其他问题

[1. UML类图 1](#_101)

[2. 算法问题 1](#_102)

[2.1. 求解平方根 1](#_103)

[2.2. 背包九讲 1](#_104)

[2.2.1. 问题背景 1](#_106)

[2.2.2. 初始化细节 1](#_107)

[2.2.3. 0/1背包 1](#_105)

[2.2.4. 完全背包 2](#_110)

[2.2.5. 多重背包 3](#_111)

[2.3. 股票问题 3](#_126)

[3. 数据库 4](#_127)

[3.1. 关系型数据库与非关系型数据库的差异 4](#_128)

[4. 负载均衡 4](#_129)

[4.1. 限流算法 4](#_130)

[4.1.1. 漏桶 4](#_131)

[4.1.2. 令牌桶 4](#_132)

[4.2. 企业级负载均衡 4](#_135)

[4.2.1. DR三角传输 4](#_136)

[4.2.2. NAT(DNAT) 5](#_137)

[4.2.3. Tunnel 5](#_138)

[4.2.4. FULLNAT(SNAT+DNAT) 5](#_139)

[4.2.5. 美团MGW 5](#_140)

[5. 消息中间件 10](#_147)

[5.1. 基本流程 10](#_148)

[5.2. 底层存储 10](#_149)

[5.2.1. Commit Log 10](#_150)

[5.2.2. Consume Queue 10](#_151)

[5.2.3. 落盘 10](#_152)

[6. 海量数据题 10](#_113)

[6.1. 海量日志数据，提取出某日访问百度次数最多的IP 10](#_114)

[6.2. 1G大小的文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小为1M，返回频数最高的100个词。 11](#_115)

[6.3. 在2.5亿整数中找出不重复的整数，内存不足以容纳这2.5亿个整数 11](#_116)

[7. 数据结构 11](#_120)

[7.1. 平衡树 11](#_117)

[7.1.1. 红黑树 11](#_118)

[7.1.2. AVL树 11](#_119)

[7.2. 排序 11](#_121)

[7.2.1. 快排 12](#_122)

[8. 设计模式 12](#_123)

[8.1. 简单工厂、工厂方法和抽象方法 12](#_124)

[8.2. 原则 12](#_125)

[9. 趣味题 12](#_133)

[9.1. 微信红包 13](#_134)

# UML类图

UML类图中类之间的关系包括：

依赖：A类是B类中某方法的局部变量；A类是B类方法的其中一个参数；A类向B类发送消息，从而影响B类发生变化。

泛化：继承关系

实现：实现接口。

关联：类之间的联系。通常是将一个类的对象作为另一个类的属性。

聚合：表示整体与部分的关系。成员是整体的一部分，但成员可以脱离整体对象独立存在。

组合：整体与部分，但是整体与部分不能分开。实心菱形箭头。

# 算法问题

## 求解平方根

1. 牛顿迭代法

根据公式Xn+1 = Xn - f(Xn)/f'(Xn)

在平方根求解中可以将f(x) = x^2 - a， 所以Xn+1 = (Xn + a/Xn)/2，不断迭代，直到一次迭代前后差值小于1e6.

## 背包九讲

### 问题背景

有N件物品和一个容量V的背包。放入第i件物品耗费的费用是Ci，得到的价值是Wi。求解将哪些物品装入背包可使价值总和最大。

### 初始化细节

#### 恰好装满背包

在初始化时除了 F[0] 为 0，其它 F[1..V ] 均设为 −∞，这样就可以保证最终得到的 F[V ] 是一种恰好装满背包的最优解。

#### 不要求装满背包

F[0, 0...V]都置为0就可以。

### 0/1背包

F[i, v]表示前i件物品放入一个容量为v的背包可以获得的最大价值。

状态转移方程为:

**F[i, v] = max{F[i-1, v], F[i-1, v-Ci] + Wi}**

比如对于3件物品，消耗容量分别为1, 2, 3；价值分别为3, 4, 4， 给定的背包容量为5

构建的dp如下：

0 0 0 0 0 0

0 3 3 3 3 3

0 3 4 7 7 7

0 3 4 7 7 8

时间复杂度和空间复杂度均为O(VN)，空间复杂度可继续优化到O(V),其实就是滚动数组，因为当前行的处理结果只与上一行的值有关（在优化时需要从右向左处理）

这边还有小优化点，也就是V->C[i]，都可以进行比较，C[i] -> 1都直接原数据。

### 完全背包

完全背包的不同之处在于每种物品都有无限件可用。

状态方程为

**F[i, v] = max{F[i-1, v-kCi] + kWi | 0 <= kCi <= v}**

此时求解每个状态的时间已经不是常数了，求解F[i, v]的时间是O(v/Ci)，总的复杂度可以认为是O(NV ΣV/Ci),是比较大的。

1. 完全背包问题有一个很简单有效的优化，是这样的：若两件物品 i、j 满足 Ci ≤ Cj 且 Wi ≥ Wj，则将可以将物品 j 直接去掉，不用考虑。

2. 更高效的转化方法是：把第 i 种物品拆成费用为 Ci2 k、价值为 Wi2 k 的若干件物 品，其中 k 取遍满足 Ci2 k ≤ V 的非负整数。 这是二进制的思想。因为，不管最优策略选几件第 i 种物品，其件数写成二进制后， 总可以表示成若干个 2 k 件物品的和。这样一来就把每种物品拆成 O(log ⌊V /Ci⌋) 件物 品，是一个很大的改进。其实就是压缩dp的行，压缩物品的数量，转化为0/1背包问题

3. **O(VN)的算法（最棒）**

F[0..V ] ←0

for i ← 1 to N

for v ← Ci to V

F[v] ← max(F[v], F[v − Ci ] + Wi)

这思路更好，建立在之前的基础上。着重注意的是v从Ci到V递增，然后求相应状态。

### 多重背包

有 N 种物品和一个容量为 V 的背包。第 i 种物品最多有 Mi 件可用，每件耗费的 空间是 Ci，价值是 Wi。求解将哪些物品装入背包可使这些物品的耗费的空间总和不超 过背包容量，且价值总和最大。

方法是：将第 i 种物品分成若干件 01 背包中的物品，其中每件物品有一个系 数。这件物品的费用和价值均是原来的费用和价值乘以这个系数。令这些系数分别为 1, 2, 2^2 . . . 2^k−1 , Mi − 2^k + 1，且 k 是满足 Mi − 2^k + 1 > 0 的最大整数。例如，如果 Mi 为 13，则相应的 k = 3，这种最多取 13 件的物品应被分成系数分别为 1, 2, 4, 6 的四件物品。

#### 每种有若干件的物品能否填满给定容量的背包

下面介绍一种实现较为简单的 O(VN) 复杂度解多重背包问题的算法。它的基本 思想是这样的：设 F[i, j] 表示“用了前i种物品填满容量为j的背包后，最多还剩下几个第i种物品可用”，如果F[i, j] = −1 则说明这种状态不可行，若可行应满足 0 ≤ F[i, j] ≤ Mi。 递推求 F[i, j] 的伪代码如下：

F[0, 1 . . . V ] ← −1

F[0, 0] ← 0

for i ← 1 to N

for j ← 0 to V

if F[i − 1][j] ≥ 0

F[i][j] = Mi

else

F[i][j] = −1

for j ← 0 to V − Ci

if F[i][j] > 0

F[i][j + Ci ] ← max{F[i][j + Ci ], F[i][j] − 1}

最终 F[N][0 . . . V ] 便是多重背包可行性问题的答案。

## 股票问题

**1. 状态描述**

三维状态dp[i][k][0] or dp[i][k][1]

dp[i][k][0]：表示在第i天，此时未持有股票，并已经进行了k次交易。

dp[i][k][1]：表示在第i天，此时持有股票，并已经进行了k次交易。

**2. 状态转移方程**

dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])

表示啥都不干 表示在第i天卖出手里的股票

dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

表示啥也不干 表示在第i天买入股票

注意：在买入股票时，交易数k + 1

# 数据库

## 关系型数据库与非关系型数据库的差异

两者的主要差异在于数据存储的方式。关系型数据天然就是表格式的，因此存储在数据表的行和列中。数据表可以彼此关联协作存储，也很容易提取数据。与其相反，非关系型数据不适合存储在数据表的行和列中。

# 负载均衡

## 限流算法

### 漏桶

在网络中，漏桶算法可以控制端口的流量输出速率，平滑网络上的突发流量，实现流量整形，从而为网络提供一个稳定的流量。

把请求比作是水，水来了都先放进桶里，并以限定的速度出水，当水来得过猛而出水不够快时就会导致水直接溢出，即拒绝服务。

### 令牌桶

令牌桶算法的原理是系统会以一个恒定的速度往桶里放入令牌，而如果请求需要被处理，则需要先从桶里获取一个令牌，当桶里没有令牌可取时，则拒绝服务。从原理上看，令牌桶算法和漏桶算法是相反的，一个“进水”，一个是“漏水”。

## 企业级负载均衡

### DR三角传输

1. 通过修改数据包的目的Mac地址使流量经过二层转发到达应用服务器。

2. 应用服务器处理完成后直接将响应发送给客户端。

**优点：**

1. 应用直接将应答发送给客户端，性能好

**缺点：**

1. 应用服务器必须和负载均衡服务器位于同一个二层局域网。

2. 应有服务器必须配置VIP

### NAT(DNAT)

NAT通过修改数据包的目的IP，从而让流量到达应用服务器。

**优点：**

1. 应用服务器无需做任何配置

**缺点：**

1. 负载均衡必须以网关形式存在

### Tunnel

Tunnel类似DR模式

**优点：**

1. 和DR一样，应用直接将应答发送给客户端，性能好

**缺点：**

1. 对应用服务器要求高，需要支持tunnel

2. 应用服务器需要配置vip。

### FULLNAT(SNAT+DNAT)

FULLNAT模式是在NAT模式的基础上做一次源地址转换（即SNAT），做SNAT的好处是可以让应答流量经过正常的三层路由回到负载均衡上，这样负载均衡就不要以网关形式存在于网络中，对网络环境要求较低。

**优点：**

1. 应用服务器无需做配置

2. 对网络环境要求比较低

**缺点：**

1. 丢失client ip

### 美团MGW

下面详细介绍一下FULLNAT模式。首先负载均衡上需要存在一个localip池，在做SNAT时的源IP就是从localip池中选择的。当客户端流量到达负载均衡设备以后，负载均衡会根据调度策略在应用服务器池中选择一个应用服务器，然后将数据包的目的IP改为应用服务器的IP。同时从localip池中选择一个localip将数据包的源IP改为localip，这样应用服务器在应答时，目的IP是localip，而localip是真实存在于负载均衡上的IP地址，因此可以经过正常的三层路由到达负载均衡。由于FULLNAT比NAT模式多做了一次SNAT，并且SNAT中有选端口的操作，因此其性能要逊色于NAT模式，但是由于其较强的网络环境适应性，我们选择了FULLNAT作为MGW的转发模式。

#### 中断问题及协议栈路径性能过长

中断是影响LVS性能最重要的一个因素，假如我们一秒需要处理600万的数据包，每6个数据包产生一个硬件中断的话，那一秒就会产生100万个硬件中断，每一次产生硬件中断都会打断正在进行密集计算的负载均衡程序，中间产生大量的cache miss，对性能的影响异常大。

同时由于LVS是基于内核netfilter开发的一个应用程序，而netfilter是运行在内核协议栈的一个钩子框架。这就意味着当数据包到达LVS时，已经经过了一段很长的协议栈处理，但是这段处理对于LVS来说都不是必需的，这也造成了一部分不必要的性能损耗。

针对这两个问题，解决方法是使用轮询模式的驱动以及做kernel bypass，而DPDK提供的用户态PMD驱动恰好可以解决这两个问题。DPDK在设计时使用了大量硬件相关特性比如numa、 memory channel、 DDIO等，对性能优化非常大，同时提供了比较多网络方面的库，可以大大减小开发难度，提高开发效率。因此选择DPDK作为MGW的开发框架。

#### session锁

首先介绍一下RSS（Receive Side Scaling），RSS是一个通过数据包的元组信息将数据包散列到不同网卡队列的功能，这时候不同的CPU再去对应的网卡队列读取数据进行处理，就可以充分利用CPU资源。之前介绍MGW使用FULLNAT的模式，FULLNAT会将数据包的元组信息全部改变，这样同一个连接，请求和应答方向的数据包有可能会被RSS散列到不同的网卡队列中，在不同的网卡队列也就意味着在被不同的CPU进行处理，这时候在访问session结构的时候就需要对这个结构进行加锁保护。

解决这个问题的方法有两种，一种就是在做SNAT选端口的时候，通过选择一个端口lport0让RSS(cip0, cport0, vip0, vport0) = RSS(dip0, dport0, lip0, lport0)相等；另外一种方法就是我们为每个CPU分配一个localip，在做SNAT选IP的时候，不同的CPU选择自己的localip，等应答回来以后，再通过lip和CPU的映射关系，将指定目的IP的数据包送到指定队列上。

由于第二种方法恰好可以被网卡的flow director特性支持，因此我们选择了第二种方法来去掉session结构的锁。

￼

flow director可以根据一定策略将指定的数据包送到指定网卡队列，其在网卡中的优先级要比RSS高，因此我们在做初始化的时候就为每个CPU分配一个localip，比如为cpu0分配lip0，为cpu1分配lip1，为cpu2分配lip2，为cpu3分配lip3。 当一个请求包（cip0, cport0, vip0, vport0）到达负载均衡后，被RSS散列到了队列0上，这时这个包被cpu0处理。cpu0在对其做fullnat时，选择cpu0自己的localip lip0，然后将数据包（lip0, lport0, dip0, dport0）发到应用服务器，在应用服务器应答后，应答数据包（dip0, dport0, lip0, lport0）被发到了负载均衡服务器。此时我们就可以在flow director下一条将目的IP为lip0的数据包送到队列0的规则，这样应答数据包就会被送到队列0让cpu0处理。这时候CPU在对同一个连接两个方向的数据包进行处理的时候就是完全串行的一个操作，也就不要再对session结构进行加锁保护了

#### 上下文切换

在设计时，希望控制平面与数据平面完全分离，数据平面专心做自己的处理，不被任事件打断。因此将CPU分成两组，一组用作数据平面一组用做控制平面。同时，对数据平面的CPU进行CPU隔离，这样控制平面的进程就不会调度到数据平面的这组CPU上面了；对数据平面的线程进行CPU绑定，这样就可以让每个数据线程独占一个CPU。 其他的控制平面的程序比如Linux kernel、 SSH等都跑在控制平面的这组CPU上。

下面从MGW集群、MGW单机以及应用服务器层这三个层介绍MGW如何在每一层实现高可靠。

#### 故障切换

在故障切换的问题上，我们希望在机器故障以后，交换机可以立刻将流量切到其他机器上，因为流量不切走，意味着到达这台机器流量会被全部丢掉，产生大量丢包。经过调研测试发现，当交换机侧全部使用物理接口并且服务器侧对接口进行断电时，交换机会瞬间将流量切换到其他机器上。通过一个100ms发两个包的测试（客户端和服务端各发一个），这种操作方法是0丢包的。

由于故障切换主要依赖于交换机的感知，当服务器上出现一些异常，交换机感知不到时，交换机就无法进行故障切换操作，因此需要一个健康自检程序，每半秒进行一次健康自检，当发现服务器存在异常时就对服务器执行网口断电操作，从而让流量立刻切走。

故障切换主要依赖于网口断电操作并且网卡驱动是跑在主程序里面的，当主程序挂掉以后，就无法再对网口执行断电操作了，因此为了解决这个问题，主进程会捕获异常信号，当发现异常时就对网卡进行断电操作，在断电操作结束以后再继续将信号发给系统进行处理。

经过以上设计，MGW可以做到升级操作0丢包，主程序故障0丢包，其他异常（网线等）会有一个最长500ms的丢包，因为这种异常需要靠自检程序去检测，而自检程序的周期是500ms。

#### 故障恢复与扩容

无论是在进行故障恢复还是扩容操作，都会导致集群节点数量发生变化，这样也就会导致流量路径发生变化。当变化的流量到达集群中原有的节点时，因为原有节点都维护着一个全局的session表，因此这些流量是可以被正常转发的；但是如果流量到达了新机器上，这个机器是没有全局session表的，那么这部分流量就会全部被丢弃。为了解决这个问题，MGW在上线以后会经历一个预上线的中间状态，在这个状态上，MGW不会让交换机感知到自己上线了，这样交换机也就不会把流量切过来。首先MGW会对集群中其他节点发送一个批量同步的请求，其他节点收到请求以后会将自己的session全量的同步到新上线的节点上，新上线节点在收到全部session以后才会让交换机感知到自己上线，这时交换机再将流量切过来就可以正常被转发出去了。

在这个过程中主要存在两点问题。 第一个问题是，由于集群中并没有一个主控节点来维护一个全局的状态，如果request报丢失或者session同步的数据丢失的话，那新上线节点就没办法维护一个全局的session状态。但是考虑到所有节点都维护着一个全局的session表，因此所有节点拥有的session数量都是相同的，那么就可以在所有节点每次做完批量同步以后发送一个finish消息，finish消息中带着自己拥有的session数量。当新上线节点收到finish消息以后，便会以自己的session数量与finish中的数量做对比。当达到数量要求以后，新上线节点就控制自己进行上线操作。否则在等待一定的超时时间以后，重新进行一次批量同步操作，直到达到要求为止。

另外一个问题是在进行批量同步操作时，如果出现了新建连接，那么新建连接就不会通过批量同步同步到新上线的机器上。如果新建连接特别多，就会导致新上线机器一直达不到要求。因此，需要保证处于预上线状态的机器能接收到增量同步数据，因为新建连接可以通过增量同步同步出来。通过增量同步和批量同步就可以保证新上线机器可以最终获得一个全局的session表。

#### 一致性源IP hash调度器

源IP Hash调度器主要是保证相同的客户端的连接被调度到相同应用服务器上，也就是说建立一个客户端与应用服务器一对一的映射关系。普通的源IP Hash调度器在应用服务器发生变化以后会导致映射关系发生改变，会对业务造成影响。

因此我们开发了一致性源IP Hash调度器，保证在应用服务器集群发生变化时，只有发生变化的应用服务器与客户端的映射关系发生改变，其他都是不变的。

为了保证流量的均衡，首先在hash环上分配固定数量的虚拟节点，然后将虚拟机节点均衡的重分布到物理节点上，重分布算法需要保证两点：

1. 在物理节点发生变化时，只有少数虚拟节点映射关系发生变化，也就是要保证一致性Hash的基本原则。

2. 因为MGW是以集群的形式存在的，当多个应用服务器发生上线下线操作时，反馈到不同的MGW节点上就有可能会出现顺序不一致的问题，因此无论不同的MGW节点产生何种应用服务器上下线顺序，都需要保证最终的映射关系一致，因为如果不一致就导致相同客户端的连接会被不同的MGW节点调度到不同的应用服务器上，也就违背了源IP Hash调度器的原则。

# 消息中间件

## 基本流程

1. 消息生成者发送消息MQ收到消息

2. 将消息进行持久化，在存储中新增一条记录

3. 返回ACK给生产者

4. MQ push 消息给对应的消费者，然后等待消费者返回ACK

5. 如果消息消费者在指定时间内成功返回ACK，那么MQ认为消息消费成功，在存储中删除消息，；如果MQ在指定时间内没有收到ACK，则认为消息消费失败，会尝试重新push消息,重复执行4、5MQ删除消息

## 底层存储

### Commit Log

一个文件集合，每个文件1G大小，存储满后存下一个，为了讨论方便可以把它当做一个文件，所有消息内容全部持久化到这个文件中。

### Consume Queue

一个Topic可以有很多个Consume Queue，每个文件代表一个逻辑队列，这里存放消息在Commit Log的偏移量以及大小和tag属性。

### 落盘

#### 同步落盘

在消息真正落盘后，才返回成功给Producer，只要磁盘阵列RAID5完好，消息就不会丢。一般只用于金融场景，这种方式不是本文讨论的重点，因为没有充分利用Page Cache的特点。

#### 异步落盘

读写文件充分利用了Page Cache，即写入Page Cache就返回成功给Producer。之后的内容全部以异步刷盘方式来讨论。

# 海量数据题

## 海量日志数据，提取出某日访问百度次数最多的IP

1. 对IP进行映射处理（hash），比如mod1000，分成1000个文件。

2. 分别找出每个文件访问次数最高的ip。

3. 然后再对比这1000个“最大”的IP

## 1G大小的文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小为1M，返回频数最高的100个词。

方案：

1. 顺序读文件中，对于每个词x，取hash(x)%5000，然后按照该值存到5000个小文件（记为x0,x1,...x4999）中。这样每个文件大概是200k左右。

2. 如果其中的有的文件超过了1M大小，还可以按照类似的方法继续往下分，直到分解得到的小文件的大小都不超过1M。

3. 对每个小文件，统计每个文件中出现的词以及相应的频率（可以采用trie树/hash\_map等），并取出出现频率最大的100个词（可以用含100个结点的最小堆），并把100个词及相应的频率存入文件，这样又得到了5000个文件。下一步就是把这5000个文件进行归并（类似与归并排序）的过程了。

## 在2.5亿整数中找出不重复的整数，内存不足以容纳这2.5亿个整数

所有整数的个数为2^32，每个用2bit来表示，00表示不存在，01表示存在1次，10表示存在多次。

这样需要的内存容量为2^32\*2 = 1GB

# 数据结构

## 平衡树

### 红黑树

红黑树不追求完全平衡，是功能、性能、空间开销的折中结果。

读取略逊于AVL，维护强于AVL，空间开销与AVL类似。

### AVL树

追求完全平衡，在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多。

AVL在插入和删除时效率低，查询效率高。

## 排序

### 快排

# 设计模式

## 简单工厂、工厂方法和抽象方法

**1. 简单工厂**

通过静态方法实现，通过接收的参数的不同来返回不同的对象实例（属于同一接口）。

**2. 工厂方法**

每种产品对应一个工厂类（每个工厂生产一类产品），通过不同的工厂实例来创建不同的产品实例。

**3. 抽象工厂**

每一个工厂生产一整个产品组，比如汽车公司要生产轿车、货车和客车。

## 原则

**1. 单一职责原则**

一个类只负责一项职责。

**2. 里氏替换原则**

所有引用基类的地方必须能够透明地使用其子类对象。

**3. 依赖倒置原则**

高层模块不应该依赖底层模块，二者都应该依赖抽象。

抽象不应该依赖细节；细节依赖于抽象。

**4. 接口隔离原则**

客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。

**5. 迪米特原则**

一个类应该和其他类保持最少的了解。（降低类之间耦合）

**6. 开闭原则**

一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是通过修改已有的代码来实现变化。

# 趣味题

## 微信红包

public static double getRandomMoney(LeftMoneyPackage \_leftMoneyPackage) {

// remainSize 剩余的红包数量

// remainMoney 剩余的钱

if (\_leftMoneyPackage.remainSize == 1) {

\_leftMoneyPackage.remainSize--;

return (double) Math.round(\_leftMoneyPackage.remainMoney \* 100) / 100;

}

Random r = new Random();

double min = 0.01; //

double max = \_leftMoneyPackage.remainMoney / \_leftMoneyPackage.remainSize \* 2;

double money = r.nextDouble() \* max;

money = money min ? 0.01: money;

money = Math.floor(money \* 100) / 100;

\_leftMoneyPackage.remainSize--;

\_leftMoneyPackage.remainMoney -= money;

return money;

}