1. 整体的设计

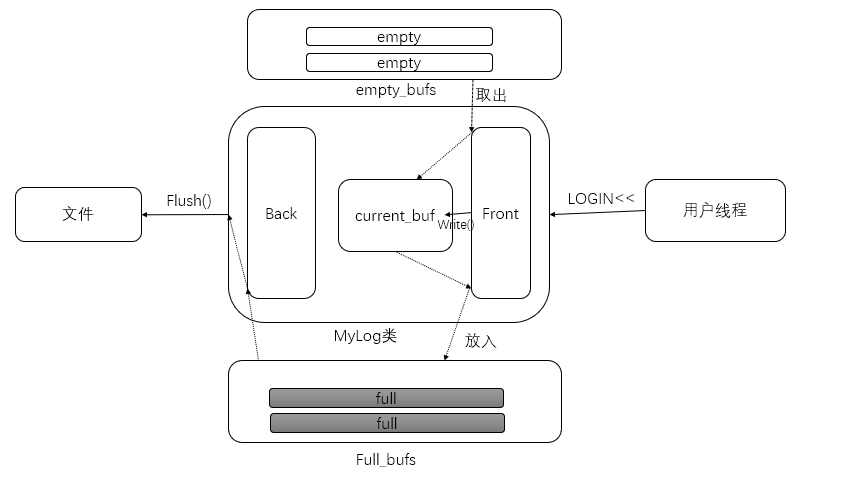
整个日志库实现了一个异步的日志功能，分为三个文件：

MyLog是整个日志库的内核，管理着所有的功能逻辑。

LogBuffer是日志库的缓存类，用于提供单个缓存空间，以及缓存的写入和把缓存内容写出到磁盘等功能。

LogFront用于实现写日志绑定当前日志所在的文件名，行号等功能。(**因为不同的位置，不同的线程调用LOGIN写日志时，行号什么的都会不同，如果硬要和MyLog整个日志库捆绑到一起的话，就不可避免的需要使用互斥锁，在一个线程写日志的时候，需要改变行号等临时信息，其他线程需要阻塞。这样效率就很低了。所以考虑现在这种实现方法，每次写日志的位置都创建一个临时的匿名LogFront类对象，对象中保存的当然就是行号等数据，因为保存的数据不多，所以创建和析构对象的开销不会很大，同时匿名对象在当前一行代码执行完之后会立刻析构，不会一直保留。**)

设计图：



LogFront

首先介绍MyLog 类的技术要点：

1.MyLog 类对象的创建：

a．日志对象是需要伴随整个程序一直存在的，

b．日志对象全局只能有一个，因此需要Singleton模式

c．日志对象创建会存在不同进程的竞争，所以需要有一个全局互斥变量比它先存在，在创建对象时需要加锁。



Singoleton的实现：

对象的创建是通过Get\_instance（）函数实现的，因为**调用函数的时候还不存在对象，所以函数要加上static关键字**（其他的static成员变量或者函数也是同理）。然后函数首先检查全局的日志对象MyLog::my\_log\_obj\_ 是否被创建，若没有则执行下面的创建步骤，若创建了，就直接返回该对象（以后任何进程都可以直接调用该函数直接获取对象不用再创建）。

创建步骤：首先因为会有不同的线程同时在没有创建对象的时候调用，所以先加锁，之后再加一个判断是否对象已经创建（**双检测机制**），因为只有一个线程会获得锁执行new函数，其他线程被唤醒后，对象已经创建完毕，他们就不用再创建一遍了，直接拿走对象就可以了。

2.static成员变量的初始化

Static成员的初始化只能放在编译期，在生成.o文件后，各个文件的数据段就已经产生了，其中也包含static成员的初始化，没初始化就使用会报错的。

static的声明在.h文件中，初始化在对应的.cpp文件中。（**初始化不能放在.h文件里，否则会报错。**）

这里有一点可能会出现问题，就是全局变量的初始化顺序，在MyLog.cpp的最上面我初始化了static 互斥量和对象，并且把互斥量的初始化放在前面。 但是linux老师说过**全局变量初始化顺序是不确定的，能不用就不用**。所以在这里可能会出问题，当然，在我的系统正常运行。（解决方法：在LOGIN宏替换的后面，加一个大括号，最前面添加一个Get\_instance（）函数的调用，不过这样每次写一条日志就要调用一次，还是有浪费的，所以我没有改）

3.日志使用了异步日志

整个日志库分为前端和后端。使用了双缓冲机制，前端往缓冲A写的时候，后端可以把缓冲B的内容往磁盘写，这样双方都不需要等待。

前端被各个线程调用，负责收集各个日志内容，组装成格式化的每条日志，然后写入当前缓存。

后端是一个单独的“背景线程”，专门用来把满缓存队列里面的日志写到文件里，再把写完的缓存加入空缓存队列中。后端线程不会一直阻塞，而是调用pthread\_cond\_timedwait等待一段时间就醒来一次，检查是否有新的日志已经写入当前缓存。没有写满整个缓存空间也会被刷新，为了保证缓存的一定的实时性。

缓存分为当前“当前使用的缓存”，空缓存队列，满缓存队列。当前缓存写满后放入满缓存队列，然后通知后端满缓存已经有任务可以写，然后再从空缓存队列取出一块缓存作为新的当前缓存。

为了保证缓存不丢失，还设置了atexit功能，在整个进程退出时先等后端执行完最后一遍，再刷新一遍所有的缓存，尽可能保证之前的缓存没有遗失。这之后还有一小段时间如果再写日志，那就没办法了。（可以通过加一个flag参数，写之前判断是否已经关闭后端线程，如果关闭了，就直接调用write写到文件里，而不是front。 不过感觉这样有点大题小做了，这么短的时间会不会写还是问题，写了也不能写多少，整个进程都该结束了，还多了那么多判断，浪费效率。）

（**之前考虑过，多线程同时写日志需要加锁从而被阻塞，是否会影响性能，是不是需要再加一个阻塞队列，从而不会被阻塞的问题。考虑了一下这样好像对性能也没有太大帮助，首先，每条日志的信息不会很多，而且是直接写在内存创建的缓存中，写起来是很快的，并不会阻塞太久。如果是在用户程序和前端之间再加一个阻塞队列的话，需要先把日志内容复制到阻塞队列中，前端再把阻塞队列的内容复制到缓存中，这样就多复制了一次，写阻塞队列的话也需要加锁，没有什么改变还多复制了一次。所以直接就在前端这里写加锁，写和释放缓存也是整个日志库唯一需要加锁的地方。**）

4.缓存单独用了一个类实现，在日志对象构造的时候，就实现分配好缓存，不够的时候可以请求增加，分配好的缓存利用BOOST库的“指针容器”来智能管理动态分配生命期，而不是利用析构函数释放，这里不使用vector+shared\_ptr是因为BOOST专门的指针容器要效率一些（移动构造函数了解一下），专门实现了指针数组ptr\_vector这种东西。也是尝试一下新技术。

通过这，也学会了怎么去看BOOST的文档，（英语阅读能力真的挺重要）。唯一需要注意的就是，使用了“指针容器”之后，指针容器里的内容是容器独有的，相当于unique\_ptr，取出需要容器pop，容器也也不能被复制给其他容器，所以要使用指针容器，需要把复制构造函数和对象复制表达式设置成私有的，这样用户就不能用=或者对象构造，因为私有对象用户调用不了。不设置的话，就会报错，为了这个错误我把代码每个函数都逐一做了一遍单元测试，结果都正常，合起来还是报错，就是因为这个原因。

Boost的使用和操作具体需要看文档，有相关的书也可以大概浏览一遍，直到boost有哪些功能，类似的，之后stl也需要这样过一遍

5.后端线程里面的无限循环条件设置成runing\_变量，初值为真，可以通过控制runing\_变成假来达成控制后端线程停止，然后再pthread\_join。这是一种很好的自由控制子线程停止的功能，比直接设置成1好的多。

6.在构造函数中通过readlink获取当前执行的进程名字，也是main函数的第一个参数作为成员变量保存起来，之后创建文件的时候需要进程名的。

7.文件滚动功能，即每个日志文件固定的大小，不可能刚好这么大，所以我设置的是写完超出文件大小的最后一条日志。

**影响文件滚动的还有文件名，时间需要设置成微秒级。否则，如果文件大小设置的非常小，很短的时间内就写满，文件滚动调用open创建新文件，这时候因为时间短，所以文件名组织起来发现和上一个文件没有变化，就直接打开老文件而不是文件滚动了。**

文件大小设置的很小，后端线程一次写的过程可能需要创建多次文件，比如当前写不下了，创建一个新文件，不能把剩下的内容全部写到新文件里面，需要判断是否能装下，不能的话需要只写入部分，再创建新的文件（需要一个循环）。

8.使用宏来实现LOGIN<<打印功能，和当前真实文件名，行号的打印功能。宏可以完成很多语言不能完成的事情，可以好好思考一下。

9.在一个类中创建线程的时候，把this指针当做线程函数的参数传进去，可以使子线程和主线程共享该类的资源，尤其是需要互斥量，条件变量这些东西的时候。

再来介绍一下LogBuffer缓存类的一些技术细节：

其实缓存类也没什么特别的，就是分配一块缓存，包括往缓存中写数据，缓存剩余大小，把缓存的数据写到文件等功能。

1.缓存的write（）函数用来写缓存，但是越界溢出不要它来控制，而是配合LeftSize（）函数由日志库来控制。

2. Flush（）函数用来写文件，需要传入文件剩余的空间参数来判断是写入所有的缓存，还是只写部分的缓存。

3.缓存的所有内容都刷新完以后，需要重置缓存指针，由日志库调用clear（）函数实现。

LogFront类：

这个类就是为了绑定每条日志可能不同的行号和文件名而设计的，因为只有这两个成员变量，所以整个类的构造代价不大。剩下的就是使用流的方式重写<<操作符了。