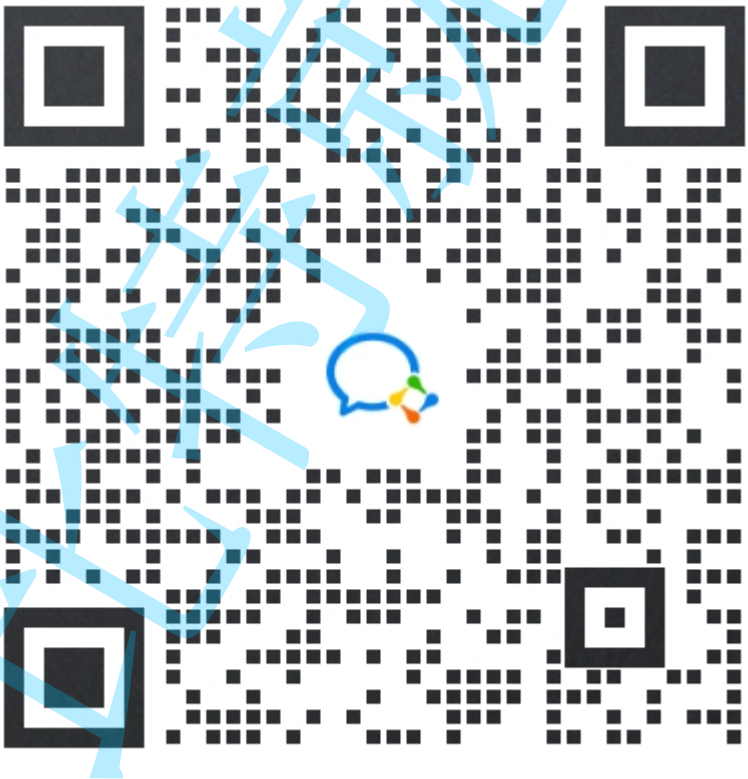
C++ - 基于多设计模式下的同步&异步日志系统

版权说明

对比特项目感兴趣，可以联系这个微信。



代码链接: https://gitee.com/qigezi/bitlog.git

1. 项目介绍

本项目主要实现一个日志系统， 其主要支持以下功能:

- 支持多级别日志消息
- 支持同步日志和异步日志
- 支持可靠写入日志到控制台、文件以及滚动文件中
- 支持多线程程序并发写日志
- 支持扩展不同的日志落地目标地

2. 开发环境

- CentOS 7
- vscode/vim
- g++/gdb
- Makefile

3. 核心技术

- 类层次设计(继承和多态的应用)
- C++11(多线程、auto、智能指针、右值引用等)
- 双缓冲区
- 生产消费模型
- 多线程
- 设计模式(单例、工厂、代理、建造者等)

4. 环境搭建

本项目不依赖其他任何第三方库， 只需要安装好CentOS/Ubuntu + vscode/vim环境即可开发。

5. 日志系统介绍

5.1 为什么需要日志系统

- 生产环境的产品为了保证其稳定性及安全性是不允许开发人员附加调试器去排查问题， 可以借助日志系统来打印一些日志帮助开发人员解决问题
- 上线客户端的产品出现bug无法复现并解决， 可以借助日志系统打印日志并上传到服务端帮助开发人员进行分析
- 对于一些高频操作（如定时器、心跳包）在少量调试次数下可能无法触发我们想要的行为，通过断点的暂停方式，我们不得不重复操作几十次、上百次甚至更多，导致排查问题效率是非常低下， 可以借助打印日志的方式查问题

- 在分布式、多线程/多进程代码中， 出现bug比较难以定位， 可以借助日志系统打印log帮助定位bug
- 帮助首次接触项目代码的新开发人员理解代码的运行流程

5.2 日志系统技术实现

日志系统的技术实现主要包括三种类型:

- 利用printf、std::cout等输出函数将日志信息打印到控制台
- 对于大型商业化项目， 为了方便排查问题，我们一般会将日志输出到文件或者是数据库系统方便查询和分析日志， 主要分为同步日志和异步日志方式
  - 同步写日志
  - 异步写日志

5.2.1 同步写日志

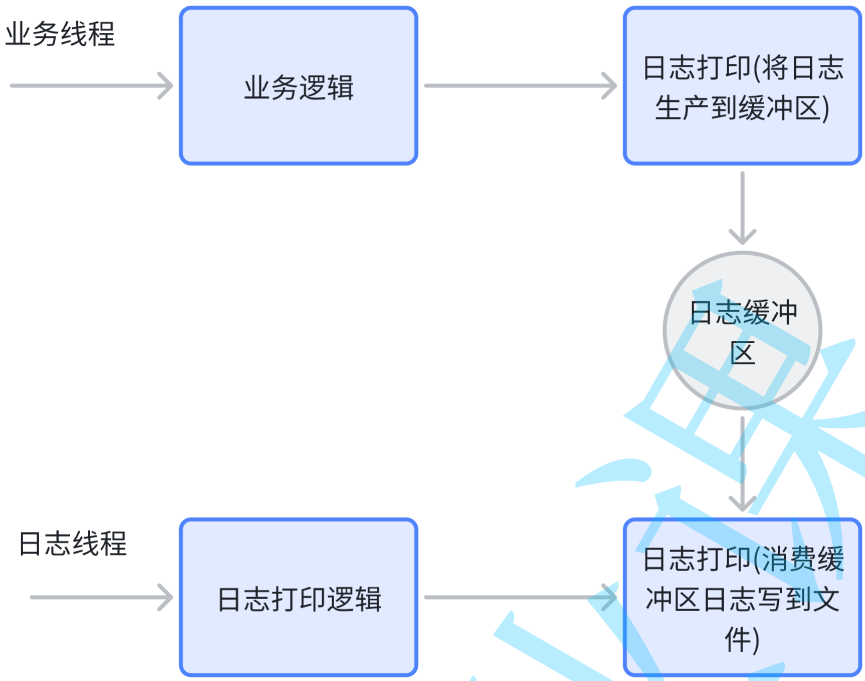
同步日志是指当输出日志时，必须等待日志输出语句执行完毕后，才能执行后面的业务逻辑语句，日志输出语句与程序的业务逻辑语句将在同一个线程运行。每次调用一次打印日志API就对应一次系统调用write写日志文件。

在高并发场景下，随着日志数量不断增加，同步日志系统容易产生系统瓶颈：

- 一方面，大量的日志打印陷入等量的write系统调用，有一定系统开销.
- 另一方面，使得打印日志的进程附带了大量同步的磁盘IO，影响程序性能

5.2.2 异步写日志

异步日志是指在进行日志输出时，日志输出语句与业务逻辑语句并不是在同一个线程中运行，而是有专门的线程用于进行日志输出操作。业务线程只需要将日志放到一个内存缓冲区中不用等待即可继续执行后续业务逻辑（作为日志的生产者），而日志的落地操作交给单独的日志线程去完成（作为日志的消费者）, 这是一个典型的生产-消费模型。



这样做的好处是即使日志没有真的地完成输出也不会影响程序的主业务，可以提高程序的性能：

- 主线程调用日志打印接口成为非阻塞操作
- 同步的磁盘IO从主线程中剥离出来交给单独的线程完成

6. 相关技术知识补充

6.1 不定参函数

在初学C语言的时候，我们都用过printf函数进行打印。其中printf函数就是一个不定参函数，在函数内部可以根据格式化字符串中格式化字符分别获取不同的参数进行数据的格式化。

而这种不定参函数在实际的使用中也非常多见，在这里简单做一介绍：

不定参宏函数

```
#include <iostream>
#include <cstdarg>

#define LOG(fmt, ...) printf("[%s:%d] " fmt "\n", __FILE__, __LINE__,
##__VA_ARGS__)
int main()
{
    LOG("%s-%s", "hello", "比特就业课");
    return 0;
}
```

C风格不定参函数

```cpp
#include <iostream>
#include <cstdarg>
void printNum(int n, ...) {
    va_list al;
    va_start(al, n);//让al指向n参数之后的第一个可变参数
    for (int i = 0; i < n; i++) {
        int num = va_arg(al, int);//从可变参数中取出一个整形参数
        std::cout << num << std::endl;
    }
    va_end(al);//清空可变参数列表--其实是将al置空
}

int main()
{
    printNum(3, 11,22,33);
    printNum(5, 44,55,66,77,88);
    return 0;
}
```

```cpp
#include <iostream>
#include <cstdarg>

void myprintf(const char *fmt, ...) {
    //int vasprintf(char **strp, const char *fmt, va_list ap);
    char *res;
    va_list al;
    va_start(al, fmt);
    int len = vasprintf(&res, fmt, al);
    va_end(al);
    std::cout << res << std::endl;
    free(res);
}

int main()
{
    myprintf("%s-%d", "小明", 18);
    return 0;
}
```

C++风格不定参函数

```cpp
#include <iostream>
#include <cstdarg>
#include <memory>
#include <functional>

void xprintf() {
    std::cout << std::endl;
}
template<typename T, typename ...Args>
void xprintf(const T &value, Args &&...args) {
    std::cout << value << " ";
    if ((sizeof ...(args)) > 0) {
        xprintf(std::forward<Args>(args)...);
    }else {
        xprintf();
    }
}

int main()
{
    xprintf("比特");
    xprintf("比特", 666);
    xprintf("比特", "就业课", 666);
    return 0;
}
```

6.2 设计模式

设计模式是前辈们对代码开发经验的总结，是解决特定问题的一系列套路。它不是语法规定，而是一套用来提高代码可复用性、可维护性、可读性、稳健性以及安全性的解决方案。

六大原则：

- 单一职责原则（Single Responsibility Principle）；
  - 类的职责应该单一，一个方法只做一件事。职责划分清晰了，每次改动到最小单位的方法或类。
  - 使用建议：两个完全不一样的功能不应该放一个类中，一个类中应该是一组相关性很高的函数、数据的封装
  - 用例：网络聊天：网络通信 & 聊天，应该分割成为网络通信类 & 聊天类
- 开闭原则（Open Closed Principle）；
  - 对扩展开放，对修改封闭
  - 使用建议：对软件实体的改动，最好用扩展而非修改的方式。

    - 用例：超时卖货：商品价格---不是修改商品的原来价格，而是新增促销价格。
- 里氏替换原则（Liskov Substitution Principle）；
    - 通俗点讲，就是只要父类能出现的地方，子类就可以出现，而且替换为子类也不会产生任何错误或异常。
    - 在继承类时，务必重写父类中所有的方法，尤其需要注意父类的protected方法，子类尽量不要暴露自己的public方法供外界调用。
    - 使用建议：子类必须完全实现父类的方法，孩子类可以有自己的个性。覆盖或实现父类的方法时，输入参数可以被放大，输出可以缩小
    - 用例：跑步运动员类-会跑步，子类长跑运动员-会跑步且擅长长跑，子类短跑运动员-会跑步且擅长短跑
- 依赖倒置原则（Dependence Inversion Principle）。
    - 高层模块不应该依赖低层模块，两者都应该依赖其抽象. 不可分割的原子逻辑就是低层模式，原子逻辑组装成的就是高层模块。
    - 模块间依赖通过抽象（接口）发生，具体类之间不直接依赖
    - 使用建议：每个类都尽量有抽象类，任何类都不应该从具体类派生。尽量不要重写基类的方法。结合里氏替换原则使用。
    - 用例：奔驰车司机类--只能开奔驰； 司机类 -- 给什么车，就开什么车； 开车的人：司机--依赖于抽象
- 迪米特法则（Law of Demeter），又叫“最少知道法则”；
    - 尽量减少对象之间的交互，从而减小类之间的耦合。一个对象应该对其他对象有最少的了解。对类的低耦合提出了明确的要求：
        - 只和直接的朋友交流，朋友之间也是有距离的。自己的就是自己的（如果一个方法放在本类中，既不增加类间关系，也对本类不产生负面影响，那就放置在本类中）。
    - 用例：老师让班长点名--老师给班长一个名单，班长完成点名勾选，返回结果，而不是班长点名，老师勾选
- 接口隔离原则（Interface Segregation Principle）；
    - 客户端不应该依赖它不需要的接口，类间的依赖关系应该建立在最小的接口上
    - 使用建议：接口设计尽量精简单一，但是不要对外暴露没有实际意义的接口。
    - 用例：修改密码，不应该提供修改用户信息接口，而就是单一的最小修改密码接口，更不要暴露数据库操作

从整体上来理解六大设计原则，可以简要的概括为一句话，用抽象构建框架，用实现扩展细节，具体到每一条设计原则，则对应一条注意事项：

- 单一职责原则告诉我们实现类要职责单一；

- 里氏替换原则告诉我们不要破坏继承体系；
- 依赖倒置原则告诉我们要面向接口编程；
- 接口隔离原则告诉我们在设计接口的时候要精简单一；
- 迪米特法则告诉我们要降低耦合；
- 开闭原则是总纲，告诉我们要对扩展开放，对修改关闭。

单例模式

一个类只能创建一个对象，即单例模式，该设计模式可以保证系统中该类只有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息，这种方式简化了在复杂环境下的配置管理。

单例模式有两种实现模式：饿汉模式和懒汉模式

- 饿汉模式: 程序启动时就会创建一个唯一的实例对象。 因为单例对象已经确定， 所以比较适用于多线程环境中， 多线程获取单例对象不需要加锁， 可以有效的避免资源竞争， 提高性能。

```
// 饿汉模式
template<typename T>
class Singleton {
private:
    static Singleton _eton;
private:
    Singleton(){}
    ~Singleton(){}
public:
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
    static T& getInstance()
    {
        return _eton;
    }
};
Singleton Singleton::_eton;
```

- 懒汉模式:第一次使用要使用单例对象的时候创建实例对象。如果单例对象构造特别耗时或者耗费济源(加载插件、加载网络资源等)， 可以选择懒汉模式， 在第一次使用的时候才创建对象。
  - 这里介绍的是《Effective C++》一书作者 Scott Meyers 提出的一种更加优雅简便的单例模式 Meyers' Singleton in C++。
  - C++11 Static local variables 特性以确保C++11起，静态变量将能够在满足 thread-safe 的前提下唯一地被构造和析构

```cpp
// 懒汉模式
template <typename T>
class Singleton {
private:
    Singleton(){}
    ~Singleton(){}
public:
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
    static T& getInstance()
    {
        static Singleton _eton;
        return _eton;
    }
};
```

工厂模式

工厂模式是一种创建型设计模式， 它提供了一种创建对象的最佳方式。在工厂模式中，我们创建对象时不会对上层暴露创建逻辑，而是通过使用一个共同结构来指向新创建的对象，以此实现创建-使用的分离。

工厂模式可以分为:

- 简单工厂模式: 简单工厂模式实现由一个工厂对象通过类型决定创建出来指定产品类的实例。假设有个工厂能生产出水果，当客户需要产品的时候明确告知工厂生产哪类水果，工厂需要接收用户提供的类别信息，当新增产品的时候，工厂内部去添加新产品的生产方式。

```cpp

//简单工厂模式：通过参数控制可以生产任何产品
//  优点：简单粗暴，直观易懂。使用一个工厂生产同一等级结构下的任意产品
//  缺点：
//      1. 所有东西生产在一起，产品太多会导致代码量庞大
//      2. 开闭原则遵循(开放拓展，关闭修改)的不是太好，要新增产品就必须修改工厂方法。
#include <iostream>
#include <string>
#include <memory>

class Fruit {
    public:
        Fruit(){}
        virtual void show() = 0;
};
class Apple : public Fruit {
    public:
```

```
        Apple() {}
        virtual void show() {
            std::cout << "我是一个苹果" << std::endl;
        }
};
class Banana : public Fruit {
    public:
        Banana() {}
        virtual void show() {
            std::cout << "我是一个香蕉" << std::endl;
        }
};
class FruitFactory {
    public:
        static std::shared_ptr<Fruit> create(const std::string &name) {
            if (name == "苹果") {
                return std::make_shared<Apple>();
            }else if(name == "香蕉") {
                return std::make_shared<Banana>();
            }
            return std::shared_ptr<Fruit>();
        }
};


int main()
{
    std::shared_ptr<Fruit> fruit = FruitFactory::create("苹果");
    fruit->show();
    fruit = FruitFactory::create("香蕉");
    fruit->show();
    return 0;
}
```

这个模式的结构和管理产品对象的方式十分简单， 但是它的扩展性非常差，当我们需要新增产品的时候，就需要去修改工厂类新增一个类型的产品创建逻辑，违背了开闭原则。

- 工厂方法模式: 在简单工厂模式下新增多个工厂，多个产品，每个产品对应一个工厂。假设现在有 A、B 两种产品，则开两个工厂，工厂 A 负责生产产品 A，工厂 B 负责生产产品 B，用户只知道产品的工厂名，而不知道具体的产品信息，工厂不需要再接收客户的产品类别，而只负责生产产品。

```
#include <iostream>
#include <string>
#include <memory>

```

```cpp

//工厂方法：定义一个创建对象的接口，但是由子类来决定创建哪种对象，使用多个工厂分别生产指定的固定产品
//  优点：
//      1. 减轻了工厂类的负担，将某类产品的生产交给指定的工厂来进行
//      2. 开闭原则遵循较好，添加新产品只需要新增产品的工厂即可，不需要修改原先的工厂类
//  缺点：对于某种可以形成一组产品族的情况处理较为复杂,需要创建大量的工厂类.
class Fruit {
    public:
        Fruit(){}
        virtual void show() = 0;
};
class Apple : public Fruit {
    public:
        Apple() {}
        virtual void show() {
            std::cout << "我是一个苹果" << std::endl;
        }
    private:
        std::string _color;
};
class Banana : public Fruit {
    public:
        Banana() {}
        virtual void show() {
            std::cout << "我是一个香蕉" << std::endl;
        }
};
class FruitFactory {
    public:
        virtual std::shared_ptr<Fruit> create() = 0;
};
class AppleFactory : public FruitFactory {
    public:
        virtual std::shared_ptr<Fruit> create() {
            return std::make_shared<Apple>();
        }
};
class BananaFactory : public FruitFactory {
    public:
        virtual std::shared_ptr<Fruit> create() {
            return std::make_shared<Banana>();
        }
};


int main()
```

```
{
    std::shared_ptr<FruitFactory> factory(new AppleFactory());
    fruit = factory->create();
    fruit->show();
    factory.reset(new BananaFactory());
    fruit = factory->create();
    fruit->show();
    return 0;
}
```

工厂方法模式每次增加一个产品时，都需要增加一个具体产品类和工厂类，这会使得系统中类的个数成倍增加，在一定程度上增加了系统的耦合度。

- 抽象工厂模式: 工厂方法模式通过引入工厂等级结构，解决了简单工厂模式中工厂类职责太重的问题，但由于工厂方法模式中的每个工厂只生产一类产品，可能会导致系统中存在大量的工厂类，势必会增加系统的开销。此时，我们可以考虑将一些相关的产品组成一个产品族（位于不同产品等级结构中功能相关联的产品组成的家族），由同一个工厂来统一生产，这就是抽象工厂模式的基本思想。

```
#include <iostream>
#include <string>
#include <memory>


//抽象工厂：围绕一个超级工厂创建其他工厂。每个生成的工厂按照工厂模式提供对象。
//  思想：将工厂抽象成两层，抽象工厂 & 具体工厂子类， 在工厂子类种生产不同类型的子产品
class Fruit {
    public:
        Fruit(){}
        virtual void show() = 0;
};
class Apple : public Fruit {
    public:
        Apple() {}
        virtual void show() {
            std::cout << "我是一个苹果" << std::endl;
        }
    private:
        std::string _color;
};
class Banana : public Fruit {
    public:
        Banana() {}
        virtual void show() {
            std::cout << "我是一个香蕉" << std::endl;
```

```
27          }
28  };
29
30  class Animal {
31      public:
32          virtual void voice() = 0;
33  };
34  class Lamp: public Animal {
35      public:
36          void voice() { std::cout << "咩咩咩\n"; }
37  };
38  class Dog: public Animal {
39      public:
40          void voice() { std::cout << "汪汪汪\n"; }
41  };
42
43  class Factory {
44      public:
45          virtual std::shared_ptr<Fruit> getFruit(const std::string &name) = 0;
46          virtual std::shared_ptr<Animal> getAnimal(const std::string &name) = 0;
47  };
48
49  class FruitFactory : public Factory {
50      public:
51          virtual std::shared_ptr<Animal> getAnimal(const std::string &name) {
52              return std::shared_ptr<Animal>();
53          }
54          virtual std::shared_ptr<Fruit> getFruit(const std::string &name) {
55              if (name == "苹果") {
56                  return std::make_shared<Apple>();
57              }else if(name == "香蕉") {
58                  return std::make_shared<Banana>();
59              }
60              return std::shared_ptr<Fruit>();
61          }
62  };
63
64  class AnimalFactory : public Factory {
65
66      public:
67          virtual std::shared_ptr<Fruit> getFruit(const std::string &name) {
68              return std::shared_ptr<Fruit>();
69          }
70          virtual std::shared_ptr<Animal> getAnimal(const std::string &name) {
71              if (name == "小羊") {
72                  return std::make_shared<Lamp>();
73              }else if(name == "小狗") {
```

```
74                 return std::make_shared<Dog>();
75             }
76             return std::shared_ptr<Animal>();
77         }
78 };
79
80 class FactoryProducer {
81     public:
82         static std::shared_ptr<Factory> getFactory(const std::string &name) {
83             if (name == "动物") {
84                 return std::make_shared<AnimalFactory>();
85             }else {
86                 return std::make_shared<FruitFactory>();
87             }
88         }
89 };
90
91 int main()
92 {
93     std::shared_ptr<Factory> fruit_factory = FactoryProducer::getFactory("水
    果");
94     std::shared_ptr<Fruit> fruit = fruit_factory->getFruit("苹果");
95     fruit->show();
96     fruit = fruit_factory->getFruit("香蕉");
97     fruit->show();
98
99     std::shared_ptr<Factory> animal_factory = FactoryProducer::getFactory("动
    物");
100     std::shared_ptr<Animal> animal = animal_factory->getAnimal("小羊");
101     animal->voice();
102     animal = animal_factory->getAnimal("小狗");
103     animal->voice();
104     return 0;
105 }
```

抽象工厂模式适用于生产多个工厂系列产品衍生的设计模式，增加新的产品等级结构复杂，需要对原有系统进行较大的修改，甚至需要修改抽象层代码，违背了“开闭原则”。

建造者模式：

建造者模式是一种创建型设计模式， 使用多个简单的对象一步一步构建成一个复杂的对象，能够将一个复杂的对象的构建与它的表示分离，提供一种创建对象的最佳方式。主要用于解决对象的构建过于复杂的问题。

建造者模式主要基于四个核心类实现：

- 抽象产品类：

- 具体产品类：一个具体的产品对象类
- 抽象Builder类：创建一个产品对象所需的各个部件的抽象接口
- 具体产品的Builder类：实现抽象接口，构建各个部件
- 指挥者Director类：统一组建过程，提供给调用者使用，通过指挥者来构造产品

```cpp
#include <iostream>
#include <memory>

/*抽象电脑类*/
class Computer {
    public:
        using ptr = std::shared_ptr<Computer>;
        Computer() {}
        void setBoard(const std::string &board) {_board = board;}
        void setDisplay(const std::string &display) {_display = display;}
        virtual void setOs() = 0;
        std::string toString() {
            std::string computer = "Computer:{\n";
            computer += "\tboard=" + _board + ",\n";
            computer += "\tdisplay=" + _display + ",\n";
            computer += "\tOs=" + _os + ",\n";
            computer += "}\n";
            return computer;
        }
    protected:
        std::string _board;
        std::string _display;
        std::string _os;
};

/*具体产品类*/
class MacBook : public Computer {
    public:
        using ptr = std::shared_ptr<MacBook>;
        MacBook() {}
        virtual void setOs() {
            _os = "Max Os X12";
        }
};

/*抽象建造者类：包含创建一个产品对象的各个部件的抽象接口*/
class Builder {
    public:
        using ptr = std::shared_ptr<Builder>;
```

```
40          virtual void buildBoard(const std::string &board) = 0;
41          virtual void buildDisplay(const std::string &display) = 0;
42          virtual void buildOs() = 0;
43          virtual Computer::ptr build() = 0;
44  };
45
46  /*具体产品的具体建造者类：实现抽象接口，构建和组装各个部件*/
47  class MacBookBuilder : public Builder {
48      public:
49          using ptr = std::shared_ptr<MacBookBuilder>;
50          MacBookBuilder(): _computer(new MacBook()) {}
51          virtual void buildBoard(const std::string &board) {
52              _computer->setBoard(board);
53          }
54          virtual void buildDisplay(const std::string &display) {
55              _computer->setDisplay(display);
56          }
57          virtual void buildOs() {
58              _computer->setOs();
59          }
60          virtual Computer::ptr build() {
61              return _computer;
62          }
63      private:
64          Computer::ptr _computer;
65  };
66
67  /*指挥者类，提供给调用者使用，通过指挥者来构造复杂产品*/
68  class Director {
69      public:
70          Director(Builder* builder):_builder(builder){}
71          void construct(const std::string &board, const std::string &display) {
72              _builder->buildBoard(board);
73              _builder->buildDisplay(display);
74              _builder->buildOs();
75          }
76      private:
77          Builder::ptr _builder;
78  };
79
80  int main()
81  {
82      Builder *buidler = new MackBookBuilder();
83      std::unique_ptr<Director> pd(new Director(buidler));
84      pd->construct("英特尔主板", "VOC显示器");
85      Computer::ptr computer = buidler->build();
86      std::cout << computer->toString();
```

```
    return 0;
}
```

代理模式

代理模式指代理控制对其他对象的访问， 也就是代理对象控制对原对象的引用。在某些情况下，一个对象不适合或者不能直接被引用访问，而代理对象可以在客户端和目标对象之间起到中介的作用。

代理模式的结构包括一个是真正的你要访问的对象(目标类)、一个是代理对象。目标对象与代理对象实现同一个接口，先访问代理类再通过代理类访问目标对象。代理模式分为静态代理、动态代理：

- 静态代理指的是，在编译时就已经确定好了代理类和被代理类的关系。也就是说，在编译时就已经确定了代理类要代理的是哪个被代理类。
- 动态代理指的是，在运行时才动态生成代理类，并将其与被代理类绑定。这意味着，在运行时才能确定代理类要代理的是哪个被代理类。

以租房为例，房东将房子租出去，但是要租房子出去，需要发布招租启示， 带人看房，负责维修，这些工作中有些操作并非房东能完成，因此房东为了图省事，将房子委托给中介进行租赁。 代理模式实现：

```
/*房东要把一个房子通过中介租出去理解代理模式*/
#include <iostream>
#include <string>

class RentHouse {
    public:
        virtual void rentHouse() = 0;
};

/*房东类：将房子租出去*/
class Landlord : public RentHouse {
    public:
        void rentHouse() {
            std::cout << "将房子租出去\n";
        }
};
/*中介代理类：对租房子进行功能加强，实现租房以外的其他功能*/
class Intermediary : public RentHouse {
    public:
        void rentHouse() {
            std::cout << "发布招租启示\n";
            std::cout << "带人看房\n";
            _landlord.rentHouse();
            std::cout << "负责租后维修\n";
        }
```

```
    private:
        Landlord _landlord;
};

int main()
{
    Intermediary intermediary;
    intermediary.rentHouse();
    return 0;
}
```

7. 日志系统框架设计

本项目实现的是一个多日志器日志系统，主要实现的功能是让程序员能够轻松的将程序运行日志信息落地到指定的位置，且支持同步与异步两种方式的日志落地方式。

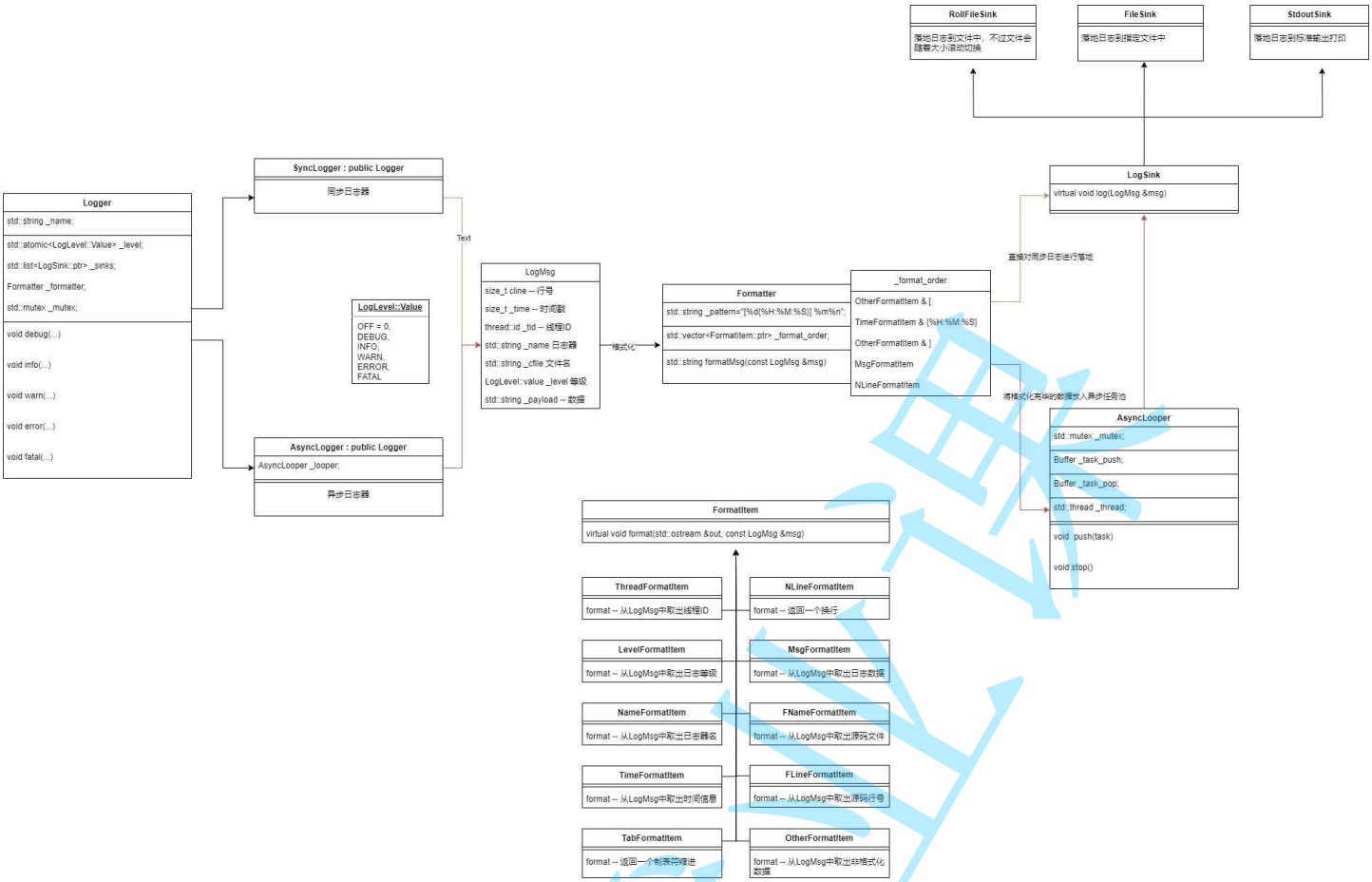
项目的框架设计将项目分为以下几个模块来实现。

7.1 模块划分

- 日志等级模块：对输出日志的等级进行划分，以便于控制日志的输出，并提供等级枚举转字符串功能。
  - OFF：关闭
  - DEBUG：调试，调试时的关键信息输出。
  - INFO：提示，普通的提示型日志信息。
  - WARN：警告，不影响运行，但是需要注意一下的日志。
  - ERROR：错误，程序运行出现错误的日志
  - FATAL：致命，一般是代码异常导致程序无法继续推进运行的日志
- 日志消息模块：中间存储日志输出所需的各项要素信息
  - 时间：描述本条日志的输出时间。
  - 线程ID：描述本条日志是哪个线程输出的。
  - 日志等级：描述本条日志的等级。
  - 日志数据：本条日志的有效载荷数据。
  - 日志文件名：描述本条日志在哪个源码文件中输出的。
  - 日志行号：描述本条日志在源码文件的哪一行输出的。
- 日志消息格式化模块：设置日志输出格式，并提供对日志消息进行格式化功能。
  - 系统的默认日志输出格式：%d{%H:%M:%S}%T[%t]%T[%p]%T[%c]%T%f:%l%T%m%n
  - -> 13:26:32 [2343223321] [FATAL] [root] main.c:76 套接字创建失败\n

    - %d{%H:%M:%S}：表示日期时间，花括号中的内容表示日期时间的格式。
    - %T：表示制表符缩进。
    - %t：表示线程ID
    - %p：表示日志级别
    - %c：表示日志器名称，不同的开发组可以创建自己的日志器进行日志输出，小组之间互不影响。
    - %f：表示日志输出时的源代码文件名。
    - %l：表示日志输出时的源代码行号。
    - %m：表示给与的日志有效载荷数据
    - %n：表示换行
    - 设计思想：设计不同的子类，不同的子类从日志消息中取出不同的数据进行处理。
- 日志消息落地模块：决定了日志的落地方向，可以是标准输出，也可以是日志文件，也可以滚动文件输出....
    - 标准输出：表示将日志进行标准输出的打印。
    - 日志文件输出：表示将日志写入指定的文件末尾。
    - 滚动文件输出：当前以文件大小进行控制，当一个日志文件大小达到指定大小，则切换下一个文件进行输出
    - 后期，也可以扩展远程日志输出，创建客户端，将日志消息发送给远程的日志分析服务器。
    - 设计思想：设计不同的子类，不同的子类控制不同的日志落地方向。
- 日志器模块：
    - 此模块是对以上几个模块的整合模块，用户通过日志器进行日志的输出，有效降低用户的使用难度。
    - 包含有：日志消息落地模块对象，日志消息格式化模块对象，日志输出等级
- 日志器管理模块：
    - 为了降低项目开发的日志耦合，不同的项目组可以有自己的日志器来控制输出格式以及落地方向，因此本项目是一个多日志器的日志系统。
    - 管理模块就是对创建的所有日志器进行统一管理。并提供一个默认日志器提供标准输出的日志输出。
- 异步线程模块：
    - 实现对日志的异步输出功能，用户只需要将输出日志任务放入任务池，异步线程负责日志的落地输出功能，以此提供更加高效的非阻塞日志输出。

7.2 模块关系图



8. 代码设计

8.1 实用类设计

提前完成一些零碎的功能接口，以便于项目中会用到。

- 获取系统时间
- 判断文件是否存在
- 获取文件的所在目录路径
- 创建目录

```
1
2  /*
3      通用功能类，与业务无关的功能实现
4          1. 获取系统时间
5          2. 获取文件大小
6          3. 创建目录
7          4. 获取文件所在目录
8  */
9  #ifndef __M_UTIL_H__
10 #define __M_UTIL_H__
11
12 #include <iostream>
```

```cpp
#include <fstream>
#include <sstream>
#include <string>
#include <ctime>
#include <cassert>
#include <sys/stat.h>

namespace bitlog{
    namespace util{
        class date {
            public:
                static size_t now() { return (size_t)time(nullptr); }
        };
        class file {
            public:
                static bool exists(const std::string &name) {
                    struct stat st;
                    return stat(name.c_str(), &st) == 0;
                }
                static std::string path(const std::string &name) {
                    if (name.empty()) return ".";
                    size_t pos = name.find_last_of("/\\");
                    if (pos == std::string::npos) return ".";
                    return name.substr(0, pos + 1);
                }
                static void create_directory(const std::string &path) {
                    if (path.empty()) return ;
                    if (exists(path)) return ;
                    size_t pos, idx = 0;
                    while(idx < path.size()) {
                        pos = path.find_first_of("/\\", idx);
                        if (pos == std::string::npos) {
                            mkdir(path.c_str(), 0755);
                            return;
                        }
                        if (pos == idx) {idx = pos + 1; continue;}
                        std::string subdir = path.substr(0, pos);
                        if (subdir == "." || subdir == "..")
                            {idx = pos + 1; continue;}
                        if (exists(subdir))
                            {idx = pos + 1; continue;}
                        mkdir(subdir.c_str(), 0755);
                        idx = pos + 1;
                    }
                }
        };
    }
```

```
60 }
61 #endif
```

8.2 日志等级类设计

日志等级总共分为7个等级，分别为：

- OFF 关闭所有日志输出
- DRBUG 进行debug时候打印日志的等级
- INFO 打印一些用户提示信息
- WARN 打印警告信息
- ERROR 打印错误信息
- FATAL 打印致命信息- 导致程序崩溃的信息

```
1  #ifndef __M_LEVEL_H__
2  #define __M_LEVEL_H__
3
4  namespace bitlog{
5      class LogLevel{
6          public:
7              enum class value {
8                  DEBUG,
9                  INFO,
10                 WARN,
11                 ERROR,
12                 FATAL,
13                 OFF
14             };
15             static const char *toString(LogLevel::value l) {
16                 switch(l) {
17                     #define TOSTRING(name) #name
18                     case LogLevel::value::DEBUG: return TOSTRING(DEBUG);
19                     case LogLevel::value::INFO: return TOSTRING(INFO);
20                     case LogLevel::value::WARN: return TOSTRING(WARN);
21                     case LogLevel::value::ERROR: return TOSTRING(ERROR);
22                     case LogLevel::value::FATAL: return TOSTRING(FATAL);
23                     case LogLevel::value::OFF: return TOSTRING(OFF);
24                     #undef TOSTRING
25                     default: return "UNKNOW";
26                 }
27                 return "UNKNOW";
28             }
29     };
```

```
}
#endif
```

8.3 日志消息类设计

日志消息类主要是封装一条完整的日志消息所需的内容，其中包括日志等级、对应的logger name、打印日志源文件的位置信息（包括文件名和行号）、线程ID、时间戳信息、具体的日志信息等内容。

```cpp
#ifndef __M_MSG_H__
#define __M_MSG_H__
#include "util.hpp"
#include "level.hpp"
#include <thread>

namespace bitlog{
    struct LogMsg {
        using ptr = std::shared_ptr<LogMsg>;
        size_t _line;//行号
        size_t _ctime;//时间
        std::thread::id _tid;//线程ID
        std::string _name;//日志器名称
        std::string _file;//文件名
        std::string _payload;//日志消息
        LogLevel::value _level;//日志等级
        LogMsg() {}
        LogMsg( std::string name,
                std::string file,
                size_t line,
                std::string payload,
                LogLevel::value level):
                _name(name),
                _file(file),
                _payload(payload),
                _level(level),
                _line(line),
                _ctime(util::date::now()),
                _tid(std::this_thread::get_id()) {}
    };
}

#endif
```

8.4 日志输出格式化类设计

日志格式化（Formatter）类主要负责格式化日志消息。其主要包含以下内容

- pattern成员：保存日志输出的格式字符串。
  - %d 日期
  - %T 缩进
  - %t 线程id
  - %p 日志级别
  - %c 日志器名称
  - %f 文件名
  - %l 行号
  - %m 日志消息
  - %n 换行
- std::vector<FormatItem::ptr> items成员：用于按序保存格式化字符串对应的子格式化对象。

FormatItem类主要负责日志消息子项的获取及格式化。其包含以下子类

- MsgFormatItem ： 表示要从LogMsg中取出有效日志数据
- LevelFormatItem ： 表示要从LogMsg中取出日志等级
- NameFormatItem ： 表示要从LogMsg中取出日志器名称
- ThreadFormatItem ： 表示要从LogMsg中取出线程ID
- TimeFormatItem ： 表示要从LogMsg中取出时间戳并按照指定格式进行格式化
- CFileFormatItem ： 表示要从LogMsg中取出源码所在文件名
- CLineFormatItem ： 表示要从LogMsg中取出源码所在行号
- TabFormatItem ： 表示一个制表符缩进
- NLineFormatItem ： 表示一个换行
- OtherFormatItem ： 表示非格式化的原始字符串

示例：`"[%d{%H:%M:%S}] %m%n"`

```
pattern = "[%d{%H:%M:%S}] %m%n"
items = {
    {OtherFormatItem(), "["},
    {TimeFormatItem(), "%H:%M:%S"},
    {OtherFormatItem(), "]"},
    {MsgFormatItem (), ""},
    {NLineFormatItem (), ""}
}

```

```
LogMsg msg = {
    size_t _line = 22;
    size_t _ctime = 12345678;
    std::thread::id _tid = 0x12345678;
    std::string _name = "logger";
    std::string _file = "main.cpp";
    std::string _payload = "创建套接字失败";
    LogLevel::value _level = ERROR;
};
```

格式化的过程其实就是按次序从Msg中取出需要的数据进行字符串的连接的过程。

最终组织出来的格式化消息： "[22:32:54] 创建套接字失败\n"

代码实现：

```
#ifndef __M_FMT_H__
#define __M_FMT_H__

#include "util.hpp"
#include "message.hpp"
#include "level.hpp"
#include <memory>
#include <vector>
#include <tuple>

namespace bitlog{

class FormatItem{
    public:
        using ptr = std::shared_ptr<FormatItem>;
        virtual ~FormatItem() {}
        virtual void format(std::ostream &os, const LogMsg &msg) = 0;
};
class MsgFormatItem : public FormatItem {
    public:
        MsgFormatItem(const std::string &str = ""){}
        virtual void format(std::ostream &os, const LogMsg &msg) {
            os << msg._payload;
        }
};
class LevelFormatItem : public FormatItem {
    public:
        LevelFormatItem(const std::string &str = ""){}
        virtual void format(std::ostream &os, const LogMsg &msg) {
            os << LogLevel::toString(msg._level);
```

```
        }
};
class NameFormatItem : public FormatItem {
    public:
        NameFormatItem(const std::string &str = ""){}
        virtual void format(std::ostream &os, const LogMsg &msg) {
            os << msg._name;
        }
};
class ThreadFormatItem : public FormatItem {
    public:
        ThreadFormatItem(const std::string &str = ""){}
        virtual void format(std::ostream &os, const LogMsg &msg) {
            os << msg._tid;
        }
};
class TimeFormatItem : public FormatItem {
    private:
        std::string _format;
    public:
        TimeFormatItem(const std::string &format = "%H:%M:%S"):_format(format){
            if (format.empty()) _format = "%H:%M:%S";
        }
        virtual void format(std::ostream &os, const LogMsg &msg) {
            time_t t = msg._ctime;
            struct tm lt;
            localtime_r(&t, &lt);
            char tmp[128];
            strftime(tmp, 127, _format.c_str(), &lt);
            os << tmp;
        }
};
class CFileFormatItem : public FormatItem {
    public:
        CFileFormatItem(const std::string &str = ""){}
        virtual void format(std::ostream &os, const LogMsg &msg) {
            os << msg._file;
        }
};
class CLineFormatItem : public FormatItem {
    public:
        CLineFormatItem(const std::string &str = ""){}
        virtual void format(std::ostream &os, const LogMsg &msg) {
            os << msg._line;
        }
};
class TabFormatItem : public FormatItem {
```

```
78      public:
79          TabFormatItem(const std::string &str = ""){}
80          virtual void format(std::ostream &os, const LogMsg &msg) {
81              os << "\t";
82          }
83  };
84  class NLineFormatItem : public FormatItem {
85      public:
86          NLineFormatItem(const std::string &str = ""){}
87          virtual void format(std::ostream &os, const LogMsg &msg) {
88              os << "\n";
89          }
90  };
91  class OtherFormatItem : public FormatItem {
92      private:
93          std::string _str;
94      public:
95          OtherFormatItem(const std::string &str = ""):_str(str){}
96          virtual void format(std::ostream &os, const LogMsg &msg) {
97              os << _str;
98          }
99  };
100 class Formatter {
101     public:
102         using ptr = std::shared_ptr<Formatter>;
103         /*
104             %d 日期
105             %T 缩进
106             %t 线程id
107             %p 日志级别
108             %c 日志器名称
109             %f 文件名
110             %l 行号
111             %m 日志消息
112             %n 换行
113         */
114         Formatter(const std::string &pattern = "[%d{%H:%M:%S}][%t][%p][%c]
    [%f:%l] %m%n"):
115             _pattern(pattern){
116             assert(parsePattern());
117         }
118         const std::string pattern() { return _pattern; }
119         std::string format(const LogMsg &msg) {
120             std::stringstream ss;
121             for (auto &it : _items) {
122                 it->format(ss, msg);
123             }
```

```
124             return ss.str();
125         }
126         std::ostream& format(std::ostream &os, const LogMsg &msg) {
127             for (auto &it : _items) {
128                 it->format(os, msg);
129             }
130             return os;
131         }
132         FormatItem::ptr createItem(const std::string &fc, const std::string
    &subfmt) {
133             if (fc == "m") return FormatItem::ptr(new MsgFormatItem(subfmt));
134             if (fc == "p") return FormatItem::ptr(new LevelFormatItem(subfmt));
135             if (fc == "c") return FormatItem::ptr(new NameFormatItem(subfmt));
136             if (fc == "t") return FormatItem::ptr(new
    ThreadFormatItem(subfmt));
137             if (fc == "n") return FormatItem::ptr(new NLineFormatItem(subfmt));
138             if (fc == "d") return FormatItem::ptr(new TimeFormatItem(subfmt));
139             if (fc == "f") return FormatItem::ptr(new CFileFormatItem(subfmt));
140             if (fc == "l") return FormatItem::ptr(new CLineFormatItem(subfmt));
141             if (fc == "T") return FormatItem::ptr(new TabFormatItem(subfmt));
142             return FormatItem::ptr();
143         }
144         //pattern解析
145         bool parsePattern() {
146             //std::string _pattern
    ="sg{}fsg%d{%H:%M:%S}%Tsdf%t%T[%p]%T[%c]%T%f:%l%T%m%n";
147             //std::cout << _pattern << std::endl;
148             //每个要素分为三部分:
149             //  格式化字符  : %d  %T  %p...
150             //  对应的输出子格式 :  {%H:%M:%S}
151             //  对应数据的类型 :  0-表示原始字符串，也就是非格式化字符，1-表示格式化数据
    类型
152             //  默认格式 "%d{%H:%M:%S}%T%t%T[%p]%T[%c]%T%f:%l%T%m%n"
153             std::vector<std::tuple<std::string, std::string, int>> arry;
154             std::string format_key;//存放%后的格式化字符
155             std::string format_val;//存放格式化字符后边 {} 中的子格式字符串
156             std::string string_row;//存放原始的非格式化字符
157             bool sub_format_error = false;
158             int pos = 0;
159             while (pos < _pattern.size()) {
160                 if (_pattern[pos] != '%') {
161                     string_row.append(1, _pattern[pos++]);
162                     continue;
163                 }
164                 if (pos+1 < _pattern.size() && _pattern[pos+1] == '%') {
165                     string_row.append(1, '%');
166                     pos += 2;
```

```
167                     continue;
168                 }
169                 if (string_row.empty() == false) {
170                     arry.push_back(std::make_tuple(string_row, "", 0));
171                     string_row.clear();
172                 }
173                 //当前位置是%字符位置
174                 pos += 1;//pos指向格式化字符位置
175                 if (pos < _pattern.size() && isalpha(_pattern[pos])) {
176                     format_key = _pattern[pos];//保存格式化字符
177                 }else {
178                     std::cout << &_pattern[pos-1] << "位置附近格式错误! \n";
179                     return false;
180                 }
181                 //pos指向格式化字符的下一个位置，判断是否包含有子格式 %d{%Y-%m-%d}
182                 pos += 1;
183                 if (pos < _pattern.size() && _pattern[pos] == '{') {
184                     sub_format_error = true;
185                     pos += 1;//pos指向花括号下一个字符处
186                     while(pos < _pattern.size()) {
187                         if (_pattern[pos] == '}') {
188                             sub_format_error = false;
189                             pos += 1;//让pos指向}的下一个字符处
190                             break;
191                         }
192                         format_val.append(1, _pattern[pos++]);
193                     }
194                 }
195                 arry.push_back(std::make_tuple(format_key, format_val, 1));
196                 format_key.clear();
197                 format_val.clear();
198             }
199             if (sub_format_error) {
200                 std::cout << "{}对应出错\n";
201                 return false;
202             }
203             if (string_row.empty() == false)
204                 arry.push_back(std::make_tuple(string_row, "", 0));
205             if (format_key.empty() == false)
206                 arry.push_back(std::make_tuple(format_key, format_val, 1));
207             for (auto &it : arry) {
208                 if (std::get<2>(it) == 0) {
209                     FormatItem::ptr fi(new OtherFormatItem(std::get<0>(it)));
210                     _items.push_back(fi);
211                 }else {
212                     FormatItem::ptr fi = createItem(std::get<0>(it),
    std::get<1>(it));
```

```
213                     if (fi.get() == nullptr) {
214                         std::cout <<"没有对应的格式化字符: %"<<std::get<0>(it) <<
    std::endl;
215                         return false;
216                     }
217                     _items.push_back(fi);
218                 }
219             }
220             return true;
221         }
222     private:
223         std::string _pattern;
224         std::vector<FormatItem::ptr> _items;
225 };
226 }
227 #endif
```

8.5 日志落地(LogSink)类设计（简单工厂模式）

日志落地类主要负责落地日志消息到目的地。

它主要包括以下内容：

- Formatter日志格式化器：主要是负责格式化日志消息，
- mutex互斥锁：保证多线程日志落地过程中的线程安全，避免出现交叉输出的情况。

这个类支持可扩展，其成员函数log设置为纯虚函数，当我们需要增加一个log输出目标， 可以增加一个类继承自该类并重写log方法实现具体的落地日志逻辑。

目前实现了三个不同方向上的日志落地：

- 标准输出：StdoutSink
- 固定文件：FileSink
- 滚动文件：RollSink
  - 滚动日志文件输出的必要性：
    - 由于机器磁盘空间有限， 我们不可能一直无限地向一个文件中增加数据
    - 如果一个日志文件体积太大，一方面是不好打开，另一方面是即时打开了由于包含数据巨大，也不利于查找我们需要的信息
    - 所以实际开发中会对单个日志文件的大小也会做一些控制，即当大小超过某个大小时（如1GB），我们就重新创建一个新的日志文件来滚动写日志。 对于那些过期的日志， 大部分企业内部都有专门的运维人员去定时清理过期的日志，或者设置系统定时任务，定时清理过期日志。
  - 日志文件的滚动思想：

日志文件滚动的条件有两个:文件大小 和 时间。我们可以选择:

- 日志文件在大于 1GB 的时候会更换新的文件
- 每天定点滚动一个日志文件

本项目基于文件大小的判断滚动生成新的文件

```
#ifndef __M_SINK_H__
#define __M_SINK_H__
#include "util.hpp"
#include "message.hpp"
#include "formatter.hpp"
#include <memory>
#include <mutex>

namespace bitlog{
class LogSink {
    public:
        using ptr = std::shared_ptr<LogSink>;
        LogSink() {}
        virtual ~LogSink() {}
        virtual void log(const char *data, size_t len) = 0;
};
class StdoutSink : public LogSink {
    public:
        using ptr = std::shared_ptr<StdoutSink>;
        StdoutSink() = default;
        void log(const char *data, size_t len) {
            std::cout.write(data, len);
        }
};
class FileSink : public LogSink {
    public:
        using ptr = std::shared_ptr<FileSink>;
        FileSink(const std::string &filename):_filename(filename) {
            util::file::create_directory(util::file::path(filename));
            _ofs.open(_filename, std::ios::binary | std::ios::app);
            assert(_ofs.is_open());
        }
        const std::string &file() {return _filename; }
        void log(const char *data, size_t len) {
            _ofs.write((const char*)data, len);
            if (_ofs.good() == false) {
                std::cout << "日志输出文件失败! \n";
            }
        }
```

```
40      private:
41          std::string _filename;
42          std::ofstream _ofs;
43  };
44  class RollSink : public LogSink {
45      public:
46          using ptr = std::shared_ptr<RollSink>;
47          RollSink(const std::string &basename, size_t max_fsize):
48              _basename(basename), _max_fsize(max_fsize), _cur_fsize(0){
49              util::file::create_directory(util::file::path(basename));
50          }
51          void log(const char *data, size_t len) {
52              initLogFile();
53              _ofs.write(data, len);
54              if (_ofs.good() == false) {
55                  std::cout << "日志输出文件失败! \n";
56              }
57              _cur_fsize += len;
58          }
59      private:
60          void initLogFile() {
61              if (_ofs.is_open() == false || _cur_fsize >= _max_fsize) {
62                  _ofs.close();
63                  std::string name = createFilename();
64                  _ofs.open(name, std::ios::binary | std::ios::app);
65                  assert(_ofs.is_open());
66                  _cur_fsize = 0;
67                  return;
68              }
69              return;
70          }
71          std::string createFilename() {
72              time_t t = time(NULL);
73              struct tm lt;
74              localtime_r(&t, &lt);
75              std::stringstream ss;
76              ss << _basename;
77              ss << lt.tm_year + 1900;
78              ss << lt.tm_mon + 1;
79              ss << lt.tm_mday;
80              ss << lt.tm_hour;
81              ss << lt.tm_min;
82              ss << lt.tm_sec;
83              ss << ".log";
84              return ss.str();
85          }
86      private:
```

```
87          std::string _basename;
88          std::ofstream _ofs;
89          size_t _max_fsize;
90          size_t _cur_fsize;
91  };
92
93  class SinkFactory {
94      public:
95          template<typename SinkType, typename ...Args>
96          static LogSink::ptr create(Args &&...args) {
97              return std::make_shared<SinkType>(std::forward<Args>(args)...);
98          }
99  };
100
101 }
102 #endif
```

8.6 日志器类(Logger)设计（建造者模式）

日志器主要是用来和前端交互， 当我们需要使用日志系统打印log的时候， 只需要创建Logger对象，调用该对象debug、info、warn、error、fatal等方法输出自己想打印的日志即可，支持解析可变参数列表和输出格式， 即可以做到像使用printf函数一样打印日志。

当前日志系统支持同步日志 & 异步日志两种模式，两个不同的日志器唯一不同的地方在于他们在日志的落地方式上有所不同：

同步日志器：直接对日志消息进行输出。

异步日志器：将日志消息放入缓冲区，由异步线程进行输出。

因此日志器类在设计的时候先设计出一个Logger基类，在Logger基类的基础上，继承出SyncLogger同步日志器和AsyncLogger异步日志器。

且因为日志器模块是对前边多个模块的整合，想要创建一个日志器，需要设置日志器名称，设置日志输出等级，设置日志器类型，设置日志输出格式，设置落地方向，且落地方向有可能存在多个，整个日志器的创建过程较为复杂，为了保持良好的代码风格，编写出优雅的代码，因此日志器的创建这里采用了建造者模式来进行创建。

```
1  #ifndef __M_LOG_H__
2  #define __M_LOG_H__
3  #include "util.hpp"
4  #include "level.hpp"
5  #include "message.hpp"
6  #include "formatter.hpp"
7  #include "sink.hpp"
8  #include "looper.hpp"
```

```cpp
#include <vector>
#include <list>
#include <atomic>
#include <unordered_map>
#include <cstdarg>
#include <type_traits>

namespace bitlog{
class Logger {
    public:
        enum class Type {
            LOGGER_SYNC = 0,
            LOGGER_ASYNC
        };
        using ptr = std::shared_ptr<Logger>;
        Logger(const std::string &name,
            Formatter::ptr formatter,
            std::vector<LogSink::ptr> &sinks,
            LogLevel::value level = LogLevel::value::DEBUG):
            _name(name),  _level(level), _formatter(formatter),
            _sinks(sinks.begin(), sinks.end()){
        }
        std::string loggerName() { return _name; }
        LogLevel::value loggerLevel() { return _level; }
        void debug(const char *file, size_t line, const char *fmt, ...) {
            if (shouldLog(LogLevel::value::DEBUG) == false) {
                return ;
            }
            va_list al;
            va_start(al, fmt);
            log(LogLevel::value::DEBUG, file, line, fmt, al);
            va_end(al);
        }
        void info(const char *file, size_t line, const char *fmt, ...) {
            if (shouldLog(LogLevel::value::INFO) == false) return ;
            va_list al;
            va_start(al, fmt);
            log(LogLevel::value::INFO, file, line, fmt, al);
            va_end(al);
        }
        void warn(const char *file, size_t line, const char *fmt, ...) {
            if (shouldLog(LogLevel::value::WARN) == false) return ;
            va_list al;
            va_start(al, fmt);
            log(LogLevel::value::WARN, file, line, fmt, al);
            va_end(al);
        }
```

```
56          void error(const char *file, size_t line, const char *fmt, ...) {
57              if (shouldLog(LogLevel::value::ERROR) == false) return ;
58              va_list al;
59              va_start(al, fmt);
60              log(LogLevel::value::ERROR, file, line, fmt, al);
61              va_end(al);
62          }
63          void fatal(const char *file, size_t line, const char *fmt, ...) {
64              if (shouldLog(LogLevel::value::FATAL) == false) return ;
65              va_list al;
66              va_start(al, fmt);
67              log(LogLevel::value::FATAL, file, line, fmt, al);
68              va_end(al);
69          }
70      public:
71          class Builder {
72              public:
73                  using ptr = std::shared_ptr<Builder>;
74                  Builder():_level(LogLevel::value::DEBUG),
75                      _logger_type(Logger::Type::LOGGER_SYNC) {}
76                  void buildLoggerName(const std::string &name) { _logger_name =
    name; }
77                  void buildLoggerLevel(LogLevel::value level) { _level = level;
    }
78                  void buildLoggerType(Logger::Type type) { _logger_type = type;
    }
79                  void buildFormatter(const std::string pattern) {
80                      _formatter = std::make_shared<Formatter>(pattern);
81                  }
82                  void buildFormatter(const Formatter::ptr &formatter) {
83                      _formatter = formatter;
84                  }
85                  template<typename SinkType, typename ...Args>
86                  void buildSink(Args &&...args) {
87                      auto sink = SinkFactory::create<SinkType>
    (std::forward<Args>(args)...);
88                      _sinks.push_back(sink);
89                  }
90                  virtual Logger::ptr build() = 0;
91              protected:
92                  Logger::Type _logger_type;
93                  std::string _logger_name;
94                  LogLevel::value _level;
95                  Formatter::ptr _formatter;
96                  std::vector<LogSink::ptr> _sinks;
97          };
98      protected:
```

```
 99          bool shouldLog(LogLevel::value level) { return level >= _level; }
100          void log(LogLevel::value level,const char *file,
101              size_t line, const char *fmt, va_list al) {
102              char *buf;
103              std::string msg;
104              int len = vasprintf(&buf, fmt, al);
105              if (len < 0) {
106                  msg = "格式化日志消息失败！！";
107              }else {
108                  msg.assign(buf, len);
109                  free(buf);
110              }
111              //LogMsg(name, file, line, payload, level)
112              LogMsg lm(_name, file, line, std::move(msg), level);
113              std::stringstream ss;
114              _formatter->format(ss, lm);
115              logIt(std::move(ss.str()));
116          }
117          virtual void logIt(const std::string &msg) = 0;
118      protected:
119          std::mutex _mutex;
120          std::string _name;
121          Formatter::ptr _formatter;
122          std::atomic<LogLevel::value> _level;
123          std::vector<LogSink::ptr> _sinks;
124  };
125
126  class SyncLogger : public Logger {
127      public:
128          using ptr = std::shared_ptr<SyncLogger>;
129          SyncLogger(const std::string &name,
130              Formatter::ptr formatter,
131              std::vector<LogSink::ptr> &sinks,
132              LogLevel::value level = LogLevel::value::DEBUG):
133              Logger(name, formatter, sinks, level){
134              std::cout << LogLevel::toString(level)<<" 同步日志器: "<< name<< "创
    建成功...\n";
135          }
136      private:
137          virtual void logIt(const std::string &msg) {
138              std::unique_lock<std::mutex> lock(_mutex);
139              if (_sinks.empty()) { return ; }
140              for (auto &it : _sinks) {
141                  it->log(msg.c_str(), msg.size());
142              }
143          }
144  };
```

```
145
146
147 class LocalLoggerBuilder: public Logger::Builder {
148     public:
149         virtual Logger::ptr build() {
150             if (_logger_name.empty()) {
151                 std::cout << "日志器名称不能为空！！";
152                 abort();
153             }
154             if (_formatter.get() == nullptr) {
155                 std::cout<<"当前日志器: " << _logger_name;
156                 std::cout<<" 未检测到日志格式,默认设置为: ";
157                 std::cout<<" %d{%H:%M:%S}%T%t%T[%p]%T[%c]%T%f:%l%T%m%n\n";
158                 _formatter = std::make_shared<Formatter>();
159             }
160             if (_sinks.empty()) {
161                 std::cout<<"当前日志器: "<<_logger_name<<" 未检测到落地方向，默认为
    标准输出!\n";
162                 _sinks.push_back(std::make_shared<StdoutSink>());
163             }
164             Logger::ptr lp;
165             if (_logger_type == Logger::Type::LOGGER_ASYNC) {
166                 lp = std::make_shared<AsyncLogger>(_logger_name,_formatter,
    _sinks, _level);
167             }else {
168                 lp = std::make_shared<SyncLogger>(_logger_name, _formatter,
    _sinks, _level);
169             }
170             return lp;
171         }
172 };
173 }
174
175 #endif
```

8.7 双缓冲区异步任务处理器（AsyncLooper）设计

设计思想：异步处理线程 + 数据池

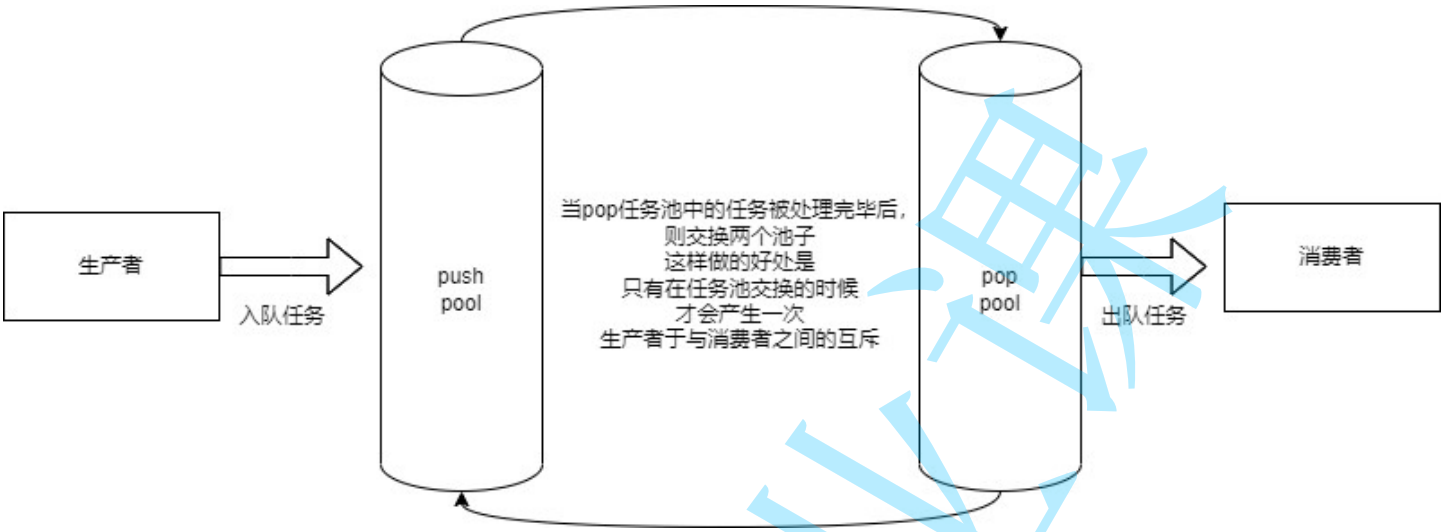
使用者将需要完成的任务添加到任务池中，由异步线程来完成任务的实际执行操作。

任务池的设计思想：双缓冲区阻塞数据池

优势：避免了空间的频繁申请释放，且尽可能的减少了生产者与消费者之间锁冲突的概率，提高了任务处理效率。

在任务池的设计中，有很多备选方案，比如循环队列等等，但是不管是哪一种都会涉及到锁冲突的情况，因为在生产者与消费者模型中，任何两个角色之间都具有互斥关系，因此每一次的任务添加与取

出都有可能涉及锁的冲突，而双缓冲区不同，双缓冲区是处理器将一个缓冲区中的任务全部处理完毕后，然后交换两个缓冲区，重新对新的缓冲区中的任务进行处理，虽然同时多线程写入也会冲突，但是冲突并不会像每次只处理一条的时候频繁（减少了生产者与消费者之间的锁冲突），且不涉及到空间的频繁申请释放所带来的消耗。



代码实现：

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <thread>
#include <mutex>
#include <atomic>
#include <condition_variable>
#include <functional>
#include <cassert>
namespace bitlog{
#define BUFFER_DEFAULT_SIZE (1*1024*1024)
#define BUFFER_INCREMENT_SIZE (1*1024*1024)
#define BUFFER_THRESHOLD_SIZE (10*1024*1024)
class Buffer {
    public:
        Buffer(): _reader_idx(0), _writer_idx(0), _v(BUFFER_DEFAULT_SIZE){}
        bool empty() { return _reader_idx == _writer_idx; }
        size_t readAbleSize() { return _writer_idx - _reader_idx; }
        size_t writeAbleSize() { return _v.size() - _writer_idx; }
        void reset() { _reader_idx = _writer_idx = 0; }
        void swap(Buffer &buf)  {
            _v.swap(buf._v);
            std::swap(_reader_idx, buf._reader_idx);
            std::swap(_writer_idx, buf._writer_idx);
        }
        void push(const char *data, size_t len) {
```

```
            assert(len <= writeAbleSize());
            ensureEnoughSpace(len);
            std::copy(data, data+len, &_v[_writer_idx]);
            _writer_idx += len;
        }
        const char*begin() { return &_v[_reader_idx]; }
        void pop(size_t len) {
            _reader_idx += len;
            assert(_reader_idx <= _writer_idx);
        }
    protected:
        void ensureEnoughSpace(size_t len) {
            if (len <= writeAbleSize()) return;
            /*每次增大1M大小*/
            size_t new_capacity;
            if (_v.size() < BUFFER_THRESHOLD_SIZE)
                new_capacity = _v.size() * 2 + len;
            else
                new_capacity = _v.size() + BUFFER_INCREMENT_SIZE + len;
            _v.resize(new_capacity);
        }
    private:
        size_t _reader_idx;
        size_t _writer_idx;
        std::vector<char> _v;
};
}
```

```
#ifndef __M_LOOP_H__
#define __M_LOOP_H__
#include "util.hpp"
#include <vector>
#include <thread>
#include <mutex>
#include <atomic>
#include <condition_variable>
#include <functional>
#include "buffer.hpp"

namespace bitlog{
    class AsyncLooper {
        public:
            using Functor = std::function<void(Buffer &buffer)>;
            using ptr = std::shared_ptr<AsyncLooper>;
```

```
17              AsyncLooper(const Functor &cb): _running(true),
_looper_callback(cb),
18                  _thread(std::thread(&AsyncLooper::worker_loop, this)) {
19              }
20              ~AsyncLooper() { stop(); }
21              void stop(){
22                  _running = false;
23                  _pop_cond.notify_all();
24                  _thread.join();
25              }
26              void push(const std::string &msg){
27                  if (_running == false) return;
28                  {
29                      std::unique_lock<std::mutex> lock(_mutex);
30                      _push_cond.wait(lock,
31                          [&]{ return _tasks_push.writeAbleSize() >= msg.size();
});
32                      _tasks_push.push(msg.c_str(), msg.size());
33                  }
34                  _pop_cond.notify_all();
35              }
36          private:
37              void worker_loop(){
38                  while(1){
39                      {
40                          std::unique_lock<std::mutex> lock(_mutex);
41                          if (_running == false && _tasks_push.empty()) {
return; }
42                          _pop_cond.wait(lock,[&]{ return
!_tasks_push.empty()||!_running; });
43                          _tasks_push.swap(_tasks_pop);
44                      }
45                      _push_cond.notify_all();
46                      _looper_callback(_tasks_pop);
47                      _tasks_pop.reset();
48                  }
49                  return;
50              }
51          private:
52              Functor _looper_callback;
53          private:
54              std::mutex _mutex;
55              std::atomic<bool> _running;
56              std::condition_variable _push_cond;
57              std::condition_variable _pop_cond;
58              Buffer _tasks_push;
59              Buffer _tasks_pop;
```

```
60              std::thread _thread;
61      };
62  }
63  #endif
```

8.8 异步日志器(AsyncLogger)设计

异步日志器类继承自日志器类， 并在同步日志器类上拓展了异步消息处理器。当我们需要异步输出日志的时候， 需要创建异步日志器和消息处理器， 调用异步日志器的log、error、info、fatal等函数输出不同级别日志。

- log函数为重写Logger类的函数， 主要实现将日志数据加入异步队列缓冲区中
- realLog函数主要由异步线程进行调用(是为异步消息处理器设置的回调函数)，完成日志的实际落地工作。

```
1
2   class AsyncLogger : public Logger {
3       public:
4           using ptr = std::shared_ptr<AsyncLogger>;
5           AsyncLogger(const std::string &name,
6               Formatter::ptr formatter,
7               std::vector<LogSink::ptr> &sinks,
8               LogLevel::value level = LogLevel::value::DEBUG):
9               Logger(name, formatter, sinks, level),
10              _looper(std::make_shared<AsyncLooper>(
11                  std::bind(&AsyncLogger::realLog, this,
    std::placeholders::_1))) {
12              std::cout << LogLevel::toString(level)<<"异步日志器: "<<name<<"创建成
    功...\n";
13          }
14      protected:
15          virtual void log(const std::string &msg) {
16              _looper->push(msg);
17          }
18          void realLog(Buffer &msg) {
19              if (_sinks.empty()) { return; }
20              for (auto &it : _sinks) {
21                  it->log(msg.begin(), msg.readAbleSize());
22              }
23          }
24      protected:
25          AsyncLooper::ptr _looper;
26  };
```

8.9 单例日志器管理类设计（单例模式）

日志的输出，我们希望能够在任意位置都可以进行，但是当我们创建了一个日志器之后，就会受到日志器所在作用域的访问属性限制。

因此，为了突破访问区域的限制，我们创建一个日志器管理类，且这个类是一个单例类，这样的话，我们就可以在任意位置来通过管理器单例获取到指定的日志器来进行日志输出了。

基于单例日志器管理器的设计思想，我们对于日志器建造者类进行继承，继承出一个全局日志器建造者类，实现一个日志器在创建完毕后，直接将其添加到单例的日志器管理器中，以便于能够在任何位置通过日志器名称能够获取到指定的日志器进行日志输出。

```
1
2  class loggerManager{
3      private:
4          std::mutex _mutex;
5          Logger::ptr _root_logger;
6          std::unordered_map<std::string, Logger::ptr> _loggers;
7      private:
8          loggerManager(){
9              std::unique_ptr<LocalLoggerBuilder> slb(new LocalLoggerBuilder());
10             slb->buildLoggerName("root");
11             slb->buildLoggerType(Logger::Type::LOGGER_SYNC);
12             _root_logger = slb->build();
13             _loggers.insert(std::make_pair("root", _root_logger));
14         }
15         loggerManager(const loggerManager&) = delete;
16         loggerManager &operator=(const loggerManager&) = delete;
17     public:
18         static loggerManager& getInstance() {
19             static loggerManager lm;
20             return lm;
21         }
22         bool hasLogger(const std::string &name)  {
23             std::unique_lock<std::mutex> lock(_mutex);
24             auto it = _loggers.find(name);
25             if (it == _loggers.end()) {
26                 return false;
27             }
28             return true;
29         }
30         void addLogger(const std::string &name, const Logger::ptr logger) {
31             std::unique_lock<std::mutex> lock(_mutex);
32             _loggers.insert(std::make_pair(name, logger));
33         }
34         Logger::ptr getLogger(const std::string &name) {
35             std::unique_lock<std::mutex> lock(_mutex);
```

```
36              auto it = _loggers.find(name);
37              if (it == _loggers.end()) {
38                  return Logger::ptr();
39              }
40              return it->second;
41          }
42          Logger::ptr rootLogger() {
43              std::unique_lock<std::mutex> lock(_mutex);
44              return _root_logger;
45          }
46  };
47
48  class GlobalLoggerBuilder: public Logger::Builder {
49      public:
50          virtual Logger::ptr build() {
51              if (_logger_name.empty()) {
52                  std::cout << "日志器名称不能为空！！";
53                  abort();
54              }
55              assert(loggerManager::getInstance().hasLogger(_logger_name) ==
    false);
56              if (_formatter.get() == nullptr) {
57                  std::cout << "当前日志器: " << _logger_name <<;
58                  std::cout << " 未检测到日志格式，默认设置为";
59                  std::cout << "[ %d{%H:%M:%S}%T%t%T[%p]%T[%c]%T%f:%l%T%m%n
    ]!\n";
60                  _formatter = std::make_shared<Formatter>();
61              }
62              if (_sinks.empty()) {
63                  std::cout << "当前日志器: " << _logger_name <<;
64                  std::cout << " 未检测到落地方向，默认设置为标准输出!\n";
65                  _sinks.push_back(std::make_shared<StdoutSink>());
66              }
67              Logger::ptr lp;
68              if (_logger_type == Logger::Type::LOGGER_ASYNC) {
69                  lp = std::make_shared<AsyncLogger>(_logger_name,_formatter,
    _sinks, _level);
70              }else {
71                  lp = std::make_shared<SyncLogger>(_logger_name, _formatter,
    _sinks, _level);
72              }
73              loggerManager::getInstance().addLogger(_logger_name, lp);
74              return lp;
75          }
76  };
```

8.10 日志宏&全局接口设计（代理模式）

提供全局的日志器获取接口。

使用代理模式通过全局函数或宏函数来代理Logger类的log、debug、info、warn、error、fatal等接口，以便于控制源码文件名称和行号的输出控制，简化用户操作。

当仅需标准输出日志的时候可以通过主日志器来打印日志。 且操作时只需要通过宏函数直接进行输出即可。

```
#ifndef __M_BIT_H__
#define __M_BIT_H__
#include "logger.hpp"

namespace bitlog {
    Logger::ptr getLogger(const std::string &name) {
        return loggerManager::getInstance().getLogger(name);
    }
    Logger::ptr rootLogger() {
        return loggerManager::getInstance().rootLogger();
    }

    #define debug(fmt, ...) debug(__FILE__, __LINE__, fmt, ##__VA_ARGS__)
    #define info(fmt, ...) info(__FILE__, __LINE__, fmt, ##__VA_ARGS__)
    #define warn(fmt, ...) warn(__FILE__, __LINE__, fmt, ##__VA_ARGS__)
    #define error(fmt, ...) error(__FILE__, __LINE__, fmt, ##__VA_ARGS__)
    #define fatal(fmt, ...) fatal(__FILE__, __LINE__, fmt, ##__VA_ARGS__)

    #define LOG_DEBUG(logger, fmt, ...) (logger)->debug(fmt, ##__VA_ARGS__)
    #define LOG_INFO(logger, fmt, ...) (logger)->info(fmt, ##__VA_ARGS__)
    #define LOG_WARN(logger, fmt, ...) (logger)->warn(fmt, ##__VA_ARGS__)
    #define LOG_ERROR(logger, fmt, ...) (logger)->error(fmt, ##__VA_ARGS__)
    #define LOG_FATAL(logger, fmt, ...) (logger)->fatal(fmt, ##__VA_ARGS__)

    #define LOGD(fmt, ...) LOG_DEBUG(bitlog::rootLogger(), fmt, ##__VA_ARGS__)
    #define LOGI(fmt, ...) LOG_INFO(bitlog::rootLogger(), fmt, ##__VA_ARGS__)
    #define LOGW(fmt, ...) LOG_WARN(bitlog::rootLogger(), fmt, ##__VA_ARGS__)
    #define LOGE(fmt, ...) LOG_ERROR(bitlog::rootLogger(), fmt, ##__VA_ARGS__)
    #define LOGF(fmt, ...) LOG_FATAL(bitlog::rootLogger(), fmt, ##__VA_ARGS__)
}

#endif
```

9. 功能用例

```cpp
#include "bitlog.h"

void loggerTest(const std::string &logger_name) {
    bitlog::Logger::ptr lp = bitlog::getLogger(logger_name);
    assert(lp.get());
    LOGF("------------example--------------------");
    lp->debug("%s", "logger->debug");
    lp->info("%s", "logger->info");
    lp->warn("%s", "logger->warn");
    lp->error("%s", "logger->error");
    lp->fatal("%s", "logger->fatal");
    LOG_DEBUG(lp, "%s", "LOG_DEBUG");
    LOG_INFO(lp, "%s", "LOG_INFO");
    LOG_WARN(lp, "%s", "LOG_WARN");
    LOG_ERROR(lp, "%s", "LOG_ERROR");
    LOG_FATAL(lp, "%s", "LOG_FATAL");
    LOGF("------------------------------------");

    std::string log_msg = "hello bitejiuyeke-";
    size_t fsize = 0;
    size_t count = 0;
    while(count < 1000000) {
        std::string msg = log_msg + std::to_string(count++);
        lp->error("%s", msg.c_str());
    }
}

int main(int argc, char *argv[])
{
    //实例化全局日志器建造者
    bitlog::GlobalLoggerBuilder::ptr lbp(new bitlog::GlobalLoggerBuilder);
    lbp->buildLoggerName("stdout_and_file_logger");//设置日志器名称
    lbp->buildFormatter("[%d][%c][%f:%l][%p] %m%n");//设置日志输出格式
    lbp->buildLoggerLevel(bitlog::LogLevel::value::DEBUG);//设置日志限制输出等级
    lbp->buildSink<bitlog::StdoutSink>(); //创建一个标准输出的落地方向
    lbp->buildSink<bitlog::FileSink>("./logs/sync.log");//创建一个文件落地方向
    lbp->buildSink<bitlog::RollSink>("./logs/roll-", 10 * 1024 * 1024);//创建滚动日志落地方向
    lbp->buildLoggerType(bitlog::Logger::Type::LOGGER_SYNC);//设置日志器类型为同步日志
    lbp->build(); //建造日志器
    loggerTest("stdout_and_file_logger");

    return 0;
}
```

10. 功能测试：

测试一个日志器中包含有所有的落地方向，观察是否每个方向都正常落地，分别测试同步方式和异步方式落地后数据是否正常。

```cpp
#include "bitlog.h"
#include "bench.h"
#include <unistd.h>

void loggerTest(const std::string &logger_name) {
    bitlog::Logger::ptr lp = bitlog::getLogger(logger_name);
    assert(lp.get());
    LOGF("------------example--------------------");
    lp->debug("%s", "logger->debug");
    lp->info("%s", "logger->info");
    lp->warn("%s", "logger->warn");
    lp->error("%s", "logger->error");
    lp->fatal("%s", "logger->fatal");
    LOG_DEBUG(lp, "%s", "LOG_DEBUG");
    LOG_INFO(lp, "%s", "LOG_INFO");
    LOG_WARN(lp, "%s", "LOG_WARN");
    LOG_ERROR(lp, "%s", "LOG_ERROR");
    LOG_FATAL(lp, "%s", "LOG_FATAL");
    LOGF("-------------------------------------");

    std::string log_msg = "hello bitejiuyeke-";
    size_t fsize = 0;
    size_t count = 0;
    while(count < 1000000) {
        std::string msg = log_msg + std::to_string(count++);
        lp->error("%s", msg.c_str());
    }
}
void functional_test() {
    bitlog::GlobalLoggerBuilder::ptr lbp(new bitlog::GlobalLoggerBuilder);
    lbp->buildLoggerName("stdout_and_file_logger");
    lbp->buildFormatter("[%d][%c][%f:%l][%p] %m%n");
    lbp->buildLoggerLevel(bitlog::LogLevel::value::DEBUG);
    lbp->buildSink<bitlog::StdoutSink>();
    lbp->buildSink<bitlog::FileSink>("./logs/sync.log");
    lbp->buildSink<bitlog::RollSink>("./logs/roll-", 10 * 1024 * 1024);
    lbp->buildLoggerType(bitlog::Logger::Type::LOGGER_ASYNC);
    lbp->build();
    loggerTest("stdout_and_file_logger");
}

```

```
int main(int argc, char *argv[])
{
    functional_test();
    return 0;
}
```

11. 性能测试

下面对日志系统做一个性能测试，测试一下平均每秒能打印多少条日志消息到文件。

主要的测试方法是：每秒能打印日志数 = 打印日志条数 / 总的打印日志消耗时间

主要测试要素：同步/异步 & 单线程/多线程

- 100w+条指定长度的日志输出所耗时间
- 每秒可以输出多少条日志
- 每秒可以输出多少MB日志

测试环境：

- CPU：AMD Ryzen 7 5800H with Radeon Graphics　　3.20 GHz
- RAM：16G DDR4 3200
- ROM：512G-SSD
- OS：ubuntu-22.04TLS虚拟机（2CPU核心/4G内存）

```
#ifndef __M_BENCH_H__
#define __M_BENCH_H__
#include "bitlog.h"
#include <chrono>

namespace bitlog {
void bench(const std::string &loger_name, size_t thread_num,
    size_t msglen, size_t msg_count)
{
    Logger::ptr lp = getLogger(loger_name);
    if (lp.get() == nullptr) return;
    std::string msg(msglen, '1');
    size_t msg_count_per_thread = msg_count / thread_num;
    std::vector<double> cost_time(thread_num);
    std::vector<std::thread> threads;
    std::cout << "输入线程数量: " << thread_num << std::endl;
    std::cout << "输出日志数量: " << msg_count << std::endl;
    std::cout << "输出日志大小: " << msglen * msg_count / 1024  << "KB" <<
std::endl;
    for (int i = 0; i < thread_num; i++) {
```

```
        threads.emplace_back([&, i](){
            auto start = std::chrono::high_resolution_clock::now();
            for(size_t j = 0; j < msg_count_per_thread; j++) {
                lp->fatal("%s", msg.c_str());
            }
            auto end = std::chrono::high_resolution_clock::now();
            auto cost=std::chrono::duration_cast<std::chrono::duration<double>>(end-start);
            cost_time[i] = cost.count();
            auto avg = msg_count_per_thread / cost_time[i];
            std::cout << "线程" << i << "耗时: " << cost.count() << "s";
            std::cout << " 平均: " << (size_t)avg << "/s\n";
        });
    }
    for(auto &thr : threads) {
        thr.join();
    }
    double max_cost = 0;
    for (auto cost : cost_time) max_cost = max_cost < cost ? cost : max_cost;
    std::cout << "总消耗时间: " << max_cost << std::endl;
    std::cout << "平均每秒输出: " << (size_t)(msg_count / max_cost) << std::endl;
}
}

#endif
```

```
#include "bitlog.h"
#include "bench.h"
#include <unistd.h>

void sync_bench_thread_log(size_t thread_count, size_t msg_count, size_t msglen)
{
    static int num = 1;
    std::string logger_name = "sync_bench_logger" + std::to_string(num++);
    LOGI("************************************************");
    LOGI("同步日志测试: %d threads, %d messages", thread_count, msg_count);

    bitlog::GlobalLoggerBuilder::ptr lbp(new bitlog::GlobalLoggerBuilder);
    lbp->buildLoggerName(logger_name);
    lbp->buildFormatter("%m%n");
    lbp->buildSink<bitlog::FileSink>("./logs/sync.log");
    lbp->buildLoggerType(bitlog::Logger::Type::LOGGER_SYNC);
    lbp->build();
    bitlog::bench(logger_name, thread_count, msglen, msg_count);
```

```
    LOGI("**********************************************");
}
void async_bench_thread_log(size_t thread_count, size_t msg_count, size_t msglen)
{
    static int num = 1;
    std::string logger_name = "async_bench_logger" + std::to_string(num++);
    LOGI("**********************************************");
    LOGI("异步日志测试: %d threads, %d messages", thread_count, msg_count);

    bitlog::GlobalLoggerBuilder::ptr lbp(new bitlog::GlobalLoggerBuilder);
    lbp->buildLoggerName(logger_name);
    lbp->buildFormatter("%m");
    lbp->buildSink<bitlog::FileSink>("./logs/async.log");
    lbp->buildLoggerType(bitlog::Logger::Type::LOGGER_ASYNC);
    lbp->build();
    bitlog::bench(logger_name, thread_count, msglen, msg_count);
    LOGI("**********************************************");
}
void bench_test() {
    // 同步写日志
    sync_bench_thread_log(1, 1000000, 100);
    sync_bench_thread_log(5, 1000000, 100);
    /*异步日志输出，为了避免因为等待落地影响时间所以日志数量降低为小于缓冲区大小进行测试*/
    async_bench_thread_log(1, 100000, 100);
    async_bench_thread_log(5, 100000, 100);
}

int main(int argc, char *argv[])
{
    bench_test();
    return 0;
}
```

```
dev@bite:~/logger-v2$ ./logger
当前日志器: root 未检测到日志格式，默认设置为[ %d{%H:%M:%S}%T%t%T[%p]%T[%c]%T%f:%l%T%m%n ]!
当前日志器: root 未检测到落地方向，默认设置为标准输出!
DEBUG 同步日志器: root创建成功...
[09:48:34][140319692223424][INFO][root][logger.cc:62] ************************************
[09:48:34][140319692223424][INFO][root][logger.cc:63] 异步日志测试: 1 threads, 10000000 messages
DEBUG异步日志器: async_bench_logger1创建成功...
输入线程数量: 1
```

```
输出日志数量: 10000000
输出日志大小: 976562KB
线程0耗时: 22.8947s 平均: 436782/s
总消耗时间: 22.8947
平均每秒输出: 436782条日志
平均每秒输出: 41MB
[09:48:57][140319692223424][INFO][root][logger.cc:72] ************************************
[09:48:57][140319692223424][INFO][root][logger.cc:62] ************************************
[09:48:57][140319692223424][INFO][root][logger.cc:63] 异步日志测试: 5 threads, 10000000 messages
DEBUG异步日志器: async_bench_logger2创建成功...
输入线程数量: 5
输出日志数量: 10000000
输出日志大小: 976562KB
线程3耗时: 7.3118s 平均: 273530/s
线程1耗时: 7.46903s 平均: 267772/s
线程0耗时: 7.48037s 平均: 267366/s
线程2耗时: 7.59597s 平均: 263297/s
线程4耗时: 7.60846s 平均: 262865/s
总消耗时间: 7.60846
平均每秒输出: 1314325条日志
平均每秒输出: 125MB
[09:49:05][140319692223424][INFO][root][logger.cc:72] ************************************
[09:49:05][140319692223424][INFO][root][logger.cc:46] ************************************
[09:49:05][140319692223424][INFO][root][logger.cc:47] 同步日志测试: 1 threads, 10000000 messages
DEBUG 同步日志器: sync_bench_logger1创建成功...
输入线程数量: 1
输出日志数量: 10000000
输出日志大小: 976562KB
线程0耗时: 8.55658s 平均: 1168691/s
总消耗时间: 8.55658
平均每秒输出: 1168691条日志
平均每秒输出: 111MB
[09:49:14][140319692223424][INFO][root][logger.cc:56] ************************************
[09:49:14][140319692223424][INFO][root][logger.cc:46] ************************************
[09:49:14][140319692223424][INFO][root][logger.cc:47] 同步日志测试: 5 threads, 10000000 messages
DEBUG 同步日志器: sync_bench_logger2创建成功...
输入线程数量: 5
输出日志数量: 10000000
```

```
输出日志大小：976562KB
线程1耗时：9.42852s 平均：212122/s
线程3耗时：9.56269s 平均：209146/s
线程4耗时：9.62333s 平均：207828/s
线程2耗时：9.68728s 平均：206456/s
线程0耗时：9.71674s 平均：205830/s
总消耗时间：9.71674
平均每秒输出：1029151条日志
平均每秒输出：98MB
[09:49:23][140319692223424][INFO][root][logger.cc:56]
**************************************
```

能够通过上边的测试看出来，一些情况：

在单线程情况下，异步效率看起来还没有同步高，这个我们得了解，现在的IO操作在用户态都会有缓冲区进行缓冲区，因此我们当前测试用例看起来的同步其实大多时候也是在操作内存，只有在缓冲区满了才会涉及到阻塞写磁盘操作，而异步单线程效率看起来低，也有一个很重要的原因就是单线程同步操作中不存在锁冲突，而单线程异步日志操作存在大量的锁冲突，因此性能也会有一定的降低。

但是，我们也要看到限制同步日志效率的最大原因是磁盘性能，打日志的线程多少并无明显区别，线程多了反而会降低，因为增加了磁盘的读写争抢，而对于异步日志的限制，并非磁盘的性能，而是cpu的处理性能，打日志并不会因为落地而阻塞，因此在多线程打日志的情况下性能有了显著的提高。

12. 扩展

- 丰富sink类型：
  - 支持按小时按天滚动文件
  - 支持将log通过网络传输落地到日志服务器(tcp/udp)
  - 支持在控制台通过日志等级渲染不同颜色输出方便定位
  - 支持落地日志到数据库
  - 支持配置服务器地址，将日志落地到远程服务器
- 实现日志服务器负责存储日志并提供检索、分析、展示等功能

13. 参考

https://www.imangodoc.com/174918.html

https://blog.csdn.net/w1014074794/article/details/125074038

https://zhuanlan.zhihu.com/p/472569975

https://zhuanlan.zhihu.com/p/460476053

https://gitee.com/davidditao/DDlog

https://www.cnblogs.com/ailumiyana/p/9519614.html

https://gitee.com/lqk1949/plog/

https://www.cnblogs.com/horacle/p/15494358.html

https://blog.csdn.net/qq_29220369/article/details/127314390