(L
           Char('<')) || e.contains(LChar('>')) || e.contains(LChar('|'))
           )
91.       {
92.           throw LException("路径含有非法字符!");
93.       }
94.   }
95. #endif
96.
97.   // windows 和 linux 下不能出现同名文件和目录, 这其实也好理解, 因为文件和
       目录本质上是一样的
98.   // 这里可能会出现这种情况, 就是用户想要的是一个目录, 但是他忘记写最后
       的 / 了, 这很常见, 就需要帮助补齐
99.   // 同时这里也是为了确定到底该路径指向的路径是文件还是目录
100.   // 需要明确一下 exists() 的返回结果, 见 snippet/ExistsTest
101.   // 首先, 用户如果传入的是目录, 那么只能去匹配目录, 因此找不到就代表不存在
102.   // 其次, 用户如果传入的是文件, 那么能匹配到文件或者目录, 需要进一步判断修
       正
103.   // 因此, 只要检测用户传入的路径不存在, 那么就一定是不存在的路径
104.   LString path = absolutePath();
105.   if (!Lark::FileSystem::exists(path))
106.   {
107.       LLog::warn() << "当前文件或目录不存在!";
108.       clear();
109.   }
110.   else
```



```
111. {
112.     // 如果用户传入的是文件类型的路径, 那么仍有可能是目录, 需要检测是否为目
    录做进一步的判断
113.     if (!m_fileSystemPath.m_pathList.last().isEmpty() && Lar
        k::FileSystem::exists(path.append(LChar('/'))))
114.     {
115.         m_fileSystemPath.m_pathList.append(LString());
116.     }
117. }
118. }
```

从代码里, 可以看到, 这两个结构对于跨平台的要求也是非常严格的, 因此不只一次在代码里面用到了`#ifdef`这样的预处理宏用户处理平台差异, 当然经过更细节的设计, 保证了这样的预处理宏只在构造函数中使用, 保证了代码的可读性和可维护性。

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、新知识点的学习出发，阐述实际学习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

在初期的工作当中，代码的研发工作都是在 Linux 系统上完成的，在我的电脑上就是通过 windows 宿主系统加上 vmware 挂载 Ubuntu 系统的虚拟机完成的。但是在中期的工作中，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 MSVC16，对应 VS 的版本是 VS2019。同 Linux，对应的 c++ 语言版本严格规定为 c++11，构建依赖的 CMake 版本不低于 3.12。

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。这部分最核心的工作我目前尚未涉及到，但是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

在中期的工作中，由于需要在 windows 上开发代码，因此使用 VS Studio 作为代码编辑器，VS 自带的 MSVC 作为 c++ 编译器。我在初期报告中提到过，VS 工具链虽然强大，但是功能较为复杂，甚至有些繁琐，因此目前组内采用的是 MSVC 加上 win10 的 SDK 作为编译的工具链，然后在 vscode 中使用 CMake 工具辅助构建项目的方式，这样能有效的提高工作效率和简洁程度。

3.2 新知识点的学习

3.2.1 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。

3.2.2 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。

3.2.3 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。[11]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。[12]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 中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 中期任务完成度

根据前面的任务，可以将中期任务分为四个部分，代码走查、代码优化重构、课题调研以及协助 LarkTestKit，具体见表 1-1。

第一，代码走查。剩余三个扩展模块的走查工作已完成 Xml 模块的走查，并做好文档的记录，见图 4-1。剩余两个模块暂未开始，但是 Network 模块是我在进入公司之前已经修改过一版，公司相对来讲比较认可，走查优先级不是特别高。因此该部分的完成度可以认为是 50%。

序号	模块名称	任务名称	任务描述	负责人	执行人	计划完成时间	实际完成时间	备注
3	LarkSDK-XML	流式解析器	Parser	张天	刘治学	2024Q2		1. 解析器类 Parser 设计一言难尽，代码中多次进行一个字节节的匹配，并且整体代码量冗余 2. Parser 和代码存在各种规范问题，例如换行不规范、注释不全、注释不规范、函数可以内联等等
1		流式读取器	LXmlReader	张天	刘治学	2024Q2		1. Reader 的接口设计混乱，注释不全、不规范，可读性几乎没有 2. 与 Parser 一样，全是硬解析、硬匹配，循环套循环，函数套函数是非常常见的事情，处理具有一定规模的数据性能和正确性堪忧
2		流式写入器	LXmlWriter	张天	刘治学	2024Q2		1. 代码存在注释不全、不规范、换行不规范、函数可以内联等问题，可读性不高 2. 代码中的某些函数具有很多个重载版本，绝大多数没有做到很好的代码复用，而是各自实现，导致代码量冗余，并且极难维护
3	LarkSDK-DB							
4	LarkSDK-Network							

图 4-1 代码走查工作记录表

第二，代码优化重构。经过中期组内的共同努力，已经基本完成 util 部分的优化重构。我个人也负责完成了字符串列表（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）等工具类的处理，具体见图 1-2。该部分的完成度为 100%。

第三，课题调研。针对产品迭代中暴露的需求，开展了相关的调研工作。目前已完成标准库 std::string 的 sso 优化对 LVector 插入的影响的研究，LDir 和 LFileInfo 的语义明确和设计两个课题，对应的思路也在工程代码中修改完毕。目前正准备开展后续的 QGraphicScene 的调研工作，具体见图 1-3。该部分的完成度为 67%。

第四，协助 LarkTestKit。LarkTestKit 目前已经完成初步验收，结果稳定可用。并同时我协助学长一起完成了 LarkTest 使用说明文档的编写，具体如图 4-2。该部分的完成度为 100%。

LarkTest使用说明文档

1.简介

LarkTest是Lark5的单元测试框架，从使用层面来说它大体对标GoogleTest，而在部分功能以及细节上有所不同。本文档将介绍LarkTest的基本使用方式。

2.开始使用LarkTestKit

在开始使用LarkTest之前，首先需要正确安装并配置LarkTestKit，目前它通过LarkStudio安装并以静态库的方式呈现。

安装完毕后，就可以开始编写测试用例文件。在测试用例文件当中编写测试代码之前，需要完成以下前置工作：

1.定义宏选项

LarkTest中的某些部分使用了以宏定义为选项开关的条件编译，在使用这些部分的功能之前，需要定义相应的宏选项。

(1)开启UITEST，即LarkTest中的UI模拟测试：

```
#define LUITEST_OPEN
```

(2)开启UI模拟测试中在事件循环启动后仍能跨线程投递UI事件的测试用例：

```
#define UIEVENT_POST_OPEN
```

(3)每个UI测试用例默认开启主控制窗体ControlWindow：

```
#define DEFAULT_CONTROL_WINDOW
```

(4)默认状态下LarkTest的测试用例文件允许“断言宏简化”，即LarkTest断言宏中的“LARK_”前缀可以省略(同时即使不省略也合法)，与GoogleTest中的断言在形式上一致。如果不想简化断言宏，可以定义该选项：

图 4-2 LarkTest 使用说明文档

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 4-1：

表 4-1 后期任务计划表

序号	工作内容	工作开始时间	工作结束时间
1	继续维护单元测试	2024-06-01	持续进行
2	走查 DB 和 Network 模块代码	2024-06-01	大概一个月
3	进行 QGraphicsScene 的课题调研	2024-06-01	大概一个月

5 参考文献

- [1] Woosley, R., J. Molnar, and J. Somberg. "On the QT." (2021).
- [2] Srinivasan, S., et al. "Efficient interaction between OS and architecture in heterogeneous platforms." *ACM SIGOPS Oper. Syst. Rev.* (2011).
- [3] Kärkkäinen, Juha, Marcin Piatkowski, and S. Puglisi. "String Inference from Longest-Common-Prefix Array." *Theor. Comput. Sci.* (2022).
- [4] Hegde, Shrinidhi G, and G. Ranjani. "Package Management System in Linux." 2021 Asian Conference on Innovation in Technology (ASIANCON) (2021).
- [5] Abdulla, P., et al. "Efficient handling of string-number conversion." *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2020).
- [6] Brede, N., and N. Botta. "On the correctness of monadic backward induction." *Journal of Functional Programming* (2020).
- [7] 贺志朋. 浅析QT入门之信号与槽机制[J]. 山东工业技术, 2016, No. 228(22): 142. DOI: 10.16640/j.cnki.37-1222/t.2016.22.122.
- [8] Ortega, J.. "Reference Management Tools." *A-Z Common Reference Questions for Academic Librarians* (2019).
- [9] Nuriev, Marat Gumerovich, E. Belashova, and Konstantin Alekseevich Barabash. "Markdown File Converter to LaTeX Document." (2023).
- [10] Anandan, Ranjith Kumar, et al. "Improving Discoverability and Indexing of Interplanetary File system using Activitypub." 2022 International Conference on Advances in Computing, Communication and Applied Informatics (ACCAI) (2022).
- [11] Peng, Zhen-yuan, and Bo Zhou. "Minimum status of trees with given parameters." *RAIRO Oper. Res.* (2020).
- [12] Olukotun, K., et al. "Efficient Multiway Hash Join on Reconfigurable Hardware." *ArXiv* (2019).