《数据结构》上机报告

2019 年 10 月 24 日

姓名：李佳庚 学号： 1852409 班级： 计算机 1 班 得分：

实验题目	队列
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。
基本要求	(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。 (9) 程序要添加适当的注释，程序的书写要采用 缩进格式 。 (10) 程序要具在一定的 健壮性，即当输入数据非法时， 程序也

<table>
<tr><td></td><td>能适当地做出反应，如 插入删除时指定的位置不对 等等。

(11) 程序要做到界面友好，在程序运行时用户可以根据相应的提示信息进行操作。

(12) 根据实验报告模板详细书写实验报告，在实验报告中给出主要算法的复杂度分析。

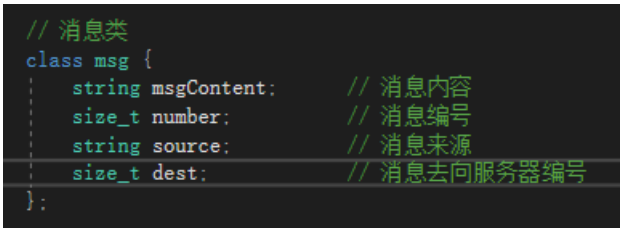
<table><tr><td>已完成基本内容（序号）：</td><td>1，2，3，4，5，6，7，8，9，10，11，12</td></tr></table></td></tr>
<tr><td>选做要求</td><td>1. 利用负载均衡技术，实现高并发。

<table><tr><td>已完成选做内容（序号）</td><td>1.</td></tr></table></td></tr>
<tr><td>数据结构设计</td><td>老师说要考虑实际情况，不要太注重代码实现，要考虑管理。
那我就稍微对这个高并发问题谈一谈自己浅薄的看法。

对于高并发而言，这个题目里面只有两台服务器，每个服务器只能一秒钟处理 1000 个请求。而读入的每秒是 3000 个请求。

一个每秒产生 3000 个请求的项目，不可能买不起第三台一秒钟处理 1000 个请求的服务器吧？这就很离谱。纵使真的买不起，也不应该像图中描写的那样，根据自身能够处理的请求个数来获取请求。这样势必导致优先级莫名其妙就是比较高的服务器经常高负荷工作。

本身利用消息队列，就已经是有负载均衡的想法了。
但是图中却搞反了。
应该是消息队列根据系统来分配消息，而不应该是系统根据自身处理能力拉取消息。
根据这个观点，我构建了一个“负载均衡类”

首先，表明消息的属性：
 

1. 一条消息，要有自己的报文。
2. 对于一天之内的消息，还应该有对应的消息编号。这个编号假设 size_t 类型可以储存的下。
3. 消息有其消息来源，每个消息都有特别的消息来源。对应消息来源，我们可以使用 http/dns 负载均衡进行合理分配。所以在不知道到底要使用哪一项技术之前，需要保留应该存储的信息。
4. Dest 代表消息的去向。这里可以用 string 类型继续表示服务器的 ip 地址，但是此处替换成了服务器的编号。</td></tr>
</table>

之后，便可以建立消息队列。

消息队列，首先是用来容纳消息的一个容器，其次，还应该可以对消息进行各种操作，包括增加，接受，弹出，发送……

```
// 消息队列类
class msgQueue {
public:

    // 消息类
    class msg { ... };

    vector<msg> pool;           // 消息池，用以存储消息
    size_t capacity;            // 消息池限定最大容量
    size_t num;                 // 消息池当前消息数目
```

1. 消息队列 msgQueue 拥有一个消息池，以此来容纳消息。
2. 消息池有对应的最大容积。
3. 以及当前存储的消息数目。

之后，利用消息队列，我们可以理解负载均衡类的构造:

```
// 负载均衡类
class LoadBalance {
public:
    // 消息队列类
    class msgQueue { ... };

private:

    msgQueue msgQUEUE;                          // 消息队列
    static const size_t serverMaxNum = 10;      // 可容纳最大量服务器个数
    size_t serverNum;                           // 当前服务器个数
    size_t serverNum_down;                      // 当前宕机服务器个数
    size_t serverNum_work;                      // 当前正在工作的服务器个数
    size_t serverNum_wait;                      // 当前正在等待的服务器个数

    string serverIP_db[serverMaxNum][13];       // 当前服务器所在IP地址
```

1. 首先，负载均衡在服务器数目有限的情况下，这里需要一个消息队列作为缓冲。
2. 使用一个静态常变量作为负载均衡类所能容纳的服务器最大数目。
3. 下面四个变量以此表示了“服务器个数”、“宕机服务器个数”、“可以工作的服务器个数”以及“正在等待的服务器个数”
4. 最后一个变量表示服务器集群的 IP 地址。

由此可以表示一个负载均衡类。

功能(函数)说明

对于 msgQueue

```
// 消息队列类
class msgQueue {
```

```
public:
    // 建立一个空队列
    explicit msgQueue(const size_t capacity_)
        : pool(vector<msgQueue::msg>()), capacity(capacity_), num(0)
    {}

    //释放队列空间，将队列销毁
    explicit msgQueue() = default;

    //将队列清空，变成空队列
    void clear();

    //判断队列是否为空
    constexpr bool isEmpty();

    //返回队列内元素个数
    size_t getNum();

    //将队头元素弹出队列
    msgQueue::msg deQueue();

    //在队列中加一个元素
    constexpr bool enQueue(const msgQueue::msg &source);

    //从队头到队尾将队列中元素取出
    void getAllMsg();
```

用如下

对于负载均衡类

```
// 负载均衡类
class LoadBalance {
private:

    int _hash_Method();                     // 哈希负载均衡
    int _polling_Method();                  // 轮询负载均衡
    int _random_Method();                   // 随机负载均衡

    bool _isDown(int serverId = 1);         // 是否宕机判断
    bool _connect(string serverIP);         // 连接服务器
    bool _execute();                        // 执行下一条指令置位

public:
    bool accept();                          // 接受客户端发送的信息
    bool send(char balMode = 'h');          // 发送客户端信息至服务器
```

1. 设置了三种负载均衡方式，分别是哈希法、随机法以及轮询法。
2. 提供了判断对应服务器是否宕机的函数。
3. 提供了连接对应服务器的函数。
4. 利用 accept 和 send 可以帮助消息队列拉取和发送消息。

开发环境	Win10 Microsoft Visual Studio Community 2017 15.9.3 Debug x86

调试分析	（运行结果截图）
	以下为实现作业中小题 1 的 queue 类代码：
	template <typename T>
	class Queue {
	
	private:
	T *queuep;
	T *queueHead;
	T *queueTail;
	size_t length;
	size_t capacity;
	
	public:
	Queue() = default;
	explicit Queue(size_t capacity_)
	: capacity(capacity_), length(0) {
	queuep = new T[capacity];
	queueHead = queueTail = queuep;
	}
	explicit Queue(const Queue<T> &source)
	: capacity(source.capacity) {
	queueHead = queueTail = queuep = new T[capacity];
	
	for (T elem : source.queuep) {
	*queueTail = elem;
	queueTail++;
	length++;
	}
	}
	explicit Queue(Queue<T> &&source) noexcept
	: queuep(source.queuep),
	queueHead(source.queueHead),
	queueTail(source.queueTail),
	capacity(source.capacity), length(source.length) {
	source.queueHead = source.queueTail = source.queuep = nullptr;
	capacity = length = 0;
	}
	
	constexpr bool isFull() noexcept {
	return (length == capacity) ? true : false;
	}
	constexpr bool isEmpty() noexcept {
	return (length == 0) ? true : false;
	}

```
constexpr size_t getLength() noexcept {
    return length;
}

bool enQueue(T data_) noexcept {
    if (length == capacity)
        return 0;
    *queueTail = data_;
    if (queueTail == queuep + capacity - 1)
        queueTail = queuep;
    else
        queueTail++;
    length++;
    return 1;
}

T deQueue() {
    if (isEmpty())
        return T();

    T tmp = *queueHead;
    if (queueHead == queuep + capacity - 1)
        queueHead = queuep;
    else
        queueHead++;
    length--;
    return tmp;

}

void display(char mode = 'p') {
    if (mode == 'p') {
        for (T *work = queuep; work != queuep + capacity; work++)
            std::cout << *work << " ";
        std::cout << std::endl;
    }
    else if (mode == 'h') {
        size_t i = 0;
        for (T *work = queueHead; i < length; i++) {
            std::cout << *work << " ";
            if (work == queuep + capacity - 1)
                work = queuep;
            else
                work++;
```

```
            }
        }
        else if (mode == 'f') {
            ;
        }
        else
            ;
    }
};
```

运行结果：

```
10
enqueue 1
enqueue 2
enqueue 3
enqueue 4
enqueue 5
enqueue 6
enqueue 7
enqueue 8
enqueue 9
enqueue 10
enqueue 11
```

```
Queue is Full
dequeue
1
dequeue
2
dequeue
3
dequeue
4
dequeue
5
dequeue
6
dequeue
7
dequeue
8
dequeue
9
dequeue
10
```

```
dequeue
Queue is Empty
```

心得体会	（对整个实验过程做出总结，对重要的算法做出性能分析。） 这道题对我来说很有趣。 虽然不会写具体实现的代码，比如如何构造一个可以和服务器连接的消息队列类，（比如 dequeue 之后的处理，以及 enqueue 之前的处理）但是在网上浏览了大量与负载均衡技术相关的文章之后，我好像对那些电商如何应对购物节请求的爆发有了点了解。 在阅读之后，我比较肤浅地理解了轮询、最小连接以及随机哈希法在处理负载均衡时的作用。也知道了 dns、数据链路、ip、http 层的负载均衡手段。 这个作业，不仅仅锻炼了我对队列的理解，还让我第一次接触了解决高并发的一个手段。我觉得挺好的，今天晚上还是有点收获。 虽然仅仅是了解，还不如不会，但是至少被科普了一些东西吧。