作业PA2-2 报告

1. 涉及数据结构和相关背景

- 队列的特性
- 队列的应用
- 如何进行解耦和削峰
- 如何进行不同线程之间的沟通
- 生产者消费者模式
- 异步通信

2. 实验内容

2.1 问题描述

某宝平台定期要搞一次大促，大促期间的并发请求可能会很多，比如每秒3000个请求。假设我们现在有两台机器处理请求，并且每台机器只能每次处理1000个请求。如图（1）所示，那多出来的1000个请求就被阻塞了（没有系统响应）。

请你实现一个消息队列，如图（2）所示。系统B和系统C根据自己能够处理的请求数去消息队列中拿数据，这样即便每秒有8000个请求，也只是把请求放在消息队列中。如何去拿消息队列中的消息由系统自己去控制，甚至可以临时调度更多的机器并发处理这些请求，这样就不会让整个系统崩溃了。

注：现实中互联网平台的每秒并发请求可以达到千万级。

2.2 基本要求

定义顺序队列类型，使其具有如下功能：

（1）建立一个空队列；

（2）释放队列空间，将队列销毁；

（3）将队列清空，变成空队列；

（4）判断队列是否为空；

（5）返回队列内的元素个数；

（6）将队头元素弹出队列（出队）；

（7）在队列中加入一个元素（入队）；

（8）从队头到队尾将队列中的元素依次输出。

2.3 数据结构设计

- 基本数据结构--模板链栈实现
  - 链栈节点

```
template<class T>
struct Node
{
    T item;
    Node* next;
    Node* prev;
};
```

  - 链栈构造函数及析构函数

```
LinkQueue()
{
    cnt = 0;
    Node<T>* anode = new Node<T>;
    anode->next = NULL;
    anode->prev = NULL;
    //声明两个节点
    head = anode;
    rear = anode;
}

~LinkQueue()
{
    Node<T>* p = head;
    for (int i = 0; i < cnt; i++)
    {
        dequeue();
    }
}
```

  - 链栈出队

```
void dequeue()
{
    if (isEmpty())
    {
        cout << "ERROR : Queue is Empty." << endl;
    }

    else if (cnt == 1)
    {
        cnt--;
    }

    else
    {
        Node<T>* p = head;
```

```
        head = p->prev;
        delete p;
        cnt--;
    }
}
```

> 链栈出队的时候要注意在最后尾指针不能删除，而是需要进行特判，将器元素删除即可

- 链栈入队

```
void enqueue(T item)
{
    if (cnt == 0)
    {
        rear->item = item;
        cnt++;
    }

    else
    {
        //这个链表是往左边延申的
        Node<T>* anode = new Node<T>;
        anode->item = item;
        anode->next = rear;
        rear->prev = anode;
        rear = anode;
        rear->prev = NULL;
        cnt++;
    }
}
```

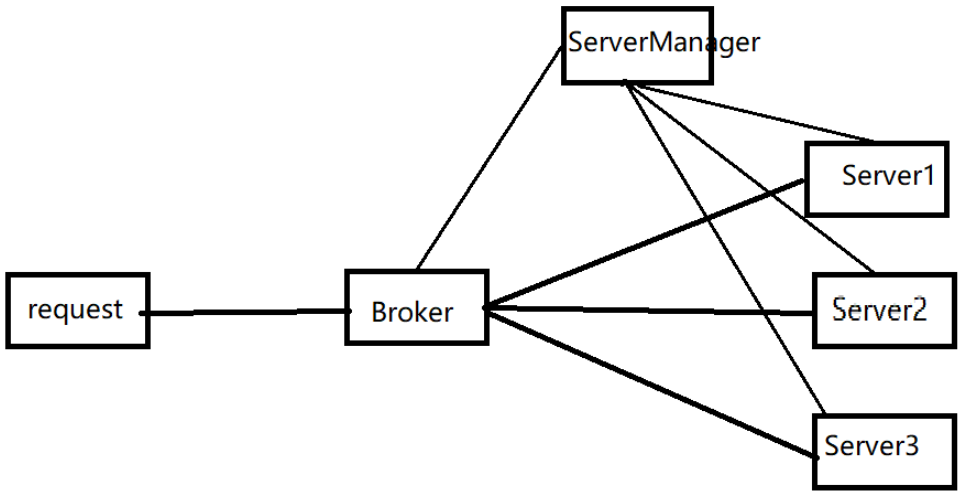
- 取队首数值、返回长度、判空、展示队列信息

```
unsigned int size()
{
    return cnt;
}

bool isEmpty()
{
    return cnt == 0;
}

T getHead()
{
    return head->item;
}

void showInfo()
{
    if (isEmpty())
    {
```

```
            cout << "ERROR : Queue is empty." << endl;
        }
        else
        {
            for (Node<T>* ptr = head; ptr != NULL; ptr = ptr->prev)
            {
                cout << ptr->item << endl;  //注意不一定能直接输出
            }
        }
    }
```

> 利用模板进行链栈的模拟，进行展示信息的时候需要注意这里的item不一定是内置的数据类型，有可能需要运算符重载或者进行子类继承重写方法进行覆盖

- 总体设置



  - request Source产生需求发送给Broker
  - Broker充当缓冲队列，进行缓冲请求的存储
  - 进入Broker的请求会直接给客户端进行回显，表示提交成功
  - Server用Server Manager进行管理，每一个Server处理相同的业务，当空闲时主动在队列中获取请求进行处理
  - 当Broker处于空状态时，触发Broker的consumer锁，此时Server均无法进行请求的处理
  - 当Broker处于队满状态时，触发Producer锁。此时request产生的需求无法进入队列中，显示提交失败
  - 当Broker中的存储率达到了其最大值的75%以上时，Broker会寻找另外可用的Server，将其加到自己的下游，分担请求处理任务

- 生产者消费者模式：
  - 利用中间添加缓冲区的方式进行两个线程之间的通讯（产生请求和处理需求），使得需求产生端与服务端的耦合度降低，对后续进行维护或者在服务端添加新功能有益
  - 对于某服务器故障，其他服务器能够正常工作
  - 对于处理需求的时间来说时同步的
  - 用户回显，减少了用户的等待时间

- 同步与异步通信
  - 同步通信
    - 同步调用的优点：
      - 时效性较强，可以立即得到结果
    - **同步调用的问题：**
      - 耦合度高、
      - 性能和吞吐能力下降
      - 有额外的资源消耗（不能同步的时间消耗以及硬件资源消耗）
      - 有级联失败问题 （Server挂了导致前方Broker一直请求也挂了）
  - 异步通信
    - 优点
      - 耦合度极低，每个服务都可以灵活插拔，可替换
      - 吞吐量提升：**无需等待订阅者处理完成**，响应更快速
      - 故障隔离：**服务没有直接调用，不存在级联失败问题**
      - 调用间没有阻塞，不会造成无效的资源占用
      - 流量削峰：不管发布事件的流量波动多大，都由Broker接收，订阅者可以按照自己的速度去处理事件（这个就是此题中阻塞队列中的作用）
    - 缺点
      - **需要依赖于Broker的可靠、安全、性能**
      - 不好管理和问题排查

2.4 功能说明

- 设置模拟需求

```
#pragma once
#include <iostream>
#include <string>
using namespace std;


class Order
{
public:
    Order(string name = "unknown", string req = "none", int time = 0, int
seq=0)
    {
        this->username = name;
        this->request = req;
        this->waitingTime = time;
        this->seqN = seq;
    }



    string username;    //用户姓名
    string request;     //用户请求内容
```

```
    int waitingTime;    //用户等待时间
    int seqN;           //需求序列号
};
```

> 在这里Order具有用户行，请求以及请求的序列号，主要用来尽心需求之间的区分

- 设置需求源

```
#pragma once
#include "Order.h"
#include <ctime>
#include <string>
#include <iostream>
using namespace std;


class ReSource
{
public:
    ReSource();

    string randStr();       //随机产生用户名

    string randRequest();   //随机产生请求

    Order Request();        //发出请求

    string orderList[4] = { "pay", "collect", "browse", "search" };     //请求集合

};
```

> 需求源包括
>
> - 产生随机的用户名和随机的请求(在一定范围内请求)
> - 产生随机的序列号用于需求区分

  - 需求源获取随机信息

```
string ReSource::randStr()
{
    /*产生随机用户名*/
    string temp = "";
    temp += (char)(rand() % 26 + 'A');
    for (int i = 1; i <= 4; i++)
    {
        temp += (char)(rand() % 26 + 'a');
    }

    return temp;
```

```
}

string ReSource::randRequest()
{
    /*产生随机请求（在请求集中）*/
    return this->orderList[rand() % 4];
}
```

- 需求源产生信息

```
Order ReSource::Request()
{
    return Order(this->randStr(), randRequest(), 0, rand()%1000);

}
```

- 设置代理器

```
#pragma once
#include "LinkQueue.hpp"
#include "Order.h"
#include "RequestSource.h"


class Broker : public LinkQueue<Order>
{

public:
    Broker(string name="local", int max=500);

    bool receiveRequest();  //受到消息源的请求

    Order popRequest();     //给出请求

    void _lockProducer();   //检查生产者锁

    void _lockConsumer();   //检查消费者锁

    int returnMesseage();   //向用户发送回显

    void showInfo();        //展示缓冲队列中的信息

    int addServer();        //添加服务器


    bool producerLock;      //生产者锁
    bool consumerLock;      //消费者锁
    int maxCap;             //缓冲区最大容量
    string name;            //broker名称
```

```
};
```

- 代理器实现
  - 构造函数

```
Broker::Broker(string name, int max)
{
    /*输入名称（默认是LOCAL）以及该缓冲队列的最大容量*/
    this->producerLock = false;     //设置生产者锁和消费者锁均没有
    this->consumerLock = false;
    this->name = name;
    this->maxCap = max;

}
```

  - 受到需求源产生的需求

```
bool Broker::receiveRequest()
{
    /*需求进入缓冲队列中*/
    if (this->producerLock == false)
    {
        this->enqueue(ReSource().Request());    //其实在实现时是主动去源力去去取
        cout << "Request is Submitted!" << endl;        //即时向用户发送回显
        return true;
    }
    return false;
}
```

  - 给出需求

```
Order Broker::popRequest()
{
    /*后端服务器接收到请求*/
    Order temp = this->getHead();   //利用继承链队列出队列
    this->dequeue();
    return temp;
}
```

  - 进行生产者锁和消费者锁的设置

```
void Broker::_lockProducer()
{
    /*当队列中元素数量达到了最大数量时将生产者锁开启*/
    if (this->cnt == this->maxCap)
    {
        this->producerLock = true;
    }
    else
```

```
    {
        this->producerLock = false;
    }
}

void Broker::_lockConsumer()
{
    /*当队里额中没有元素的时候消费者锁开启*/
    if (this->isEmpty())
    {
        this->consumerLock = true;
    }
    else
    {
        this->consumerLock = false;
    }

}
```

- 展示缓冲区内的需求信息

```
void Broker::showInfo()
{
    /*主要是重写了链队列中的信息展示，因为Order无法直接输出*/
    if (this->isEmpty())
    {
        cout << "ERROR : Queue is empty." << endl;
    }
    else
    {
        /*将模板中的直接输出节点元素具体化*/
        for (Node<Order>* ptr = this->head; ptr != NULL; ptr = ptr-
>prev)
        {
            cout << "(" << ptr->item.seqN << ")" << '\t' << ptr-
>item.username << '\t' << ptr->item.request << endl;
        }
    }
}
```

- 对用户需求进行回应

```
int Broker::returnMesseage()
{
    /*当需求进入缓冲队列之后即时向用户发回显*/
    cout << "Request is submitted" << endl;
    return 1;
}
```

- 添加server

```
int Broker::addServer()
{
    /*设置当Broker中存储的请求超过Broker最大容量的75%时，就在最后天机贡多的Server来减轻Broker的工作强度*/
    if (this->size() > 75 * this->maxCap / 100)
    {
        return (this->size() - this->maxCap * 75 / 100) / (this->maxCap * 25 / 100 / 5);        //设置增加的Server的数量是多于75%的部分与总体容量5%的比值，返回要添加的数量
    }
    return 0;
}
```

- 设置Server

```
#pragma once
#include <iomanip>
#include "Order.h"
#include "Broker.h"
#include "LinkQueue.hpp"

class Server
{
public:

    Server(bool iM = true, int seq = -1);    //设置Server的空闲状态

    bool handle();                           //Server进行数据的处理

    bool connectBroker(Broker* tpbk);        //连接相应的Broker

    bool fetch();                            //在缓冲队列中取需求

    bool work();                             //Server工作时涉及到的函数集


    bool idleMode;                           //是否闲置
    int seqNum;                              //Sever序列号
    Broker* bk;                              //连接的Broker
    Order temp;                              //正在处理的需求

};
```

- 服务器处理请求

```
bool Server::handle()
{
    /*服务器处理需求*/
    cout << "NO." << setw(5) << this->seqNum << "Server is working on
request:"
         << setw(15) << "(" << this->temp.seqN << ")" << this->temp.request
<< endl;

    return true;
}
```

- 服务器连接代理器

```
bool Server::connectBroker(Broker* tpbk)
{

    this->bk = tpbk;
    return true;
}
```

- 服务器取代理器中的需求

```
bool Server::fetch()
{
    if (this->bk->consumerLock == false)
    {
        this->temp = this->bk->popRequest();
        return true;
    }
    return false;
```

- 服务器工作

```
bool Server::work()
{
    /*首先进行fetch，fetch成功之后进行需求的处理*/
    if (this->fetch())
    {
        this->handle();
        return true;
    }

    return false;

}
```

- 设置Server管理器

```
#pragma once
#include "Server.h"
#include "Broker.h"

class ServerManager
{
public:
    ServerManager(int n=5);                 //初始化Server的数量

    void connectBroker(Broker* b);          //连接Broker

    int addsv();                            //添加服务器


    int totNum;                             //当前所有工作的夫区其的数量
    Server servergroup[1000];               //Server存储集合
    Broker* bk;                             //连接的Broker

};
```

- 初始化ServerManager

```
ServerManager::ServerManager(int n)
{
    this->totNum = 0;
    for (int i = 1; i <= n; i++)    //初始化Server的时候需要指定服务器台数
    {
        Server sv = Server();       /*初始化Server并且将其添加到集合当中*/
        sv.seqNum = i;
        this->servergroup[totNum] = sv;

        totNum++;
    }
}
```

- 连接到Broker上

```
void ServerManager::connectBroker(Broker* b)
{
    this->bk = b;
}
```

- 添加服务器

```
int ServerManager::addsv()
{
    if (this->bk->addServer())              //在Broker当中返回了需要多少台Server
    {
        for (int i = 1; i <= this->bk->addServer(); i++)  //添加Server
        {
            totNum++;
            Server sv = Server();
```

```
            sv.seqNum = i;
            this->servergroup[totNum] = sv;
        }
    }

    return this->bk->addServer();
}
```

- 主函数调用
  - 初始化缓冲队列和需求源

```
Broker broker = Broker();
ReSource source = ReSource();
```

  - 创建服务器管理器

```
int numServer;
cout << "Enter the number of Server" << endl;
cin >> numServer;
ServerManager svg = ServerManager(numServer);  //利用服务器管理器创建服务器
svg.connectBroker(&broker);
```

  - 将服务器与缓冲队列连接

```
for (int i = 0; i < svg.totNum; i++)
{
    svg.servergroup[i].connectBroker(&broker);
    //cout << svg.servergroup[i].bk->name << endl;
}
```

  - 进行模拟

```
int choice = 0;
cout << "Enter your choice:" << endl << "0: quit " << endl << "1:
simulation" << endl;
cin >> choice;
if (choice == 1)
{
    while (1)
    {
        //模拟需求进入
        for (int i = 1; i <= rand() % numServer; i++)
        {
            if (!broker.receiveRequest())
            {
                break;
            }

            broker._lockProducer();
        }
```

```
            //当需求达到缓冲队列的一定比例时，添加一些服务器应对洪峰
            int added = svg.addsv();
            numServer += added;

            //将添加的服务器与缓冲队列进行连接
            for (int i = 0; i < svg.totNum; i++)
            {
                svg.servergroup[i].connectBroker(&broker);
                //cout << svg.servergroup[i].bk->name << endl;
            }


            //显示当前缓冲队列中的需求
            broker.showInfo();
            cout << endl;


            //更新生产者锁和消费者锁
            broker._lockConsumer();
            broker._lockProducer();


            //模拟Server处理需求
            for (int i = 1; i <= rand()% numServer; i++)
            {
                int num = rand() % numServer;
                svg.servergroup[num].work();

                //更新锁
                broker._lockConsumer();
                broker._lockProducer();
                Sleep(400);              //Server处理时间

            }
            cout << endl;

        }
    }
```

2.5 调试分析

- 在实际实现时，不用yield无法实现一个线程下进行多线程的模拟，所以在实现时进行了相应的随机近似，通过随机请求数量进入、进行随机Server处理来队多Server进行模拟
- 进行模拟后，会产生随机数量的请求以及随机的Server进行处理，当缓冲队里列中请求过多时，自动添加Server进行任务处理
- 运行状态：

```
(706)   Tugkf   pay
(328)   Ksocg   pay
(793)   Scbqm   search
(884)   Ynfyd   browse
(738)   Hnxef   collect

NO.     4Server is working on request:          (289)collect
NO.     2Server is working on request:          (753)collect
NO.     3Server is working on request:          (424)collect
NO.     6Server is working on request:          (648)collect

                                                                Request is Submitted!
                                                                Request is Submitted!
                                                                Request is Submitted!
(706)   Tugkf   pay
(328)   Ksocg   pay
(793)   Scbqm   search
(884)   Ynfyd   browse
(738)   Hnxef   collect
(465)   Amich   collect
(178)   Pxjww   browse
(955)   Vsevs   browse

NO.     6Server is working on request:          (706)pay
NO.     6Server is working on request:          (328)pay

                                                                Request is Submitted!
                                                                Request is Submitted!
                                                                Request is Submitted!
                                                                Request is Submitted!
(793)   Scbqm   search
(884)   Ynfyd   browse
(738)   Hnxef   collect
(465)   Amich   collect
(178)   Pxjww   browse
(955)   Vsevs   browse
(640)   Pjoui   collect
(723)   Jdwwu   search
(438)   Zkgnp   pay
(915)   Ydmqb   search

NO.     2Server is working on request:          (793)search
NO.     6Server is working on request:          (884)browse

                                                                Request is Submitted!
(738)   Hnxef   collect
(465)   Amich   collect
(178)   Pxjww   browse
(955)   Vsevs   browse
(640)   Pjoui   collect
(723)   Jdwwu   search
(438)   Zkgnp   pay
(915)   Ydmqb   search
(701)   Uxgby   pay

NO.     2Server is working on request:          (738)collect
NO.     2Server is working on request:          (465)collect

                                                                Request is Submitted!
                                                                Request is Submitted!
                                                                Request is Submitted!
(178)   Pxjww   browse
```

- 其中左侧是随机的请求序列号和Server的工作情况，最右侧是客户端的回显，可以看到缓冲区起到了阻塞队列的作用

- 链队列进行删除的时候要注意在删除最后一个元素时，不要把尾节点也删除了，造成指针的越界
- 头文件互相包含时要注意
  - 可以在cpp实现文件当中进行对方的包含，防止出现互相包含且建立对方对象的错误
- 进行多实例化管理时可以设置一个类似于抽象类的管理器，便于管理所有的实例化对象（此处为Server）

3. 实验总结

- 本次实验中，进一步加深了对于队列特性的认识，进行了队列的实际应用实践
- 学习了消费者生产者模式以及缓冲队列、异步通信、错流削峰的知识
- 学习了如何让各个模块之间达到耦合最小化，加强各个模块之间的独立性和复用性