OOD作业

一、扩展阅读

[OO真经]:
https://www.cnblogs.com/leoo2sk/archive/2009/04/09/1432103.html

[function bind的救赎]
https://blog.csdn.net/myan/article/details/5928531

二、问答题

1. 类与类之间的关系有哪几种？各自的特点怎样的?

- 继承关系(泛化):继承是指一个类继承另一个类的功能，并增加自己新功能的能力。
- 依赖关系：简单理解就是一个类A用到了另一个类B，而这种使用关系是具有偶然性的零时性的，非常弱的，但是类B的变化会影响到类A。
- 关联关系：关联关系是两个类之间的语义级别的强依赖关系，关联可以是单向的、双向的，
- 聚合关系：聚合关系是一种特例，他体现的是一种整体与局部的关系，即has-a关系。此时整体与局部是可分离的，他们可以具有各自的生命周期。
- 组合关系：组合也是关联关系的一种特例，他体现的是一种类和类或者类和接口间的纵向关系。
- 实现关系：实现是指一个class了哦实现一个interface接口的功能，实现是类与接口之间最长见的关系

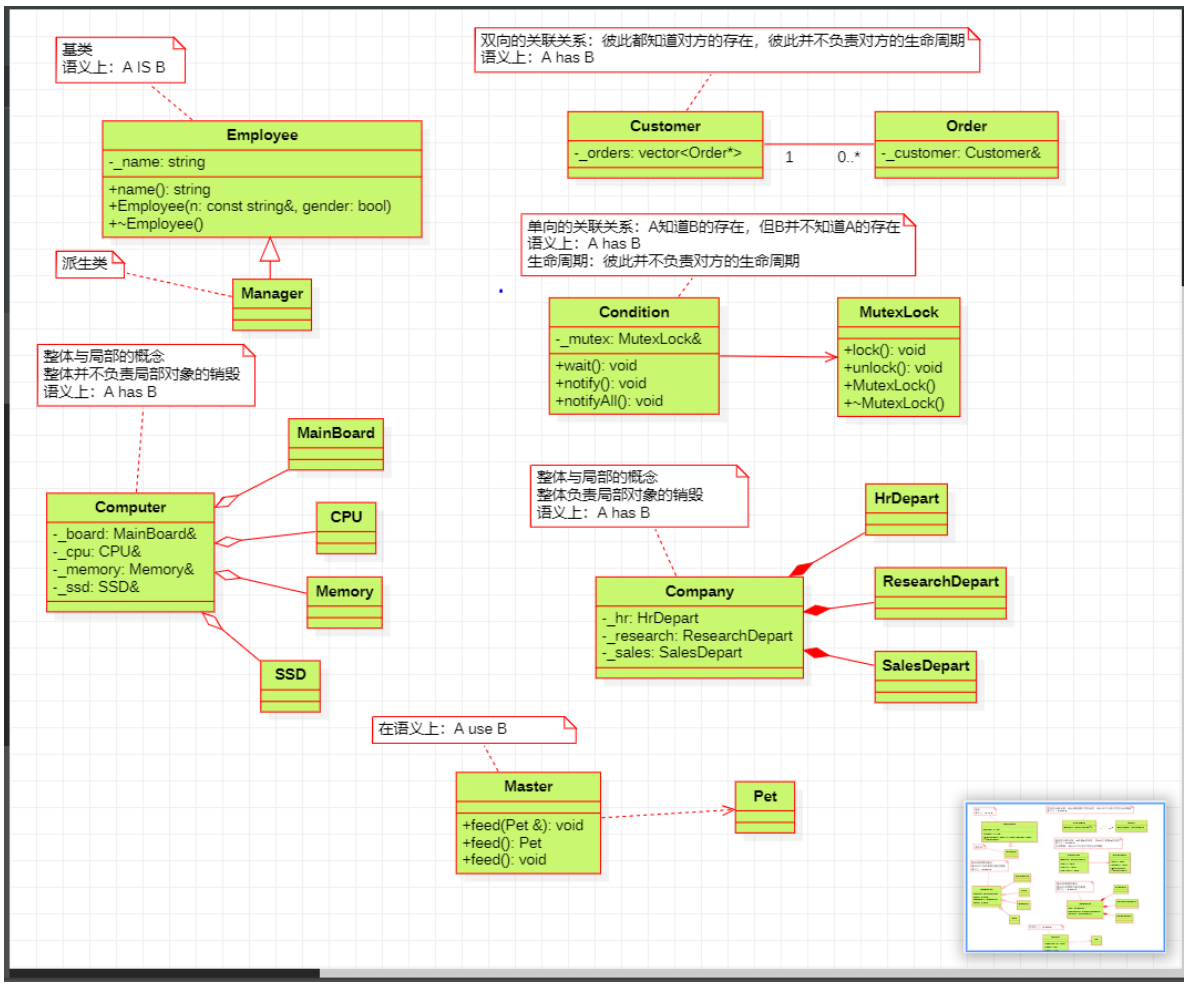
  》关系的强弱程度：组合>聚合>关联>依赖

1. 面向对象设计有哪些原则?

- 单一职责原则（the single responsibility principle）：解耦增强内聚性
- 开闭原则(the open closed principle)：对扩展开放，对修改闭合
- 里式替换原则(the liskov substitution principle)：派生类必须能完全替换基类的功能
- 依赖倒置原则（the dependency inversion principle）：面向接口编程，依赖与抽象（依赖抽象而不是实例）
- 接口分离原则(the interface segregation principle)：多个特定客户端的接口要好于一个宽泛用途的接口
- 迪米特法则(law of demeter)：对象之间要尽量少了解对方。
- 组合复用原则(composite reuse principle)：尽量使用对象组合，而不是继承来达到复用的目的。

三、画图题

1. 运用所学的UML类图的知识，画出“文本查询程序的扩展”作业的类图。(提交类图png格式)



四、编程题

1. 在封装 `Linux` 下互斥锁和条件变量互斥锁 `MutexLock` 和条件变量 `Condition` 类的框架如下：

```
class MutexLock {
public:
   //...
    void lock();
    void unlock();
private:
      //...
};

class Condition {
public:
   //...
    void wait();
    void notify();
    void notifyall();
 private:
    //...
};
```

合并到第三题

2. 在`MutexLock`和`Condition`的基础之上，运用动态多态的知识继续封装线程类`Thread`，然后实现生产者与消费者问题（提示：其中线程类中有一个纯虚函数,一个线程要执行的任务交给他的派生类来完成。）

```
class Thread {
public:
 //...
 void start();
private:
 virtual void run()=0;   //实现任务
private:
 //...
};

class MyThread
: public Thread {
public:
 //...
private:
 void run()
 {
     //....do something
 }
private:
 //...
};
```

```
#include <iostream>
#include <mutex>
using std::cin;
using std::cout;
using std::endl;
class Condition;
class Thread;
class MutexLock {
    friend Condition;
    friend Thread;
public:
    MutexLock()
    {
        init();
    }
    static void init(){
        pthread_mutex_init(&_mutex,NULL);
    }
    static void lock(){
        pthread_mutex_lock(&_mutex);
    }
    static void unlock(){
        pthread_mutex_unlock(&_mutex);
    }
    static void destroy(){
        pthread_mutex_destroy(&_mutex);

    }
    ~MutexLock(){
```

```
        destroy();
    }
private:
protected:
    pthread_mutex_t &getMutex(){
        return _mutex;
    }
    static pthread_mutex_t _mutex;
};

class Condition {
    friend Thread;
public:
    Condition(){
        init();
    }
    static void init(){
        pthread_cond_init(&_cond,NULL);
    }
    static void wait(){
        pthread_cond_wait(&_cond,&MutexLock::_mutex);
    }
    static void notify(){
        pthread_cond_signal(&_cond);
    }
    static void notifyall(){
        pthread_cond_broadcast(&_cond);
    }
    static void destroy(){
        pthread_cond_destroy(&_cond);

    }
    ~Condition(){
        destroy();
    }
private:
    static pthread_cond_t _cond;
};

class Thread {
    /* friend MutexLock; */
    /* friend Condition; */
public:
    //...
    void start(){
        run();
    }
private:
    virtual void run()=0;   //实现任务
private:
    pthread_t _thid;
};

class MyThread_product
: public Thread {
public:
    //...
private:
```

```
    void run() override
    {
        MutexLock::lock();
        cout<<"i am product"<<endl;
        MutexLock::unlock();
        Condition::notify();
    }
private:
    //...
};
class MyThread_consume
: public Thread {
public:
    //...
private:
/* public: */
    void run() override
    {
        Condition::wait();
        MutexLock::lock();
        cout<<"i am consume"<<endl;
        MutexLock::unlock();

    }
private:
    //...
};
int main()
{
    MutexLock mu;
    Condition ct;
    /* MyThread_consume m1; */
    /* m1.start(); */
    /* MyThread_product m2; */
    return 0;
}
```