

问题集锦

因为每次面试都要刷一遍面试题，对于一些平时少用的点也比较容易忘记，过后再复习一遍，比较浪费时间。所以这里总结一下，方便以后查看。

当前版本： C++

更新时间： 2020-03-10

第一部分

暂缺，目前一般都是在线笔试，参考 [编程部分](#)

第二部分

Tips：问答部分多准备一些实现原理性的东西

C++虚函数实现原理

C++中的虚函数（Virtual Function）是用来实现 动态多态性（多态）的，指的是当基类指针调用派生类中的成员函数。如果基类指针指向不同的派生类，则它调用的同一个函数就可以实现不同的逻辑，这种机制可以让基类指针有“多种形态”，它的实现依赖于 虚函数表（Virtual Table）。虚函数表是指在每个包含虚函数的类中都存在一个函数地址的数组。

虚函数形式： 使用 `virtual` 关键字修饰的成员函数

虚函数表的地址总是在对象实例的最前面的位置，其后依次是对象实例的成员。所以我们可以通过对象实例的地址（首地址）得到虚函数表的地址。并进行调用。

假如有如下定义： `Base b;` 则在32bit内存寻址的操作系统下（int==32bit）：

虚函数表地址 `vptr` 的值就是： `(int*)*(int*)&b`

第一个虚函数 `vfunc1` 的地址就是： `*(int*)*(int*)&b`

第二个虚函数 `vfunc2` 的地址就是： `*((int*)*(int*)&b + 1)`

虚函数在多态上发挥的威力要借助继承才能实现，继承分为：单继承、多继承。

单继承

- 子类 未覆盖 父类虚函数表

- 虚函数按照其声明顺序放在表中
- 父类的虚函数在子类的虚函数之前

- 子类 覆盖 父类虚函数表

- 覆盖的 `f()` 函数被放在了虚表中原来父类虚函数的位置

2. 没有被覆盖的函数依旧

多继承

- 子类 未覆盖 父类虚函数表

1. 每个父类都有自己的虚表

2. 子类的成员函数被放在了第一个父类的表中。 (所谓的第一个父类是按照声明的顺序来判断的)

这样做就是为了解决不同的父类类型的指针指向同一个子类实例，而能够调用到实际的函数。

- 子类 覆盖 父类虚函数表

1. 所有父类虚表中 `f()` 均被替换成子类的函数指针

2. 其他同 未覆盖

这样，就可以使用任意静态类型的父类来指向子类，并调用父类的 `f()`。

C++编码，运算符重载

为什么上一章节多态里面为什么C++没有设置 返回值重载 呢？

因为，C++调用一个函数是可以忽略其返回值的，在这种情况下编译器就无法根据返回值来确定调用哪一个函数。所以，重载不能用返回值类型来区别。

运算符重载，其实就是将运算符看做特殊名称的函数。例如：

```
Box operator+(const Box&);
```

有一个不可重载的运算符列表：

- `.`：成员访问运算符
- `.* , ->*`：成员指针访问运算符
- `::`：域运算符
- `sizeof`：长度运算符
- `?::`：条件运算符
- `#`：预处理运算符

分布式系统雪崩

- 定义

简单来说，由于服务提供者A不可用，到时服务调用者B对A的请求阻塞，没有相关的机制通知或解决请求阻塞，导致在服务调用者B对A请求的阻塞越来越多，阻塞请求变多并且不断对A进行请求重试导致服务调用者B所在的系统的资源会被耗尽，而服务调用者B所在的系统可能并不会只对A的调用，还有存在对其他服务提供者的调用，因为调用A把系统资源已经耗尽了，导致也无法处理对非A请求，而且这种不可用可能沿请求调用链向上传递，比如说服务调用者C会调用B的服务，因为B所在的系统不可用，导致C也不可用，这样级联导致阻塞请求越来越多，表现为多个系统都不可用了，这种现象被“雪崩效应”。

- 产生原因

复杂分布式架构的应用程序有许多依赖，其中每一个在某些时候都会不可避免的发生失败。如果这个主应用没有从哪些外部失败隔离，那么就会有被拖垮的风险。

服务提供者不可用-->服务调用者请求重试-->服务调用者所在系统资源耗尽-->服务调用者不可用一个数字我们应该关注一下，可能更加有助于我们理解服务器雪崩。

例如，1个应用依赖30个服务，每个服务有99.99%可用，那么预期：

$99.99^2 = 99.7\%$ 的正常运行时间

10亿次请求中有0.3% = 3,000,000次失败

2小时停机时间/月，即使所有的依赖都有很好的正常运行时间

服务提供者不可用

1. 硬件故障，如服务器宕机，机房断电
2. 程序bug，如JVM长时间FullGC
3. 缓存击穿，一般发生在缓存应用重启，所有缓存被清空，以及短时间大量缓存失效时。大量的缓存请求不命中，使请求直击后端，总成服务提供者超负荷运行，引起服务不可用
4. 流量激增，如异常流量，失败重试加大流量等

服务调用者重试

1. 用户重试，不断刷新页面甚至提交表单
2. 代码逻辑重试，在代码中加入异常情况下，请求重试的功能，这在因为网络抖动导致请求超时的情况下是很有用的。但对服务提供者本身不可用情况下，会加大对服务提供者的请求流量负担。

服务调用者不可用

1. 同步等待造成的资源耗尽，当调用者使用同步调用时，会产生大量的等待线程占用系统资源。一旦线程资源被耗尽，服务调用者提供的服务也将处于不可用状态。
- 如何防止

应对策略从造成雪崩的原因出发，提供不同的原因下的解决方案：

1. 硬件故障：多机房容灾、异地多活等
2. 程序bug：修改程序bug、及时释放资源等
3. 缓存穿透：`缓存预加载`、`缓存异步加载`等
4. 流量激增：服务`自动扩容`、流量控制（`限流`、`关闭重试`）等

限流分为：网关限流（比如iptables可以控制并发数）、用户交互限流

网关限流还能通过Nginx+Lua的网关进行流量控制，这里需要了解一下[OpenResty](#)。

5. 同步等待：`资源隔离`（主要是说对调用服务的线程池进行隔离）、MQ解耦、不可用服务调用快速失败等。资源隔离通常指不同服务调用采用不同的线程池；不可用服务调用快速失败一般通过`熔断`和`超时`两种机制的结合。如使用Hystric做故障隔离，熔断器机制等可以解决依赖服务不可用的问题。

通过实践发现，线程同步等待是最常见引发的雪崩效应的场景。

超时策略

如果是一个服务会被系统中的其他部分频繁调用，一个部分的故障可能会导致级联故障。例如，调用服务的操作可以配置为执行超时，**如果服务在指定时间内响应，将回复一个失败消息**。

然而，这种策略可能会导致许多并发请求到同一个操作被阻塞，知道超时期限届满。这些阻塞的请求可能会存储关键的系统资源，如内存、线程、数据库连接等。因此，这些资源可能会枯竭，导致需要使用相同的资源系统故障。在这种情况下，它将是优选的操作立即失败。设置较短的超时可能有助于解决这个问题，但是一个操作请求从发出到收到成功或者失败的消息需要的时间是不确定的。

熔断策略

熔断器的模式使用断路器来检测故障是否已得到解决，防止请求反复尝试执行一个可能会失败的操作，从而减少等待纠正故障的时间，相对与超时策略更加灵活。

举例一个熔断功能原理：

Hystrix提供的熔断器就有类似功能，**当在一定时间段内服务调用方调用服务提供方的服务的次数达到设定的阈值，并且出错的次数也达到设置的出错阈值，就会进行服务降级，让服务调用方之间执行本地设置的降级策略，而不再发起远程调用（比较常见的策略就是直通）**。但是Hystrix提供的熔断器具有自我反馈，自我恢复的功能，Hystrix会根据调用接口的情况，让熔断器在**closed, open, half-open**三种状态之间自动切换。

open：状态说明打开熔断，也局势服务调用方执行本地降级策略，不进行远程调用。

closed：状态说明关闭了熔断，这时候服务调用方直接发起远程调用。

half-open：状态，则是一个中间状态，当熔断器处于这种状态的时候，直接发起远程调用。

TCP拥塞控制算法

TCP协议通过维护一个拥塞窗口来进行拥塞控制，拥塞控制的原则是：只要网络中没有出现拥塞，拥塞窗口的值就可以再增大一些，以便把更多的数据包发送出去；但只要网络出现拥塞，拥塞窗口的值就应该减小一些，以减少注入到网络中的数据包数。

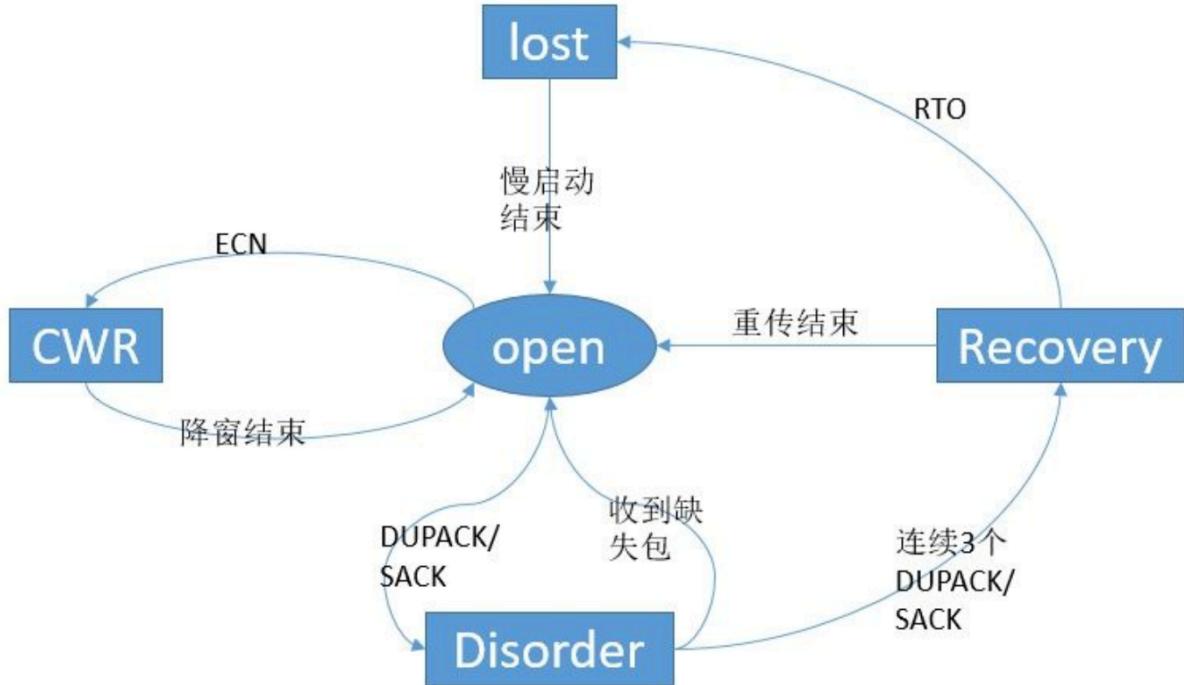
若出现拥塞而不进行控制，整个网络的吞吐量将随输入负荷的增大而下降。

当然，我们有必要了解一下TCP协议两个比较重要的控制算法，一个是流量控制，另一个就是阻塞控制。

TCP协议通过滑动窗口来进行流量控制，它是控制发送方的发送速度从而使接受者来得及接收并处理。而拥塞控制是作用于网络，它是防止过多的包被发送到网络中，避免出现网络负载过大，网络拥塞的情况。

拥塞算法需要掌握其状态机和四种算法。拥塞控制状态机的状态有五种，分别是Open, Disorder, CWR, Recovery和Loss状态。四个算法为慢启动，拥塞避免，拥塞发生时算法和快速恢复。

Congestion Control State Machine (拥塞控制状态机)



TCP拥塞控制状态机

知乎 @张天

1. Open状态

Open状态是拥塞控制状态机的默认状态。这种状态下，当ACK到达时，发送方根据拥塞窗口 cwnd (Congestion Window) 是小于还是大于慢启动阈值ssthresh (slow start threshold)，来按照慢启动或者拥塞避免算法来调整拥塞窗口。

2. Disorder状态

当发送方检测DACK (重复确认) 或者SACK (选择性确认) 时，状态机将转变为Disorder状态。在此状态下，发送方遵循飞行 (in-flight) 包守恒原则，即一个新包只有在一个老包离开网络后才发送，也就是发送方收到老包的ACK后，才会发送一个新包。

3. CWR状态

发送方接收到一个拥塞通知时，并不会立刻减少拥塞窗口cwnd，而是每收到两个ACK就减少一个段，直到窗口的大小减半为止。当cwnd正在减小并且网络中没有重传包时，这个状态就叫CWR (Congestion Window Reduced, 拥塞窗口减少) 状态。CWR状态可以转变成Recovery或者Loss状态。

4. Recovery状态

当发送方接收到足够 (推荐为3个) 的DACK (重复确认) 后，进入该状态。该状态下，拥塞窗口cnwd 每收到两个ACK就减少一个段 (segment) 直到cwnd等于慢启动值ssthresh，也就是刚进入Recover状态时cwnd的一半大小。发送方保持Recovery状态直到所有进入Recovery状态时正在发送的数据段都成功地被确认，然后发送方恢复成Open状态，重传超市有可能终端Recovery状态，进入Loss状态。

5. Loss状态

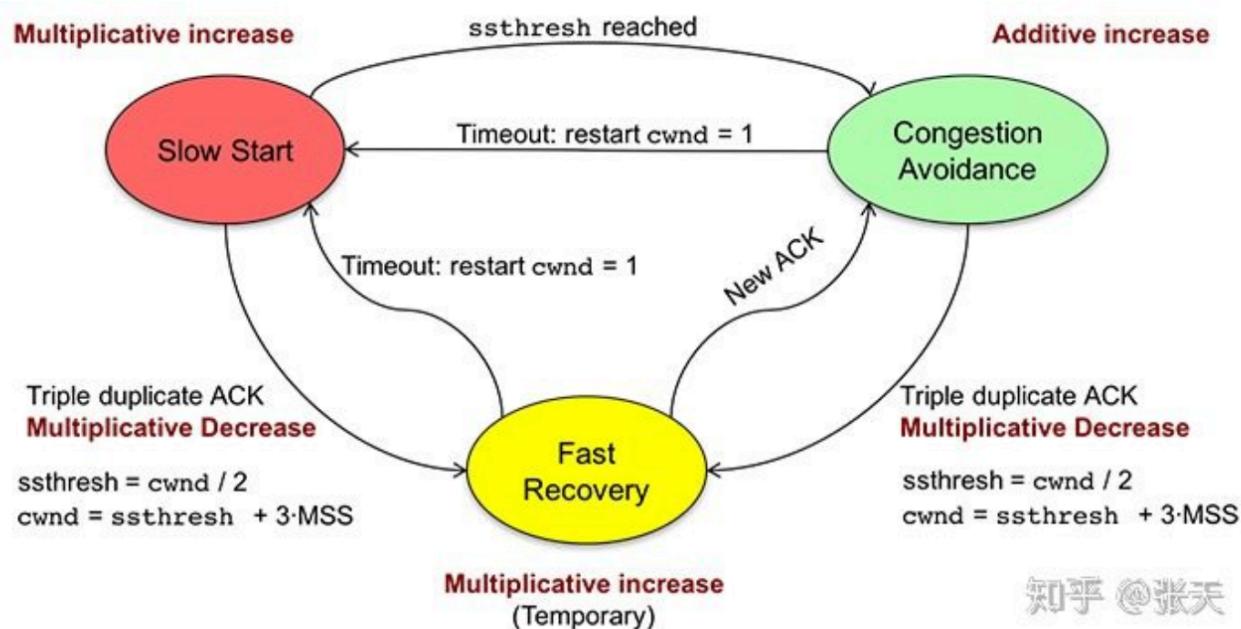
当一个RTO (重传超时时间) 到期后，发送方进入Loss状态。所有正在发送的数据标记为丢失，拥塞窗口cwnd设置为一个段 (segment)，发送方再次以慢启动算法增大拥塞窗口cwnd。

Loss和Recovery状态的区别是：Loss状态下，拥塞窗口在发送方设置为一个段后增大，而Recovery状态下，拥塞窗口只能被减小。Loss状态不能被其他的状态中断，因此，发送方只有在所有的Loss开始时正在传输的数据都得到成功确认后，才能退到Open状态。

四大算法

拥塞控制主要是四大算法：

1. 慢启动
2. 拥塞避免
3. 拥塞发生
4. 快速回复



1. 慢启动算法 (Slow Start)

所谓慢启动，也就是TCP连接刚建立，一点一点地提速，试探一下网络的承受能力，以免直接扰乱了网络通道的秩序。

2. 拥塞避免算法 (Congestion Avoidance)

当拥塞窗口大小 $cwnd \geq$ 慢启动阈值 $ssthresh$ 后，就进入拥塞避免算法。算法简述如下：

- 收到一个ACK，则 $cwnd = cwnd + 1/cwnd$
- 每当过了一个往返延迟时间RTT， $cwnd$ 大小+1

过了慢启动阈值后，拥塞避免算法可以避免窗口增长过快导致窗口拥塞，而是缓慢的增加调整到网络的最佳值。

3. 拥塞状态时的算法

一般来说，TCP拥塞控制默认认为网络丢包是由于网络拥塞导致的，所以一般的TCP拥塞控制算法以丢包为网络进入拥塞状态的信号。对于丢包有两种判断方式：

- 超时重传RTO (Retransmission Timeout) 超时
- 受到三个重复确认ACK

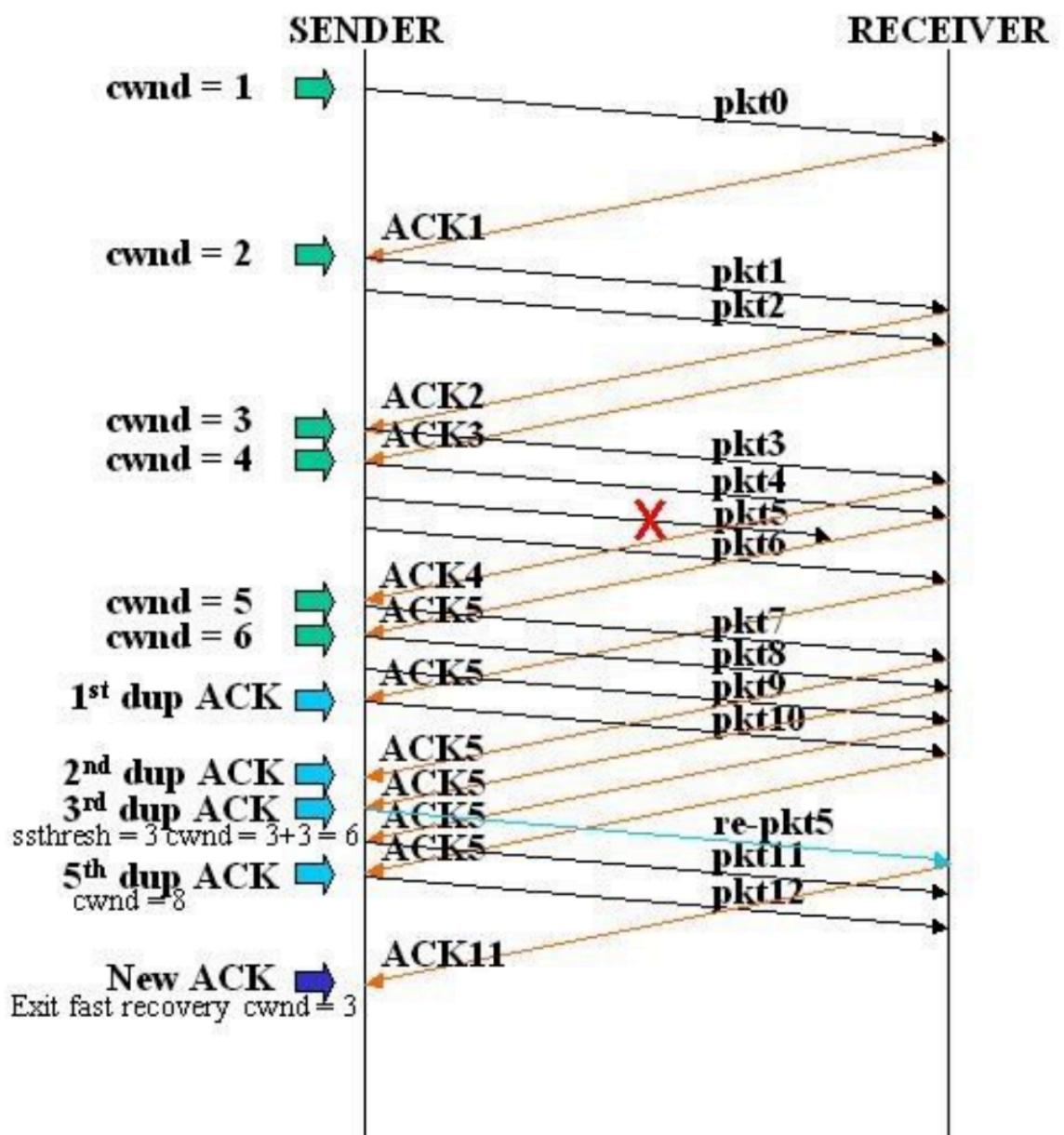
超时重传是TCP协议保证数据可靠性的一个重要机制，其原理是在发送一个数据以后就开启一个计时器，在一定时间内如果没有得到发送数据报的ACK报文，那么就重新发送数据，直到发送成功为止。

但是如果发送端接收到3个以上的重复ACK，TCP就意识到数据发生丢失，需要重传。这个机制不需要等到重传定时器超时，所以叫做 **快速重传**，而快速重传后没有使用慢启动算法，而是拥塞避免算法，所以这又叫做**快速恢复**算法。

4. 快速恢复算法 (Fast Recovery) -- 拥塞避免算法

在进入快速恢复之前， $cwnd$ 和 $ssthresh$ 已经被更改为原有 $cwnd$ 的一般。快速恢复算法逻辑如下：

- $cwnd = cwnd + 3 * MSS$ ，加 $3 * MSS$ 的原因是因为收到3个重复的ACK
- 重传DACKs指定的数据包
- 如果再收到DACKs，那么 $cwnd$ 大小增加1
- 如果收到新的ACK，表明重传的包成功了，那么推出快速恢复算法。将 $cwnd$ 设置为 $ssthresh$ ，然后进入拥塞避免算法



如图所示，第五个包发生了丢失，所以导致接收方接收到三次重复ACK，也就是ACK5.所以将ssthresh设置当时cwnd的一半，也就是 $6/2 = 3$ ，cwnd设置为 $3 + 3 = 6$.然后重传第5个包。当收到新的ACK时，也就是ACK11，则退出快速恢复阶段，将cwnd重新设置为当前的ssthresh，也就是3，然后进入拥塞避免算法阶段。

微服务和SOA

- 概念

SOA（面向服务的架构）：面向服务的架构（SOA）是一个组件模型，它将应用程序的不同功能单元（成为服务）通过这些服务之间定义良好的接口和契约联系起来。接口是采用的中立的方式进行定义的，它应该独立于实现服务的硬件平台、操作系统和编程语言。这使得构建在各种各样的系统中的服务可以以一种统一和通用的方式进行交互。

微服务：微服务架构是一种将单个应用程序作为一套小型服务开发的方法，每种应用程序都在自己的进程中运行，并与轻量级机制（通常是HTTP资源API）进行通信。这些服务是围绕业务功能构建的，可以通过全自动部署机制独立部署。这些服务的集中管理最少，可以用不同的编程语言编写，并使用不同的数据存储技术。

- 两者区别

1. 微服务剔除SOA中复杂的ESB企业服务总线，所有的业务智能逻辑在服务内部处理，使用Http（Rest API）进行轻量化通讯
2. SOA强调按水平架构划分为：前、后端、数据库、测试等，微服务强调按垂直架构划分，按业务能力划分，每个服务完成一种特定的功能，服务即产品
3. SOA将组件以library的方式和应用部署在同一个进程中运行，微服务则是各个服务独立运行
4. 传统应用倾向于使用统一的技术平台来解决所有问题，微服务可以针对不同业务特征选择不同技术平台，去中心统一化，发挥各种技术平台的特长
5. SOA架构强调的是异构系统之间的通信和解耦合；（一种粗粒度、松耦合的服务架构）
6. 微服务架构强调的是系统按业务边界做细粒度的拆分和部署

- 微服务的技术难点

1. 服务治理，维护成本增加
2. 资源分配
3. 数据库分割带来数据冗余，保证数据一致性更为复杂
4. 负载均衡，每个微服务都得单独处理

谈谈Redis的使用，Redis的数据主从同步

这里要回头结合redis那本书，说一下源码中redis的一些处理策略。

redis优势

1. 性能极高：redis能读的速度是1100000次/s，写的速度是81000次/s
2. 丰富的数据类型：支持二进制strings, lists, hashes, sets及ordered sets
3. 原子：redis的所有操作都是原子性的，意思就是要成功执行要么失败完全不执行，单个操作是原子性的。多个操作也支持事务。
4. 丰富的特性：redis还支持publish/subscribe, 通知，key过期，pipeline等实用特性

主从同步

redis的主从结构可以采用一主多从或者级联结构，redis主从复制可以根据是否全量分为全量同步和增量同步。

- 全量同步

redis全量赋值一般发生在**slave初始化阶段**，这时slave需要将master上的所有数据都复制一份。

1. 从服务器连接主服务器，发送**sync**命令
2. 主服务器接收到**sync**命令后，开始执行**bgsave**命令生成**rdb**文件并使用缓冲区记录此后执行的所有写命令
3. 主服务器**bgsave**执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令
4. 从服务器收到快照文件后丢弃所有旧数据，载入收到的快照
5. 主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令
6. 从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令

完成上面几个步骤后就完成了从服务器数据初始化的所有操作，从服务器此时可以接收来自用户的读请求。

- 增量同步

redis增量复制是指slave初始化后正常开始工作时主服务器发生的写操作同步到从服务器过程。**增量复制**的过程主要是主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令。

- redis主从同步策略

主从刚刚连接的时候，进行全量同步；全同步结束后，进行增量同步。当然，如果有需要，slave在任何时候都可以发起全量同步。**redis策略是，无论如何首先会尝试进行增量同步，如果不成功，要求从集进行全量同步。**

- 注意点

如果多个slave断线，需要重启的时候，因为只要slave启动，就会发送**sync**请求和主机全量同步，当多个同时出现的时候，可能会导致**master IO剧增宕机**。

C++一个空类

- 空类大小

1字节

- 实现方式

```
1 #include<stdio.h>
2 #include<iostream>
3
4 class A {};
5 class B {};
6 class C: public A {
7     virtual void fun() = 0;
8 };
9
```

```

10 class D: public B, public C {
11     virtual void fund() = 0;
12 };
13
14 class E: public D {
15     static int a;
16 };
17
18 using namespace std;
19 int main(void) {
20     cout << "sizeof(A)" << sizeof(A) << endl;
21     cout << "sizeof(B)" << sizeof(B) << endl;
22     cout << "sizeof(C)" << sizeof(C) << endl;
23     cout << "sizeof(D)" << sizeof(D) << endl;
24     cout << "sizeof(E)" << sizeof(E) << endl;
25     return 0;
26 }

```

```

sizeof(A)1
sizeof(B)1
sizeof(C)8
sizeof(D)8
sizeof(E)8
-----

```

实例化的原因（空类同样可以被实例化），每个势力在内存中都有一个独一无二的地址，为了达到这个目的，编译器往往会给一个空类银行的加上一个字节，这样空类在实例化之后在内存得到了独一无二的地址。多以A的大小为1.

而虚函数分配地址为一个指针地址，64位操作系统下为8字节。

总结来说：

1. 类的非静态成员数据的类型大小之和（静态成员被编译器放在一个global data members中）
 2. 有编译器额外加入的成员变量的大小，用来支持语言的某些特性（如：指向虚函数的指针）
 3. 为了优化存取效率，进行的边缘调整
 4. 与类中的构造函数，析构函数以及其他成员函数无关
- 空类默认有哪些函数

编译器会自动为你生成一个默认构造函数、一个默认拷贝构造函数、一个默认拷贝赋值操作符、一个默认析构函数。这些函数只有在第一次被调用时，才会被编译器所创建。所有这些函数都是inline和public的。

分布式缓存

技术框架

redis

memcached

缓存的几种淘汰策略

事实上不止是缓存，只要设计的内存，或者存储空间，所用到的策略和方法都是那几种，在操作系统的虚拟内存页面置换算法，有：最佳置换算法，FIFO、LRU、LFR、CNRU等等，和这里的缓存的淘汰策略如出一辙，只是在操作系统里是用来判定怎样替换更好。我个人的观点：所有涉及到存储空间里的内容清除，置换的，都是这些算法。

缓存淘汰策略用于决定缓存系统中哪些数据应该被删去。常见算法类型包括LFU、LRU、ARC、FIFO、MRU。

- 最不经常使用算法 (LFU)

这个缓存算法使用一个计数器来记录条目被访问的频率。通过使用LFU缓存算法，最低访问数的条目首先被移除。这个方法并不经常使用，因为它无法对一个拥有最初高访问率之后长时间没有被访问的条目缓存负责。

- 自适应缓存替换算法 (ARC)

在IBM Almaden研究中心开发，这个缓存算法同事跟踪记录LFU和LRU，以及驱逐缓存条目，来获得可用缓存的最佳使用。

- 先进先出算法 (FIFO)

FIFO英文是First In First Out的缩写，是一种先进先出的数据缓存器，他与普通存储器的区别是没有外部读写地址线，这样使用起来非常简单，但缺点就是只能顺序写入数据，顺序的读出数据，其数据地址由内部读写指针自动+1完成，不能像普通存储器那样可以由地址线决定读取或写入某个指定地址。

- 最近最常使用算法 (MRU)

这个缓存算法最先移除最近最常使用的条目。一个MRU算法擅长处理一个条目越久，越容易被访问的情况。

Java内存缓存

Google Guava (一个java依赖)

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>me.xueyao.cache</groupId>
    <artifactId>java-demo</artifactId>
    <version>1.0.0</version>

    <dependencies>
        <dependency>
            <groupId>javax.cache</groupId>
            <artifactId>cache-api</artifactId>
```

```
<version>1.1.0</version>
</dependency>

<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>27.0.1-jre</version>
</dependency>
</dependencies>
</project>
```

Guava是Google guava中的一个内存缓存模块，用于将数据缓存到JVM内存中。实际项目开发中经常将一些公共或者常用的数据缓存起来方便快速访问。

如何解决线上问题，步骤和解决办法（运维知识）

分为以下几个步骤：

1. 明确责任人，避免互相推诿
2. 治标，短期解决方案，紧急处理
3. 追根溯源，长期解决方案，避免再次出现
4. 举一反三，查找类似问题
5. 如何避免，防止开发新功能再次踩雷
6. 跟进人，有的问题处理时间较长，需要有个跟进人
7. 整理分享，最好过一段时间（一个月）整理近期发生的问题，进行分享

假如觉得以上步骤有点繁琐，简化版：

1. 评估影响范围
2. 试图重现问题
3. 临时方案和终极方案
4. 风险评估以及持续优化

遇到线上问题，恢复生产、降低损失是第一要务，修复bug是其次的。

软件工程师的核心竞争力是什么？本质就是一种解决问题的能力，一步步分析、解决和预防问题。

日志分析

线上bug，一般以查日志为主，所以前期的代码日志埋点很重要。当然后期的日志关键信息分析，也很重要。所以，搭建像ELK或Splunk这样的日志分析系统，方便日志查询。

除开gdb以外，有哪些服务端调试手段

- telnet可以调试端口是否正常
- ping可以查看是否丢包
- iostat可以查看磁盘是否超载
- tcpdump、wireshark可以查看TCP包

- 自研的监控平台可以查看机器历史状态
- top可以查看cpu和内存使用情况
- valgrind可以看内存，还有free
- trace -p可以查看
- dtruss/dtrace内核调用跟踪监视工具
- traceroute可以用查看路由路径，并且测算数据来回需要多久时间。

大部分情况下，我们会在linux主机系统下，直接执行命令：`traceroute hostname`

```
~/ » traceroute www.baidu.com
ethancao@EthanCaodeMacBook-Pro
traceroute: Warning: www.baidu.com has multiple addresses; using 14.215.177.39
traceroute to www.a.shifen.com (14.215.177.39), 64 hops max, 52 byte packets
 1  10.56.236.2 (10.56.236.2)  2.388 ms  2.221 ms  2.267 ms
 2  10.39.0.5 (10.39.0.5)  1.627 ms  1.573 ms  1.607 ms
 3  218.17.197.193 (218.17.197.193)  3.851 ms  3.185 ms  3.578 ms
 4  183.56.64.13 (183.56.64.13)  4.236 ms
    183.56.64.17 (183.56.64.17)  4.519 ms
    183.56.64.13 (183.56.64.13)  4.055 ms
 5  117.176.37.59.broad.dg.gd.dynamic.163data.com.cn (59.37.176.117)  4.726 ms
    125.176.37.59.broad.dg.gd.dynamic.163data.com.cn (59.37.176.125)  3.551 ms
    117.176.37.59.broad.dg.gd.dynamic.163data.com.cn (59.37.176.117)  3.643 ms
 6  202.105.106.37 (202.105.106.37)  5.607 ms
    119.145.47.77 (119.145.47.77)  4.626 ms *
 7  113.96.0.22 (113.96.0.22)  15.477 ms
    113.96.0.18 (113.96.0.18)  9.452 ms
    113.96.5.50 (113.96.5.50)  5.690 ms
 8  * 106.96.135.219.broad.fs.gd.dynamic.163data.com.cn (219.135.96.106)
    9.215 ms  8.518 ms
 9  14.215.32.130 (14.215.32.130)  8.011 ms *  10.967 ms
10  * * *
11  * * *
```

说明

记录按序列号从1开始，每个纪录就是一跳，每跳表示一个网关，我们看到每行有三个时间，单位是ms，其实就是-q的默认参数。探测数据包向每个网关发送三个数据包后，网关响应后返回的时间；如果您用 traceroute -q 4 www.58.com，表示向每个网关发送4个数据包。

有时我们traceroute 一台主机时，会看到有一些行是以星号表示的。出现这样的情况，可能是防火墙封掉了ICMP的返回信息，所以我们得不到什么相关的数据包返回数据。

有时我们在某一网关处延时比较长，有可能是某台网关比较阻塞，也可能是物理设备本身的原因。当然如果某台DNS出现问题时，不能解析主机名、域名时，也会有延时长的现象；您可以加-n 参数来避免DNS解析，以IP格式输出数据。

如果在局域网中的不同网段之间，我们可以通过traceroute 来排查问题所在，是主机的问题还是网关的问题。如果我们通过远程来访问某台服务器遇到问题时，我们用到traceroute 追踪数据包所经过的网关，提交IDC服务商，也有助于解决问题；但目前看来在国内解决这样的问题是比较困难的，就是我们发现问题所在，IDC服务商也不可能帮助我们解决。

- Hash索引的实现原理
- Mysql的Innodb和MyIsam两种引擎的区别
- Mysql数据库索引的实现原理
- 一台机器只有1G物理内存，是否程序可以获取到2G，为什么？
- 操作系统虚拟内存是如何实现的
- Vector扩容的原理
- 什么缓解会发送RST包（TCP时序图）
- 垃圾回收、Redis、异常错误捕获、消息队列、kafka日志框架
- 秒杀系统怎么设计，性能优化（网站反应很慢，怎么优化）
- 怎么保证分布式数据的一致性，项目经验
- Redis数据结构、持久化方式和高峰期如何避免雪崩、主从同步
- 唯一索引和key索引。Http协议相关，如何实现一个hash数据结构
- IO模型
- TCP粘包问题
- web漏洞
- http2.0有哪些特性
- IO复用select、poll、epoll区别
- ES、ZK、ETCD、kafka了解

MQ丢失消息怎么处理，有没有机制保证

kafka和rabbitmq的区别

- 在应用场景方面

RabbitMQ

RabbitMQ遵循AMQP协议，由内在高并发的erlang语言开发，用在实时的对可靠性要求比较高的消息传递上，适合企业级的消息发送订阅，也是比较受到大家欢迎的。

kafka

kafka是Linkedin于2010年12月份开源的消息发布订阅系统，它主要用于处理活跃的流式数据，大数据量的数据处理上，常用日志采集，数据采集上。

ActiveMQ

1. 异步调用
2. 一对多通信
3. 做多个系统的继承，同构、异构
4. 作为RPC的替代
5. 多个应用相互解耦
6. 作为事件驱动架构的幕后支撑
7. 为了提高系统的可伸缩性

- 在架构模型方面

RabbitMQ

遵循AMQP协议，RabbitMQ的broker由Exchange, Binding, queue组成，其中exchange和binding组成了消息的路由键；客户端Producer通过连接channel和server进行通信，Consumer从queue获取消息进行消费（长连接，queue有消息会推送到consumer端，consumer循环从输入流读取数据）。rabbitMQ以broker为中心；有消息的确认机制。

kafka

kafka遵从一般的MQ结构，producer, broker, consumer，以consumer为中心，消息的消费信息保存在客户端**consumer**上，consumer根据消费的点，从broker上批量pull数据；无消息确认机制。

- 吞吐量

kafka具有高吞吐量，内部采用消息的批量处理，zero-copy机制，数据的存储和获取是本地磁盘顺序批量操作，具有O(1)的复杂度，消息处理的效率很高。

rabbitmq在吞吐量方面稍逊于kafka，他们的出发点不一样，rabbitmq支持对消息的可靠的传递，支持事务，不支持批量操作；基于存储的可靠性要求可以采用内存或者硬盘。

- 可用性

rabbitmq支持mirror的queue，主queue失效，mirror queue接管。

kafka的broker支持主备模式

- 在集群负载均衡方面

kafka采用zookeeper对集群中的broker、consumer进行管理，可以注册topic到zookeeper上；通过zookeeper的协调机制，producer保存对应topic的broker信息，可以随机或者轮询发送到broker上；并且producer可以基于语义指定分片，消息发送到broker的某分片上。

rabbitmq的负载均衡需要单独的loadbalancer进行支持。

MQ消息乱序怎么办

- 分布式锁怎么做？
- 负载均衡怎么做？
- TCP、项目，解决过的难点问题，思路过程
- 同步异步
- 进程线程

Redis扩容和缩容

Redis如何处理限频

- 分布式架构，拿Nginx来讲的
- 进程间通信
- zookeeper、kafka的知识，日志处理的架构，分布式系统的基本要素啥的

ThreadLocal用来干嘛的，底层实现原理

threadlocal使用方法很简单：

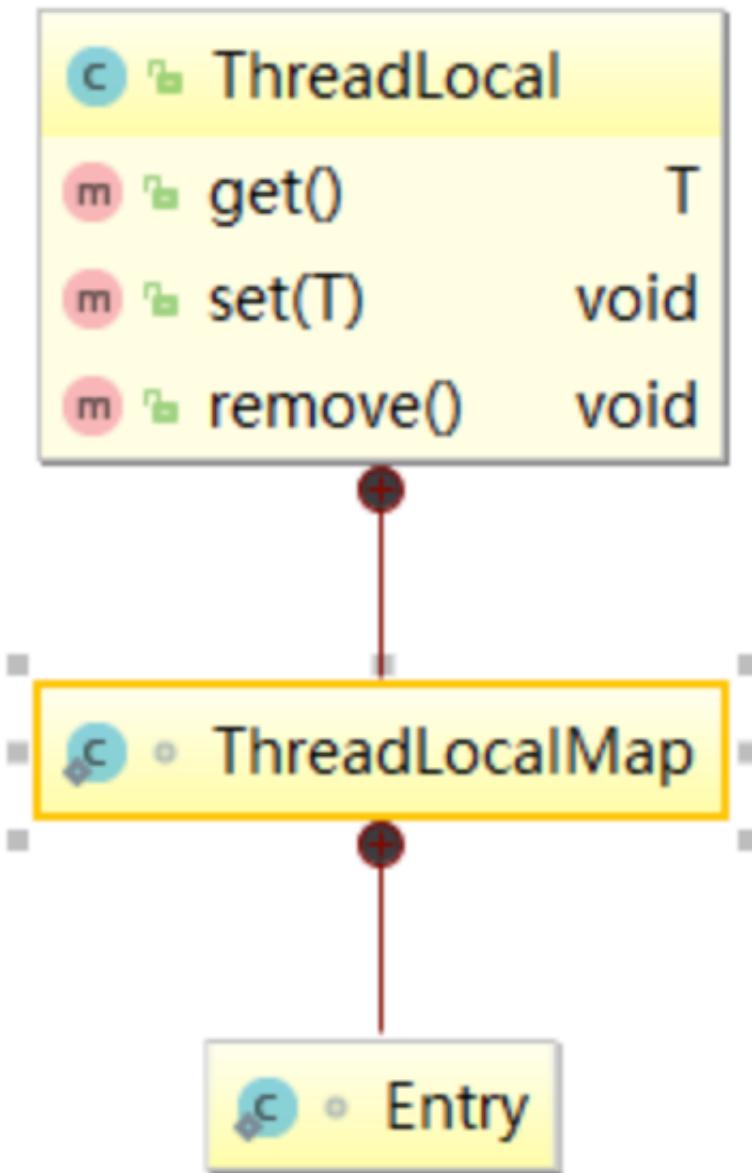
```
static final ThreadLocal<T> sThreadLocal = new ThreadLocal<T>();  
sThreadLocal.set()  
sThreadLocal.get()
```

threadlocal而是一个县城内部的存储类，可以在指定线程内部存储数据，数据存储以后，只有指定线程可以得到存储数据。

多线程访问同一个共享变量的时候容易出现并发问题，特别是多个线程对一个变量进行写入的时候，为了保证线程安全，一般使用者在访问共享变量的时候需要进行额外的同步措施才能保证线程安全性。ThreadLocal是除了加锁这种同步方式之外的一种保证，一种规避多线程访问出现线程不安全的方法，当我们在创建一个变量后，如果每个线程对其进行访问的时候访问的都是线程自己的变量，这样就不会存在编程不安全的问题。

ThreadLocal是JDK包提供的，它提供线程本地变量，如果创建一个Treadlocal变量，那么访问这个变量的每个线程都会有这个变量的副本，在实际多线程操作的时候，操作的是自己本地内存中的变量，从而规避了线程安全问题。

ThreadLocal类结果及方法解析



上图可知：`ThreadLocal`三个方法`get`、`set`、`remove`以及内部类`ThreadLocalMap`

值得注意的是，python也有`ThreadLocal`使用方式，不仅Java有

`ThreadLocal`是如何做到为每一个线程维护变量的副本的呢？其实实现的思路很简单：在`ThreadLocal`类中有一个Map，用于存储每一个线程的变量副本，Map中元素的键为线程对象，而值对应线程的变量副本。

ThreadLocalMap<threadObject, threadValCopy>对象保存了对应关系

AUC如何计算

`Are under curve`是一个模型评价指标，用于分类任务。那么这个指标代表什么呢？这个指标项表达的含义，简单来说其实就是随机抽出一对样本（一个正样本，一个负样本），然后用训练得到的分类器来对这两个样本进行预测，预测得到正样本的概率大于负样本概率的概率。

$$AUC = P(P_{\text{正样本}} > P_{\text{负样本}})$$

其他用于衡量机器学习的指标：

召回率 (Recall) 、准确率 (Precision) 、F1值等

计算AUC通常是由机器或者库完成，当然，为了更好的理解AUC，这里简述一个例子，计算AUC：

在有M个正样本，N个负样本的数据集里面。一共有M*N对样本（一对样本集，一个正样本与一个负样本）。统计这M*N对样本里正样本的预测概率大于负样本的预测概率的个数。

$$I(P_{\text{正样本}}, P_{\text{负样本}}) = \begin{cases} 1, & P_{\text{正样本}} > P_{\text{负样本}} \\ 0.5, & P_{\text{正样本}} = P_{\text{负样本}} \\ 0, & P_{\text{正样本}} < P_{\text{负样本}} \end{cases}$$

ID	label	pro
A	0	0.1
B	0	0.4
C	1	0.35
D	1	0.8

如表所示，有4条样本。2个正样本，2个负样本，那么M*N=4。即总共有4个样本对。分别是：

(D, B), (D, A), (C, B), (C, A)

比如：(D, B)，正样本D预测的概率大于负样本B预测的概率（也就是D的得分比B高），记为1

同理，(C, B)，正样本C预测概率小于负样本C预测概率，记为0

$$\left(\frac{1 + 1 + 1 + 0}{4} \right) = 0.75$$

最后可以算得，总共3个符合正样本得分高于负样本得分，故最后的AUC为0.75.

wide和deep之间的区别

推荐系统在电商等平台使用广泛，这里讨论wide&deep推荐模型，初始是由google推出的，主要用于app的推荐。

- 概念理解

wide&deep模型，旨在使得训练得到的模型能够同时获得记忆 (memorization) 和泛化 (generalization) 能力：

记忆 (memorization) 即从历史数据中发现item或者特征之间的相关性。

泛化 (generalization) 即相关性的传递，发现在历史数据中很少或者没有出现的新的特征组合。

具体到模型定义角度，wide是指广义线性模型（Wide Linear Model），deep是指深度神经网络（Deep Natural Network）。

两者区别：

Memorization趋向于更加保守，推荐用户之前有过的items。相比之下，generalization更加趋向于提高推荐系统的多样性（diversity）。

Wide & Deep包括两部分：线性模型(LR逻辑回归)+DNN（深度神经网络）部分。结合上面两者的优点，平衡memorization和generalization。

- 讲讲历史

wide & deep是google提出来的，提出时间是2016年，算起来这个算法还比较新了，附带论文《Wide & Deep Learning for Recommender Systems》，链接：<https://arxiv.org/pdf/1606.07792.pdf>，这个算法在谷歌应用商店做排序。

另外一个是DeepFM，来自于2017年华为提出来的《DeepFM: A Factorization-Machine based Neural Network for CTR Prediction》，链接：<https://arxiv.org/pdf/1703.04247.pdf>，这个华为用在应用商店做排序的。

以上两个算法即可以在广告中做CTR预估，也可以在推荐系统中做排序。

此外，阿里巴巴2018年发布了Graph Embedding (GE)，附带论文《Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba》，链接：<https://arxiv.org/pdf/1803.02349.pdf>，这个是用在淘宝的推荐系统中。

Joint Training和Ensemble

joint training和ensemble是有区别的，ensemble里训练时模型互不影响，只是在预测时一起作用，joint training在训练时模型一起训练。

广告和推荐算法工程实现

注意点

1. 将广告和推荐算法与工程后台框架在业务上要解耦合，比如定向，预估都会不可避免的出现算法调优涉及到需要改动框架代码的尴尬境地。
2. 如果涉及到微服务时效性（模块超时）问题，假如独立出来服务，要考虑多个业务（子微服务）提供服务时，主进程等待子模块返回这个操作是多线程并发，并且支持异步。
3. 考虑对不同的服务做qps和响应时间的划分，对时效性低的要和时效性高的区别对待。

问LR为什么不能用MSE

顺便写一些LR的损失函数

神经网络优化器

我们知道，神经网络的学习的目的就是寻找适合的参数，使得损失函数的值尽可能小。解决这个问题的过程被称为最优化。解决这个问题使用的算法叫做优化器。

- SGD

随机梯度下降法 (Stochastic gradient descent) : SGD的想法即使沿着梯度的方向前进一定距离。

SGD的优点是简单，容易实现。但是其缺点就是低效，因为有的时候梯度的方向并没有指向最小值的方向。低效的原因有两大方面：

1. 函数呈延伸状态，梯度指向了“谷底”。这使得损失函数值不停的在震荡。
2. 梯度方向指向了极小值。因为所有维度的梯度在这附近都接近于0，这使得损失函数在这里变化的很慢。

改进办法：引入 **Momentum**

实际上震荡是不可避免的，我们只能考虑减轻震荡。使得梯度方向不变的维度上速度变快，梯度方向有所改变的维度上的更新速度变慢，这样就可以加快收敛并减小震荡。

- AdaGrad

这种方法主要是为了解决SGD遇到鞍点或者极小值点后学习变慢的问题。

在神经网络中有一种方法经常被使用：学习率衰减方法 (Learning rate decay)，也就是说随着学习的进行，使学习率逐渐减少。AdaGrade进一步发展了这个想法，他会为参数的每一个元素适当的调整学习率。

AdaGrad的优点是可以动态的调整学习率

缺点是AdaGrad会记录过去所有的梯度平方和，最后有可能不再更新。针对这个问题有一个方法叫做 RMSProp进行了优化。

- Adam

Adam直观的来讲就是融合了Mementum和AdaGrad方法。

机器学习方法论

最宝贵的经验不是书上或者论文中明白写出来的：

1. 问题是什么？
 2. 用什么指标来度量？
 3. 问题的复杂度是什么量级？
 4. 模型不好，本质上就是**模型复杂度没有匹配问题负载**，哪里是模型复杂度的瓶颈？
- 标注数据
 - 人的智慧，创造有价值的特征，还是说可以用深度学习来创造feature？
 - 海量用户行为数据
 - 模型的调整，过于简单的模型，相当于强行约束了复杂度的上限，使得明明有海量的数据硬是用不上
5. 找不到便利的数据和特征，就可以想想是否能把问题分解成子问题，也许在子问题空间下，你能想到更好的特征、数据和模型。同事分解了问题，就能够引入更多的人员来并行工作
 6. 知道怎么做干净的实验，系统性的探索所有的可能性，排除无关因素

在所有的经验中，狭义的模型和特征工程只是**第4点**的一部分而已。

但是剩余的部分也都是技术（与具体的应用领域无关），是建立在读机器学习的能力与界限有清楚认识的基础上的一些抽象。

web server大文件上传下载

webserver+sftp? 目测不行

有一个一般解决思路：

1. 大文件上传时进行分片
2. 分片上传
3. 在服务端对分片文件进行合并

具体实现上：

1. 全端做数据分片，推荐百度的WebUploader来实现
2. 对于文件合并，前端在分片之后，在请求服务端合并的时候，请求中要带上分片序号和大小，服务器按照请求数据中给的分片序号和每片分块大小算出开始位置。与读取到的文件片段数据，写入文件即可。这里合并后的文件会存储两个路径，一个是当前网盘目录下的路径，一个是真实的永久路径（目的是为了实现秒传的功能）。
3. 对于 秒传 这个功能的实现，其实原理技术检验文件MD5，在一个文件上传前先获取文件内容MD5值或者部分取值MD5。然后查找自己的记录是否已存在相同的MD5，如果存在就直接从服务器真是路径取，而不需要重新进行分片上传了，从而达到秒传的效果。

用sed替换某一行的某个词

这里介绍一个比较常见的用法，文本内容如下：

文本内容如下：

```
aaa bbb ccc 111 222 abc
eee fff ggg 111 222 efg
111 222
aaa ccc ddd 111 222 acd
```

需求：

在有aaa的行中，将 111 替换为 AAA，将 222 替换为 BBB
即，输出结果为：

```
aaa bbb ccc AAA BBB abc
eee fff ggg 111 222 efg
111 222
aaa ccc ddd AAA BBB acd
```

方法如下：

```
sed -i '/aaa/ { s/111/AAA/g; s/222/BBB/g; }' filename
```

更改一下，需要在字符串 AAA 所在的行，头尾各添加相应字符串：

```
sed -i '/AAA/ {s/^/HEAD&/g; s/$/&TAIL/g;}' filename
```

kill命令的作用/选项解释

本质上来说，kill命令指示用来向进程发送一个信号，至于这是什么信号，是用户指定的。

也就是说，kill会向操作系统内核发送一个信号（多是终止信号）和目标进程的PID，然后系统内核根据收到的信号类型，对指定进程进行相应的操作。

常用的信号列表：

信号 编号	信号 名	含义
0	EXIT	程序退出时收到该信号
1	HUP	挂掉电话线或中断连接的挂起信号，这个信号也会造成某些进程在没有终止的情况下重新初始化
2	INT	表示结束进程，但并不是强制性的，常用的“Ctrl+C”组合键发出就是一个kill -2的信号
3	QUIT	退出
9	KILL	杀死进程，即强制结束进程
11	SEGV	段错误
15	TERM	正常结束进程，是kill命令的默认信号
-1		让进程重启
-19		让进程暂停

客户端频控

10秒频控，在10秒发起10万个请求，在第11秒发起10万请求，这样我就在2秒内发起了20w个请求，你如何解决？

```
(降低时间窗口期，10秒的窗口期换成2秒)
```

服务端频率控制方法：

参考DSP的频率控制，使用用户设备的MAC地址、uuid、muid（用户设备md5）在redis里面做最近一次访问的时间戳保存，服务端每次都检查用户是否需要被频率控制。

频率控制一方面是为了降低服务器的处理负担，另外一方面是减少重复特征，为模型训练提供有效素材。频率控制做的不好，或者缺乏服务端频率控制，很有可能会造成日志数据被“清洗”的悲剧。

如何做分库访问db

基本思想就是把一个数据库切分成多个部分放在不同的Server上，从而缓解单一数据库的性能问题。这个有点类似redis的分布式部署，**hash散列**。不太严格的讲，对于海量数据的数据库，如果是因为表多而数据不多，这个时候适合使用垂直切分，即把关系紧密（比如同一个模块）的表切分出来放在一个server上。如果表并不多，但是每张表的数据非常多，这个时候适合水平切分，即把表的数据按照某种规则（比如按ID散列）切分到多个server上。当然，现实中更多的是两种情况混杂到一起，这时候需要根据实际情况作出选择，也可能会综合使用垂直与水平切分，从而将原有数据库切分成类似矩阵一样可以无限扩充的server阵列。

垂直切分

垂直切分最大的特点就是规则简单，实施也更为方便，尤其适合各业务之间的耦合度非常低，相互影响很小，业务逻辑非常清晰的系统。在这种系统中，可以很容易做到将不同业务模块所使用的表分拆到不同的数据库中。根据不同的表来进行拆分，对应用程序的影响也更小，拆分规则也会比较简单清晰。

水平切分

水平切分与垂直切分相比，相对来说稍微复杂一些。因为要将同一个表中的不同数据拆分到不同的数据库中，对于应用程序来说，拆分规则本身就较垂直拆分更为复杂，后期的数据维护也会更为复杂一些。

主键/外键/索引

定义

主键：唯一标识一条记录，不能有重复的，不允许空

外键：表的外键是另一个表的主键，外键可以有重复的，可以是空值

索引：该字段没有重复值，但可以有一个空值

作用

主键：用来保证数据完整性

外键：用来和其他表建立联系用的

索引：是提高查询排序的速度

个数

主键：主键只能有一个

外键：一个表可以有多个外键

索引：一个表可以有多个唯一索引

总体来说，SQL的主键和外键就是起约束作用。例如：外键更新时，不能改为主键中没有的值

IO次数

- 走主键一次io
- 走联合索引 两次io

- 不走索引 多次io

在32gb的内存里对1TB数据排序

hadoop和hive

hadoop和hive关系

hive是hadoop的延伸

hadoop是一个分布式的软件处理框架，hive是一个提供了查询功能的数据仓库，最核心的就是hadoop提供了FS（file system）抽象，能够支持hive进行数据存储。那计算机类比，hadoop就是文件系统，而hive就是mysql的应用程序。

hive的udf是什么

udf（User Define Function）用户自定义函数，写过flink udf的同学都知道，是一个意思，就是基于scala或者java写了一些扩展插件，用户可以在hive里面使用扩展程序掉用这些jar插件，以执行一些自定义的操作。

一张大表和一张小表join（数据倾斜）

Hive不能把过于大型的数据集合从RDD转成collect，collect是放到本机内存，会导致内存超限，应该在最终结果获取环节使用collect。

总体来说有两种解决方案：

1. 普通的join，将小表放在前面，效率会高。hive会将小表进行缓存。
2. 使用MapJoin。

MapJoin 可以解决 数据倾斜 问题。

基本原理是：在小数据量的情况下，SQL会将用户指定的小表全部加载到执行join操作的程序的内存中，从而加快join的执行速度。而且执行map操作的时候，逐一和大表匹配，省去了reduce操作。

```
select /*+MAPJOIN(b)*/ a.a1,a.a2,b.b2 from tablea a JOIN tableb b ON a.a1=b.b1
```

MapJoin的join发生在map阶段，普通join的join发生在reduce阶段，mapjoin可以提高效率。

使用MAPJOIN时，需要注意：

LEFT OUTER JOIN的左表必须是大表；
RIGHT OUTER JOIN的右表必须是大表；
INNER JOIN左表或右表均可以作为大表；
FULL OUTER JOIN不能使用MAPJOIN；
MAPJOIN支持小表为子查询；
使用MAPJOIN时需要引用小表或是子查询时，需要引用别名；
在MAPJOIN中，可以使用不等值连接或者使用OR连接多个条件；
目前ODPS在MAPJOIN中最多支持指定6张小表，否则报语法错误；
如果使用MAPJOIN，则所有小表占用的内存总和不得超过512M（解压后的逻辑数据量）。

常见排序算法时间复杂度

(1)冒泡排序：

是相邻元素之间的比较和交换，两重循环 $O(n^2)$ ；所以，如果两个相邻元素相等，是不会交换的。所以它是一种稳定的排序方法

(2)选择排序：

每个元素都与第一个元素相比，产生交换，两重循环 $O(n^2)$ ；举个栗子，5 8 5 2 9，第一遍之后，2会与5交换，那么原序列中两个5的顺序就被破坏了。所以不是稳定的排序算法

(3)插入排序：

插入排序是在一个已经有序的小序列的基础上，一次插入一个元素。刚开始这个小序列只包含第一个元素，事件复杂度 $O(n^2)$ 。比较是从这个小序列的末尾开始的。想要插入的元素和小序列的最大者开始比起，如果比它大则直接插在其后面，否则一直往前找它该插入的位置。如果遇见了一个和插入元素相等的，则把插入元素放在这个相等元素的后面。所以相等元素间的顺序没有改变，是稳定的。

(4)快速排序

快速排序有两个方向，左边的i下标一直往右走，当 $a[i] \leq a[center_index]$ ，其中center_index是中枢元素的数组下标，一般取为数组第0个元素。而右边的j下标一直往左走，当 $a[j] > a[center_index]$ 。如果i和j都走不动了， $i \leq j$ ，交换 $a[i]$ 和 $a[j]$ ，重复上面的过程，直到 $i > j$ 。交换 $a[j]$ 和 $a[center_index]$ ，完成一趟快速排序。在中枢元素和 $a[j]$ 交换的时候，很有可能把前面的元素的稳定性打乱，比如序列为5 3 3 4 3 8 9 10 11，现在中枢元素5和3(第5个元素，下标从1开始计)交换就会把元素3的稳定性打乱，所以快速排序是一个不稳定的排序算法，不稳定发生在中枢元素和 $a[j]$ 交换的时刻。时间复杂度 $O(n\log n)$ 。

100G数据，使用4G内存排序

思路很简单，先分段排序，存储到临时文件中，然后合并。

使用滑动窗口（内存）读取数据到内存进行排序，然后依次写入文件。

(5)归并排序

归并排序是把序列递归地分成短序列，递归出口是短序列只有1个元素(认为直接有序)或者2个序列(1次比较和交换)，然后把各个有序的段序列合并成一个有序的长序列，不断合并直到原序列全部排好序。可以发现，在1个或2个元素时，1个元素不会交换，2个元素如果大小相等也没有人故意交换，这不会破坏稳定性。那么，在短的有序序列合并的过程中，稳定性是否受到破坏？没有，合并过程中我们可以保证如果两个当前元素相等时，我们把处在前面的序列的元素保存在结果序列的前面，这样就保证了稳定性。所以，归并排序也是稳定的排序算法。

(6) 基数排序

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序，最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以其是稳定的排序算法。

(7) 希尔排序(shell)

希尔排序是按照不同步长对元素进行插入排序，当刚开始元素很无序的时候，步长最大，所以插入排序的元素个数很少，速度很快；当元素基本有序了，步长很小，插入排序对于有序的序列效率很高。所以，希尔排序的时间复杂度会比 $O(n^2)$ 好一些。由于多次插入排序，我们知道一次插入排序是稳定的，不会改变相同元素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱，所以shell排序是不稳定的。

(8) 堆排序

我们知道堆的结构是节点*i*的孩子为 $2i$ 和 $2i+1$ 节点，大顶堆要求父节点大于等于其2个子节点，小顶堆要求父节点小于等于其2个子节点。在一个长为n的序列，堆排序的过程是从第 $n/2$ 开始和其子节点共3个值选择最大(大顶堆)或者最小(小顶堆)，这3个元素之间的选择当然不会破坏稳定性。但当为 $n/2-1, n/2-2, \dots, 1$ 这些个父节点选择元素时，就会破坏稳定性。有可能第 $n/2$ 个父节点交换把后面一个元素交换过去了，而第 $n/2-1$ 个父节点把后面一个相同的元素没有交换，那么这2个相同的元素之间的稳定性就被破坏了。所以，堆排序不是稳定的排序算法

C++依赖注入 (IOC)

依赖注入主要的用途，我个人理解为两点：

1. 类似java的“反射”
2. 方便后期扩展

依赖注入大概有三种方式：

- 构造器注入

将被一拉对象通过构造函数的参数注入给依赖对象，并且在初始化对象的时候注入。

优点：对象初始化完成后便可获得可使用的对象。

缺点：

1. 当需要注入的对象很多的时候，构造器的参数列表将会很长
2. 不够灵活。若有多重注入方式，每种方式只需注入指定几个依赖，哪儿就需要提供多个重载的构造函数，麻烦。

- setter方法注入

IOC Service Provider通过调用成员变量提供的setter函数将被依赖对象注入给依赖类。

优点：可以选择性地注入需要的对象

缺点：依赖对象初始化完成后由尚未注入被依赖对象，因此还不能使用。

- 接口注入

依赖类必须要实现指定的接口，然后实现该接口中的一个函数，该函数就是用于依赖注入。该函数的参数就是要注入的对象。

优点：接口注入中，接口的名字、函数的名字都不重要，主要保证函数的参数是要注入的对象类型即可。

缺点：侵入行太强，不建议使用。

PS：什么是侵入行？

如果类A要使用别人提供的一个功能，若为了使用这功能，需要在自己的类中增加额外的代码，这就是侵入性。

- 基于注解

基于Java的注解功能，在私有变量前加“@Autowired”等注解，不需要显式定义以上三种代码，便可以让外部容器传入对应的对象。该方案相当于定义了一个public的set方法，但是因为没有真正的set方法，从而不会为了实现依赖注入导致暴露了不该暴露的接口（因为set方法只想让容器访问来注入而不希望其他依赖此类的对象访问）。

下面举一个例子来帮助我们理解C++伪代码的依赖注入：

```
// 当伪码用，不用关注语法
class IB {}
class B1 : public IB{}
class B2 : public IB{}


struct A {
    A(IB *b) {}
}

// 一般处理
A *a1 = new A(new B1);
A *a2 = new A(new B2);

// 用工厂模式，A不想依赖IB的具体实现
// 省略工厂代码
A *a1 = new A(Factory::create("B1"))
A *a2 = new A(Factory::create("B2"))

// A需要依赖工厂类，工厂类以来IB的具体实现B1/B2
// 试试依赖注入容器怎么搞(API随便写的，各个容器库不一样)
IOCContainer ioc;
ioc.bind<A, B1>("B1");
ioc.bind<A, B2>("B2");

A *a1 = ioc.resolve<A>("B1")
```

- Cassandra是如何存储的？为什么不用mysql？二者的区别。
- Elastic search如何存储，如何查询，其中做了什么优化？什么是倒排索引？
- 数据库、mysql和postgre的事务是如何实现的。

- postgresql和mysql数据库的区别是什么
- 32位的系统最多可以new出多大的内存?
- 32位的系统，最多可以创建多少的线程?
- 什么进程kill -9 杀不掉（处于D状态的进程杀不掉）
- 跨国的专线（高时延，大带宽）的TCP通讯如何优化?
- 设置一个合理的发送和接收缓冲区（带宽时延乘积）
- 调大初始拥塞控制窗口大小
- 挑战一致性hash扩容过程，配额服务扩容如何做到平滑，对业务不影响，或者业务不感知
- 进程间通讯有哪些机制？哪种速度最快？为什么？
- cap定理、base理论、Redis集群

redis扩容、缩容

- 扩容（先主后从）
 1. 先复制 `redis.conf` 文件，修改端口
 2. 启动新的redis（同时启动主从节点）
 3. 使用 `redis-trib.rb`（官方工具）`add-node`添加主节点到cluster
 4. 为主节点分配slots，redis内存重新分片（数据不丢失），使用 `reshard`
 5. 然后再是从节点，进入从节点的命令行，执行 `cluster replicate` 同步主节点
- 缩容（先从后主）
 1. 使用 `redis-trib.rb` 执行`del-node`删除从节点，这里就到此为止了
 2. 对于主节点，先使用 `reshard` 命令迁移slots数据插槽到其他不需要删除的主节点
 3. 然后再使用`del-node`删除主节点

TCP TIME_WAIT

- 什么情况出现timewait

当我们主动关闭一个socket链接的时候，尽管持有这个socket的进程已经关闭，但是这个socket（文件）还处于一个TIME_WAIT状态（客户端），再次启动这个进程是无法再去bind这socket，直到这个socket从TIME_WAIT转移到CLOSED状态。简而言之，客户端主动断开连接，客户端TIME_WAIT。

- 主要作用（为什么我们需要）

首先TIME_WAIT一定是从FIN_WAIT_2转移过来的。进入TIME_WAIT意味着这个tcp socket在主动关闭的时候，自己的FIN已经被对方确认，而且对方也发了一个FIN过来表示对方也没有什么数据要主动传输了。

因此，处于TIME_WAIT等待这么久时间，首先第一个作用：对方FIN的ACK确认（我们发的）有可能在传输过程中被丢弃。于是对方会重新发送FIN（简单来说，对方不知道我们已经知道他们不发数据了），停留在TIME_WAIT可以对重传的FIN进行确认。否则，对方将一直处于LAST_ACK状态。

第二个作用，等待较长时间，让在这个socket连接上传输的数据能够从整个网络中清空出去。（TCP超时重传导致数据包还在飞，对方会误认为我们还在传数据，直到收到ACK）

- 如何解决TIME_WAIT

首先，我们要了解，如何发现这个问题。

用 `netstat -at | grep "TIME_WAIT"` 统计发现，出问题的机器共计 `TIME_WAIT` 总数是多少。然后进一步分析出是哪一个PORT所产生的的是哪一个，然后针对性看看是什么任务所引起的。这里要注意的是：大量的`TIME_WAIT`会导致port对应的创建TCP连接会失败。

解决方案：

1. 修改系统配置

`tcp_max_tw_buckets`、`tcp_tw_recycle`、`tcp_tw_reuse` 这三个配置项。

将 `tcp_max_tw_buckets` 调大，这个方式治标不治本。

开启 `tcp_tw_recycle` 选项：在shell终端输入命令“`echo 1 > /proc/sys/net/ipv4/tcp_tw_recycle`”可以开启该配置。

开启 `tcp_tw_reuse` 选项：“`echo 1 > /proc/sys/net/ipv4/tcp_tw_reuse`”，与 `tcp_tw_recycle` 开启可能带来的副作用相比，貌似还没有发现 `tcp_tw_reuse` 引起的网络问题。

以上三个参数，还可以访问 `/etc/sysctl.conf` 文件修改

2. 修改应用程序

将TCP短连接修改为长连接。通常情况下，如果发起连接的目标也是自己可控制的服务器时，他们自己的TCP通信最好采用长连接，避免大量TCP短连接每次建立、销毁产生的各种开销。

通过 `getsockopt/setsockopt` 设置socket的SO_LINGER选项，关于SO_LINGER选项的设置方法，可以百度一下。

补充说明：

我们说`TIME_WAIT`过多可能引起无法对外建立新连接，其实有一个例外但比较常见的例子：S模块作为WebServer部署在服务器上，绑定本地某个端口；客户端对S发起短连接，每次交互完成后S主动断开连接。这样，当客户端并发访问次数很高时，S模块所在的机器可能会有大量处于`TIME_WAIT`状态的TCP连接。但由于服务器模块绑定了端口，故这种情况下，并不会引起由于`TIME_WAIT`过多导致无法建立新连接的问题。也就是说，前面所描述的情况建立在客户端出现大量的`TIME_WAIT`，本文讨论的情况，通常只会在每次由操作系统随机分配端口的程序运行的机器上出现。

CAP原理和最终一致性

CAP原理中，有三个要素：

- 一致性 (Consistency)
- 可用性 (Availability)
- 分区容忍性 (Partition tolerance)

CAP原理是指，这三个要素最多只能同时实现两点，不可能三者兼顾。因此，在进行分布式架构设计时，必须做出取舍。而对于分布式数据系统，分区容忍性是基本要求，否则就失去了价值。因此，设计分布式数据系统，就是在一致性和可用性之间做一个平衡。对于大多数web应用，其实并不需要强一致性，因此，牺牲一致性而换取高可用性，是目前多数分布式数据库产品的方向。

当然，牺牲一致性，并不是完全不管数据的一致性，否则数据是混乱的，那么系统可用性再高，分布式再好也没有了价值。牺牲一致性，只是不再要求关系型数据库中的强一致性，而是只要系统能达到最终一致性即可，考虑到客户体验，这个最终一致的时间窗口，要尽可能的对用户透明。也就是需要保障“用户感知到的一致性”。通常是通过数据的多份异步复制来实现系统的高可用和数据的最终一致性，“用户感知的一致性”的时间窗口则取决于数据复制到一致状态的时间。

最终一致性 (Eventually consistent)

对于一致性，可以分为从客户端和服务端两个不同的视角。从客户端来看，一致性主要指的是多并发访问时要更新过的数据如何获取的问题。从服务端来看，则是更新如何复制分布到整个系统，以保证数据最终一致。一致性是因为有并发读写才有的问题，因此在理解一致性的问题时，一定要注意结合考虑并发读写的场景。

从客户端角度，多进程并发访问时，更新过的数据在不同进程如何获取的不同策略，决定了不同的一致性。对于关系型数据库，要求更新过的数据能被后续的访问都能看到，这是强一致性。如果能容忍后续的部分或者全部访问不到，则是弱一致性。如果经过一段时间后要求能访问到更新后的数据，则是最终一致性。

最终一致性根据更新数据后各进程访问到数据的时间和方式不同，又可以区分为：

因果一致性。如果进程A通知进程B它已更新了一个数据项，那么进程B的后续访问将返回更新后的值，且一次写入将保证取代前一次写入。与进程A无因果关系的进程C的访问遵守一般的最终一致性规则。

“读己之所写 (read-your-writes) ”一致性。当进程A自己更新一个数据项之后，它总是访问到更新过的值，绝不会看到旧值，这是因果一致性模型的特例。

会话 (session) 一致性。这是上一个模型的实用版本，它把访问存储系统的进程放到会话的上下文中。只要会话还存在，系统就保证“读己之所写”一致性。如果由于某些失败情形令会话终止，就要建立新的会话，而且系统的保证不会延续到新的会话。

单调 (Monotonic) 读一致性。如果进程已经看到过数据对象的某个值，那么任何后续访问都不会返回在那个值之前的值。

单调写一致性。系统保证来自同一个进程的写操作顺序执行。要是系统不能保证这种程度的一致性，就非常难以编程了。

上述最终一致性的不同方式可以进行组合，例如单调读一致性和读己之所写一致性就可以组合实现。并且从实践的角度来看，这两者的组合，读取自己更新的数据，和一旦读取到最新的版本不会再读取旧版本，对于此架构上的程序开发来说，会少很多额外的烦恼。

从服务端角度，如何尽快将更新后的数据分布到整个系统，降低达到最终一致性的时间窗口，是提高系统的可用度和用户体验非常重要的方面。对于分布式数据系统：

- N -- 数据复制的份
- W -- 更新数据时需要保证写完成的节点数
- R -- 读取数据的时候需要读取的节点数

如果 $W+R>N$ ，写的节点和读的节点重叠，则是强一致性，例如对于典型的一主一备同步复制的关系型数据库， $N=2$, $W=2$, $R=1$ ，则不管读的是主库还是备库的数据，都是一致的。

如果 $W+R\leq N$ ，则是弱一致性。例如，对于一主一备异步复制的关系型数据库， $N=2$, $W=1$, $R=1$ ，则如果读的是备库，就可能无法读取主库已经更新过的数据，所以是弱一致性。

对于分布式系统，为了保证高可用性，一般设置 $N \geq 3$ ，不同的NWR组合，是在可用性和一致性之间取一个平衡，以适应不同的应用场景。

BitMap算法

简单来说就是bit位代替原来的数据的字面含义，以节省空间的一种做法。

布隆过滤器，可以使用这个做。

CDN技术

什么是CDN？

CDN的全称是内容分发网络。其目的是通过在现有Internet中增加一层新的网络架构，将网站的内容发布到最接近用户的网络“边缘”，使用户可以就近取得所需的内容，提高用户访问网站的响应速度。

CDN有别于镜像，因为它比镜像更智能，或者可以做这样一个比喻：

CDN=更智能的镜像+缓存+流量导流。因而，CDN可以明显提高Internet网络中信息流动的效率。从技术上全面解决由于网络带宽小，用户访问量大，网路分布不均等问题，提高用户访问网站的响应速度。

CND的类型特点：

CND的实现分为三类：镜像、高速缓存、专线。

镜像站点（Mirror Site），最常见的，它让内容直接发布，适用于静态和准动态的数据同步。但是购买和维护新服务器的费用较高，还必须在各个地区设置镜像服务器，配备专业技术人员进行管理和维护。对于大型网站来说，更新所用的带宽成本也大大提高了。

高速缓存，成本较低，适用于静态内容。Internet的统计表明，超过80%的用户经常访问的是20%的网站的内容，在这个规律下，缓存服务器可以处理大部分客户的静态请求，而原始的服务器只需处理约20%左右的缓存请求和动态请求，于是大大加快了客户请求的响应时间，并降低了原始服务器的负载。

CDN服务一般会在全国范围内的关键节点放置缓存服务器。

专线，让用户直接访问数据源，可以实现数据的动态同步。

编程部分

N个人中至少两个人同一天生日的概率

首先我们要看这个n多大，假如 $n=366$ ，那么假设全年365天，就算前面365个人的生日都不重复，那么第366个人的生日一定与前面365个人的生日当中的其中一个重复。所以当 $n \geq 366$ ，概率为1

再回头来看一下人数n在365这个范围以内的情况

我们先把这个问题换成一个反向问题，那么至少有两个人同一天生日的反面问题是：完全没有同一天生日的概率。这个概率记做 $P(0)$ ，没有人与第一个人同一天生日

那么全概率: $P(\text{A}) = P(0) + P(A)$

其中, $P(A)$ 为至少有两个人是同一天生日的概率, $P(A) = P(1) + P(2) + \dots + P(N-1)$

其中 $P(1)$ 表示1个人与其他人同一天生日, $P(N-1)$ 表示有N-1个人与剩下来的人同一天生日

所以: $P(A)$ 就是我们要求的目标概率: $P(A) = 1 - P(0)$, 这个问题就变成了求没有人和其他人同一天生日的问题。

那么, 第一个人的生日是随机的: $365 / 365$

第二个人, $364 / 365$

....

第N个人, $365 - (n-1) / 365$

把他们的概率乘积起来, 那么就是其全不相同的概率:

$$P(0) = (365 / 365) * (364 / 365) * \dots * (365 - n + 1) / 365$$

$$= (365 * 364 * \dots * 1 * 0 * \dots * (365 - n + 1)) / 365^n$$

所以可以看出来, 当 $n \geq 366$ 时

$P(0) = 0$, 这也验证了之前我们讨论的观点

当 $n < 365$ 时

比如 $n = 2$

$$P(A) = 1 - (365 * 364) / 365^2$$

$$= 1 - 0.997$$

$$= 0.003$$

以此类推, 当到达45个人时, 概率达到0.99

代码实现:

```
1 #include <iostream>
2
3 using namespace std;
4 double calc_nobody_same_pro(int n){
5     double p_n_same = 0.0;
6     if (n >= 366) {
7         cout << "输入人数超过366" << endl;
8         p_n_same = 0.0;
9     } else if (n <= 1) {
10        cout << "输入人数不足2个" << endl;
11        p_n_same = 1.0;
12    } else {
13        p_n_same = 1.0;
14        while (n > 0) {
15            p_n_same *= ((double)(365 - n + 1) / (double)365);
```

```

16           // cout << "计算第" << n << "个人之后的概率为: " << p_n_same <<
endl;
17           n--;
18       }
19   }
20
21   return p_n_same;
22 }
23
24 double least_2_same_birth_pro(int n) {
25     double nobody_birth_same = calc_nobody_same_pro(n);
26     cout << "没有人同一天生日的概率为: " << nobody_birth_same << endl;
27     return 1 - nobody_birth_same;
28 }
29
30
31 int main(int argc, char* argv[]) {
32     int count = atoi(argv[1]);
33     cout << "输入人数为: " << count << endl;
34     double p = least_2_same_birth_pro(count);
35     cout << "至少2个人同一天生日的概率为: " << p << endl;
36     return 0;
37 }
"same_birthday.cpp" 37L, 1038C

```

整数数组中找出前边的数都比自己小，后边的数都比自己大的数

比如整数数组[2, 3, 10, 5, 8, 11, 7, 4, 9, 18, 19]

思路上来说，就是将数组分为两半，所以，考虑使用2分查找的思路解决问题

首先，最简单的思路：

从左边开始遍历，找出左边最大的数，并且找出右边最小的数，跟当前数比较。

左边最大的数记为left_biggest

右边最小的数记为right_least

当前数记为now

如果， $\text{left_biggest} < \text{now} < \text{right_least}$ ，那么now这个数就是我们想要的。反之，

如果， $\text{left_biggest} > \text{now}$ 说明左边存在比当前数更大的，说明，当前位置不合适，起码left_biggest都比自己大了，说明当前位置并不是最佳位置。那么接下来判断，

如果，right_least是否比left_biggest更大，假如成立，那么可以认为，now右边的数，都比左边大。那么，当前位置只是不满足左边都比自己小，往右找一个数比左边最大数都大的数就可以了。

例子中：

第一个数3，左边最大数2，右边最小数4，比较看来，右边最小数，比左边最大数大，并且比3大，说明符合要求。

代码实现如下。

```
#include <iostream>
#include <stdlib.h>

using namespace std;

// 返回0表示没有比自己大的数,大于0表示该数的数值,且pos表示数在数组中的偏移量
// type = 0 表示求最小值
// type = 1 表示求最大值
void find_most_number(int type, int* sub_array, int count, int& most_number,
int& most_pos) {
    most_pos = 0;
    most_number = sub_array[0];
    for (int i = 0; i < count; i++) {
        if (type == 0) {
            if (sub_array[i] < most_number) {
                most_number = sub_array[i];
                most_pos = i;
            }
        } else if (type == 1) {
            if (sub_array[i] > most_number) {
                most_number = sub_array[i];
                most_pos = i;
            }
        } else {
            break;
        }
    }
    return;
}

#define MAX 1
#define MIN 0
// 遍历原始数组,找出所有的数
void find_mid_number(int* input_array, int count) {
    int now_pos = 1;
    if (count < 3) {
        cout << "输入数组长度小于3,不处理" << endl;
    }
    for (; now_pos < count - 1 && now_pos > 0; ) {
        int left_biggest = 0;
        int right_least = 0;
        int l_b_pos = 0;
        int r_l_pos = 0;
        int now = input_array[now_pos];
        // 找出左边最大值
        find_most_number(MAX, input_array, now_pos, left_biggest, l_b_pos);
        // 找出右边最小值
```

```

        find_most_number(MIN, input_array + now_pos + 1, count - now_pos - 1,
right_least, r_l_pos);
        r_l_pos = now_pos + r_l_pos + 1;
        cout << "当前位置[" << now_pos << ":" << now << "]," << \
            "左边最大值为[" << l_b_pos << ":" << left_biggest << "]," << \
            "右边最小值为[" << r_l_pos << ":" << right_least << "]" << endl;
        if (left_biggest < now && right_least > now) {
            cout << "当前值[" << now_pos << ":" << now << "]符合目标要求" <<
endl;
            now_pos += 1;
            cout << "跳转到[" << now_pos << ":" << input_array[now_pos] << "]"
<< endl;
        } else if (now > right_least) {
            cout << "当前值[" << now_pos << ":" << now << "]右边存在比自己小的值["
<< r_l_pos << ":" << right_least << "]" << endl;
            now_pos += 1;
            cout << "跳转到[" << now_pos << ":" << input_array[now_pos] << "]"
<< endl;
        } else if (now < left_biggest) {
            cout << "当前值[" << now_pos << ":" << now << "]左边存在比自己大的值["
<< l_b_pos << ":" << left_biggest << "]" << endl;
            now_pos += 1;
            cout << "跳转到[" << now_pos << ":" << input_array[now_pos] << "]"
<< endl;
        }
    }
    return;
}

int main(int argc, char* argv[]) {
    int number_count = argc - 1;
    cout << "输入了一个长度为：" << number_count << " 的数组" << endl;
    cout << "数组内容为：[ ";
    int* input_array = (int*)malloc(sizeof(int) * number_count);
    for (int i = 1; i < argc; i++) {
        int number = atoi(argv[i]);
        input_array[i - 1] = number;
        cout << number << " ";
    }
    cout << "]" << endl;
    find_mid_number(input_array, number_count);

    free(input_array);
}

```

字符串去空格

比方说一个字符串：“1 2 3 4 5 6”

要去除空格，一般来说会要求我们不开辟其他内存空间去做这个事情。

遇到第一个空格后，最简单的办法就是将所有字符串前移。以此类推，直到所有空格被处理完毕。

还有一种更为快速的办法，就是交换空格和非空格的值，依次类推，时间复杂度为n

```
1 #include <iostream>
2
3 void strip_space(char* str, int count) {
4     for (int i = 0; i < count; i++) {
5         // 找到第一个空格
6         if (str[i] == ' ') {
7             // 为这个空格找到第一个非空格的字符
8             bool find = false;
9             for (int j = i + 1; str[j] != '\0'; j++) {
10                 if (str[j] != ' ') {
11                     // 交换
12                     *(str + i) = str[j];
13                     *(str + j) = ' ';
14                     find = true;
15                     break;
16                 }
17             }
18             if (!find) {
19                 // 如果这个空格后面再也找不到非空格字符，结束
20                 str[i + 1] = '\0';
21                 break;
22             }
23         }
24     }
25 }
26
27 using namespace std;
28 int main(int argc, char* argv[]) {
29     char str[] = " a 000 1 1 4 5 . &";
30     cout << "输入字符串: " << str << endl;
31     strip_space(str, 18);
32     cout << "去处空格之后: " << str << endl;
33 }
```

- 手写链表翻转函数，要求是不用其他辅助存储

二叉树的遍历

首先了解二叉树的结构

二叉树遍历分为，前序遍历，中序遍历，后序遍历

当然遍历二叉树之前，我们需要先创建一个二叉树。

二×查找树，当前根节点的左边全部比根节点小，当前根节点的右边全部比根节点大。

比如给定顺序字符串：“G D W S X Y P Q”

按照先序构建，即，先构建根节点，再构建左子，再构建右子。

这里为了要建立一个明确的二叉树，这里我们选择按照二×搜索树前序遍历构造，思路：

第一个元素看做根节点，然后比较第二个元素与根节点元素大小。小于根节点，放在左子树；反之放在右子树。

- 计算二叉树高度
- 字符串转整形
- 写一个lru算法
- 手写算法：多文件内容排序，求根号算法
- 排序时间复杂度
- 红黑树、二叉树、冒泡排序、快速排序
- 给定一个有序的旋转数组，查找某个元素是否在数组内，eg：[4 5 6 1 2 3] target=5
- 给定一个物品在各个时间点的价格，求进行一次买卖之后最大获利；变形为：求多次买卖的最大获利
- 设计一个最优时空复杂度的位循环队列
- 系统位谋进程分配了4个页框，该进程已访问的页号序列为2, 0, 2, 9, 3, 4, 2, 8, 2, 4, 8, 4, 5.若进程要访问的下一页页号为7.依据LRU算法，应淘汰的页号是几号？
- 手写深拷贝实现代码
- 哈夫曼编码
- 有序数组，相同的元素最多保留2个，O (1) 的算法
- 手写代码，求链表倒数第K个元素
- 手写代码，求数组中出现次数超过一半的数
- 写一个随机数生成器，能以p概率生成0, 1-p概率生成1。利用这个生成器等概率生成0-6.
- 在32gb的内存里对1tb数据排序
- 简单动态规划，排楼梯问题
- 最长公共子序列
- 二维数组，回字形输出
- 一个数组，和sum。求出数组中所有可能的任意元素（一个元素只能用一次）的和等于sum的组合。
- 柱状图求最大矩形面积
- 大整数字符串相乘
- N*N地图随机放X个雷，打印地图，每格显示周边雷的个数
- 给一个有序数组，没有重复值，给一个数组sum，从数组中找出和为sum的两个数，打印出所有可能的组合
- K个排序的列表，融合成一个有序列表
- 两个字符串的公共子串（连续的）长度

C++11新特性

以下新特性基于C++03和C++11比较，所以，更多相比于C++98的特性是没有被体现出来的

Initializer list

```
//c++03
std::vector<int> v;
v.push_back(2);
v.push_back(3);

//c++ 11
std::vector<int> v{ 1,2,3 };
```

auto type

自动类型推导

```
std::vector<int> vec = { 1,2,3,4 };
//c++ 03
for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); it++) {
    std::cout << *it << std::endl;
}

//c++ 11
for (auto it = vec.begin(); it != vec.end(); it++) {
    std::cout << *it << std::endl;
}

auto a = 3 + 2;
auto s = "123";
```

使用auto需要注意以下两点：

- auto声明的变量必须要初始化，否则编译器不能判断变量的类型
- auto不能被声明为返回值，auto不能作为形参，auto能被修饰为模板参数

auto实际上是在编译时对变量进行了类型推导，所以不会对程序的运行效率造成不良影响。另外，auto并不会影响编译速度，因为编译时本来也要右侧推导然后判断是否与左侧是否匹配。

decltype

有时我们希望从表达式的类型推断出要定义的变量类型，但是不想用该表达式的值初始化变量（初始化可以用auto）。为了满足这一需求，C++11新标准引入了decltype类型说明符，它的作用是选择并返回操作数的数据类型，在此过程中，编译器分析表达式并得到它的类型，却不实际计算表达式的值。

foreach

```

std::vector<int> vec = { 1,2,3,4 };
//c++ 03
for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); it++) {
    std::cout << *it << std::endl;
}

//c++ 11
//read only (只读)
for (auto i : vec) {
    std::cout << i << std::endl;
}
//只读
for (const auto& i : vec) {
    std::cout << i << std::endl;
}
//changes the values in v (修改)
for (auto& i : vec) {
    i = 3;
}

```

nullptr

nullptr代替了C++03的enum，更安全

enum class

enum代替了C++03的enum，更安全

```

//c++ 03
enum ship {
    bulkship,
    tankership
};

//c++ 11
enum class ship1 {
    bulkship,
    tankership
};

```

override

override关键标识for virtual function（更加安全，直观）

```

class base {
public:

```

```

virtual void fun1(int);
virtual void fun2() const;
void fun3(int);
};

class son :public base {
//c++ 03 存在隐患
/*
void fun1(float); //不小心写错了参数,ok 编译通过, create a new func
void fun2(); //不小心少写了const,ok 编译通过, create a new func
void fun3();
*/

// but in c++ 11 更安全清晰
void fun1(float) override; //编译Error: no func to override
void fun2() override; //编译Error: no func to override
void fun3() override; //编译Error: no func to override
};

```

final

final关键标识，主要是class及virtual function

```

//this is means no class can be derived from CPoint2D
class CPoint2D final {
    //No class can override Draw
    virtual void Draw() final;
};

```

default

关键字default标识， compiler generated default constructor

强制编译器生成默认构造函数

- 编译器不再生成默认构造函数

```

class CPoint2D {
public:
    CPoint2D(double x_,double y_) {
        x = x_;
        y = y_;
    }
    double x;
    double y;
};

int main(){
    CPoint2D pt; //编译报错，不存在默认构造函数，因为编译器不再生成
}

```

- 通过default强制生成

```

class CPoint2D {
public:
    CPoint2D(double x_,double y_) {
        x = x_;
        y = y_;
    }
    CPoint2D() = default; //告诉编译器强制生成
    double x;
    double y;
};

int main(){
    CPoint2D pt; //ok
}

```

delete

关键标识delete，放在函数后面，表示函数不能被调用。

- 看以前的情形，构造函数只接受int，但是输入double也是可以的，因为会自动转换

```

class dog {
public:
    dog(int age_) {
        age = age_;
    }
    int age;
};

int main(){
    dog(2); //ok
    dog(4.5); //also ok, converted from double to int
}

```

- C++11解决方案

```

class dog {
public:
    dog(int age_) {
        age = age_;
    }
    dog(double) = delete;
    int age;
};

int main(){
    dog(2); //ok
    dog(4.5); //not ok, already deleted function
}

```

Lambda function

让编程更优雅。lambda表达式是匿名函数，可以认为是一个可执行体functor。具体语法：

[捕获区] (参数区) (代码区) ;

```

int main(){
    std::cout << [](int x, int y) {return x + y; }(3, 5) << std::endl;
}

```

经常使用std::sort对结构体匹配lambda匿名函数使用，比如点的数组按照x升序排列

```

class CPoint2D {
public:
    CPoint2D(double x1, double y1) {

```

```

        x = x1; y = y1;
    }
    double x;
    double y;
};

int main(){

    std::vector<CPoint2D> Pts{ {1,1},{3,3},{2,2} };
    std::sort(begin(Pts), end(Pts), [](const CPoint2D& pt1, const CPoint2D& pt2)
{
    return pt1.x < pt2.x;
});
}

```

Smart pointer

智能指针，让你从内存的泥潭中解放出来。一共有三种智能指针：

- `shared_ptr`，基于引用计数的智能指针，会统计当前有多少个对象同时拥有该内部指针；当引用计数降为0时，自动释放。使用`use_count()`函数可以看到指针对象被引用次数，存在最大的问题是**循环引用**：

```

class B;
class A
{
public:
    shared_ptr<B> m_b;
};

class B
{
public:
    shared_ptr<A> m_a;
};

int main()
{
{
    shared_ptr<A> a(new A); //new出来的A的引用计数此时为1
    shared_ptr<B> b(new B); //new出来的B的引用计数此时为1
    a->m_b = b; //B的引用计数增加为2
    b->m_a = a; //A的引用计数增加为2
}

//b先出作用域，B的引用计数减少为1，不为0，所以堆上的B空间没有被释放，且B持有的A也没有机会被析构，A的引用计数也完全没减少

//a后出作用域，同理A的引用计数减少为1，不为0，所以堆上A的空间也没有被释放

```

```
}
```

- `weak_ptr`, 基于引用计数的智能指针, 在面对循环引用的问题将无能为力。因此C++11还引入了`weak_ptr`与之配套, `weak_ptr`只引用, 不计数。当`weak_ptr`指向一块内存时, 对应的强引用指针引用计数并不加一。`weak_ptr`只能由`shared_ptr`或者其它的`weak_ptr`构造。`weak_ptr`和`shared_ptr`共享一个引用计数对象, 在引用计数对象上增加一个`weak_count`, 但不增加`ref_count`。引用计数对象当`ref_count`减至zero时会销毁其管理的资源, `weak_ptr`可以通过`ref_count`是否为0来判断指向的资源是否可用。当`ref_count`和`weak_count`都为0时引用计数对象会销毁其自身。

```
// 默认构造函数
weak_ptr<T> wp1;
// 拷贝构造
weak_ptr<T> wp2(p); // p为shared_ptr<T>
// =构造
weak_ptr<T> wp3 = wp2
```

- `unique_ptr`, 遵循独占语义的智能指针, 在任何时间点, 资源智能唯一地被一个`unique_ptr`所占有, 当其离开作用域时自动析构。资源所有权的转移只能通过`std::move()`而不能通过赋值符号(因为他将复制构造函数和=操作符设为私有)。`release()`函数会删除释放某个`unique_ptr`对象, 但是会将指针值返回。当离开作用域之前, 会自动调用析构函数, 并销毁对象。

还有就是C++98也引入了智能指针的概念: `auto_ptr`

tuple

元组

- `std::pair<std::string, int>`的扩展版本, 可以当做一个通用的结构体来使用
- `tuple`某些情况下会让代码的易读性变差
- 什么情况下使用`tuple`

std::pair<>的扩展版, 可多个参数

```
std::pair<int, std::string> p = std::make_pair(2, "hello");
//std::pair的扩展版
std::tuple<int, std::string, char> t(2, "foo", 'a');
auto t1 = std::make_tuple(0, "dog", 'b');
std::cout << std::get<0>(t) << std::endl;
std::cout << std::get<1>(t) << std::endl;
std::cout << std::get<2>(t) << std::endl;
```

易读性会变差 (一个使用结构体, 一个使用tuple)

结构体比较清晰 `p1.name` `p1.age`

tuple易读性差, 通过`std::get<0>(t2)` `std::get<1>(t2)`取值

这个语法, 会让你看起来并不知道0和1里面存储的是什么

```

struct person {
    std::string name;
    int age;
};

person p1;
std::tuple<std::string, int> t2;

//把上面的代码遮挡起来 tuple会导致代码的易读性降低
std::cout << p1.name << p1.age << std::endl;
std::cout << std::get<0>(t2) << std::get<1>(t2) << std::endl;
//what stored in position 0? what stored in position 1?

```

什么时候使用tuple

比作为函数返回值使用，我们一般不声明结构体

```

std::tuple<std::string, int> GetNameAge(){
    return std::make_tuple("lcl", 20);
}

```

当需要比较一个tuple大小的时候

```

//comparison of tuples
std::tuple<int, int, int> time1, time2;
if (time1 > time2) {
    //...
}

```

多索引map

```

std::map<std::tuple<int, char, double>, float> m;

```

thread

线程 #include <thread>

`std::thread` 可以和普通函数和lambda表达式搭配使用。它允许向线程执行函数传递任意多参数。

```

#include <thread>
void func() {
    // do some work here
}
int main() {
    std::thread thr(func);
    t.join();
    return 0;
}

```

函数 `func()` 在新起的线程中执行。调用 `join()` 函数是为了阻塞主线程，直到这个新起的线程执行完毕。线程函数的返回值都会被忽略，但线程函数可以接受任意数目的输入参数。

`std::thread` 的其他成员函数

- `joinable()`: 判断线程对象是否可以 `join`, 当线程对象被析构的时候, 如果该函数返回 `true`, 会导致 `std::terminate` 被调用
- `join()`: 阻塞当前进程 (通常是主线程), 等待创建新的线程执行完毕被操作系统回收
- `detach()`: 将线程分离, 从此线程对象受操作系统管辖

线程管理函数

除了 `std::thread` 的成员函数外, 在 `std::this_thread` 命名空间也定义了一些列函数用于管理当前线程。

- `get_id`: 返回当前线程id
- `yield`: 告知调度器运行其他线程, 可用于当前处于繁忙的等待状态。相当于主动让出剩下的执行时间, 具体的调度算法取决于实现
- `sleep_for`: 指定的一段时间内停止当前线程的执行
- `sleep_until`: 停止当前线程的执行直到指定的时间点

`unique_lock`解决死锁问题

通过两个`unique_lock`一起上锁, 可以解决死锁问题。

三种锁的创建类型

- `defer_lock_t` 不需要从属一个 `mutex`
- `try_to_lock_t` 在没有锁住的情况下尝试获取一个 `mutex`
- `adopt_lock_t` 假设调用当前锁的线程已经有一个从属 `mutex`

alias

```
using func = void (*) (int, int);
```

使用别名

`thread_local`

线程本地变量, 用于多线程时候, 为每一个线程定义一个线程变量。

这是一个关键字, 于此同时还有其他关键字:

- `auto`
- `register` 这个关键字支持到 C++17, 已经被 std 所废弃了
- `static`
- `extern`
- `mutable` 可变的变量修饰符, 突破 `const` 关键字的限制

Spring中的Bean默认是单例的, 非线程安全

Long long

新增加的数值变量类型，目标类型将会以最小不小于64bits的宽度存储

iterator

迭代器

- `std::begin`
- `std::end`

他们的原理都是基于 `initializer_list` 初始化列表这个类，简单来说，就是通过begin构造函数，将模板类型所规定的对象进行有序化。生成一个新的迭代器对象。

头条要点

1. C++依赖注入 (IOC) , spring也有类似机制, C++观察者模式
2. C++伪缓存
3. C++ `shared_ptr` 如何解决循环引用的问题, `weak_ptr`的实现方式
4. mysql主键索引和非主键索引, 为什么非主键索引更慢 (索引覆盖、索引组织表)
5. 如何用http实现rpc框架的通信 (都是7层协议) , RPC会自动打解数据包
6. GNU编译器底层对分支代码的优化

编程题：

64 * 64的棋盘，边长为1的正方形，如何通过对角线，画出一个面积为N (比如N为10) 的闭合图形。
大概有多少种解法。

简历需要美化

然后，需要每个项目梳理一到两个点出来，体现技术性的点。

写了这么多，其实真正在面试里面能用到的不是很多，高级别的面试，比较考验真实的解决问题的能力，很多遇到的问题都是面经里面不会提到的。这也是面试官考察一个人能力的重要指标。

shopee要点

快排（递归的栈空间限制），堆排序，动态规划

1. 什么时候会产生TIME_WAIT, 什么时候会产生CLOSE_WAIT
2. LRU缓存机制
3. C10K服务端模式
4. ES6搜索（这个不太记得了）
5. 进程切换原理
6. go语言协程实现原理
7. 分布式系统雪崩产生原因，如何防止雪崩
8. tcp拥塞算法

9. 快速排序算法实现原理（口述即可）
10. 微服务和soa的区别，微服务的技术难点
11. 谈谈redis的使用，redis的数据主从同步
12. C++虚函数实现原理
13. C++多态实现原理
14. go语言锁的使用，go语言协程启动匿名函数传入参数的考察（面试官写的代码，让指出错误）
15. 分布式缓存有哪些技术框架
16. 如何解决线上问题，步骤和解决办法（运维知识）
17. 除开gdb以外，有哪些服务端调试手段
18. hash索引的实现原理

mysql的innodb和MyIsam两种引擎的区别

- InnoDB支持主外键、事务
- InnoDB是行锁，操作时候只锁一行数据，适合高并发
- InnoDB不仅缓存索引，还缓存真实数据； MyISAM只缓存索引
- InnoDB需要表空间大
- InnoDB关注事务， MyISAM关注性能（查）

mysql数据库索引的实现原理

1. C++编码，运算符重载，漏掉一个编程题
22. 编程题，使用python或go实现，1234567，变成字符串“1,124,567”
23. class A{
 int func();
 virtual int func2();
 int a;
};
24. 一台机器只有1G物理内存，是否程序可以获取到2G，为什么。
25. 描述，操作系统虚拟内存是如何实现的
26. vector扩容的原理
27. 什么环节会发送RST包

wb要点

用过什么框架

spring

轻量级IOC和AOP容器框架， IOC的三种配置方式：

- xml配置
- 基于注解的配置
- 基于java的配置

主要由以下几个模块组成：

- spring core 核心类库， 提供IOC服务
- spring context 提供框架式的Bean访问方式， 以及企业级功能（JNDI）
- spring aop AOP服务， 把应用业务逻辑和系统服务分开
- spring dao （Data Access Object） 对jdbc的抽象， 简化了数据访问异常的处理（针对系统而言）
- spring ORM （Object Relation Mapping） 对现有的ORM框架的支持（针对开发而言）
- spring web 提供了基本的面向web的综合特性， 例如多放文件上传
- spring mvc 提供面向web应用的model-view-controller实现

Akka

actor模型， 使用actor将事务执行流程化， 互不影响。actor类似go语言携程。

Java设计模式

- 微服务架构六种常用设计模式

代理设计模式

聚合设计模式

链条设计模式

聚合链条设计模式

数据共享设计模式

异步消息设计模式

- spring框架中都用到了哪些设计模式

代理模式： 在AOP和remoting中被用的比较多

单例模式： 在spring配置文件中定义的bean默认为单例模式

模板方法： 用来解决代码重复的问题

前端控制器： spring提供了dispatcherServlet来对请求进行分发

视图帮助（view helper）： spring提供一系列的jsp标签， 高效宏来辅助将分散的代码整合在视图里

依赖注入： 贯穿于BeanFactory/ApplicationContext接口的核心理念

工厂模式： BeanFactory用来创建对象的实例（简单工厂模式）

- 设计模式分类

23种经典设计模式都有哪些，如何分类？

面向对象设计原则	创建型模式 5+1	结构型模式 7	行为型模式 11
<ul style="list-style-type: none">• 单一职责原则• 开闭原则• 里氏替换原则• 依赖注入原则• 接口分离原则• 迪米特原则• 组合/聚合复用原则	<ul style="list-style-type: none">• 简单工厂模式• 工厂方法模式• 抽象工厂模式• 创建者模式<ul style="list-style-type: none">• 原型模式• 单例模式	<ul style="list-style-type: none">• 外观模式• 适配器模式• 适配器模式• 代理模式• 装饰模式• 桥接模式• 组合模式• 享元模式	<ul style="list-style-type: none">• 模板方法模式• 观察者模式• 状态模式• 策略模式• 职责链模式• 命令模式• 访问者模式• 调停者模式• 备忘录模式• 迭代器模式• 解释器模式

适配器模式

作为两个不兼容的接口之间的桥梁。它结合了两个独立接口的功能。适配器模式提供对接口的转换。如果你的客户端使用某些接口，但是你有另外一些接口，你就可以写一个适配器来连接这些接口。

适配器模式就像旅行插座转换器（图1）、Type-c转VGA转接口（图4）一样。

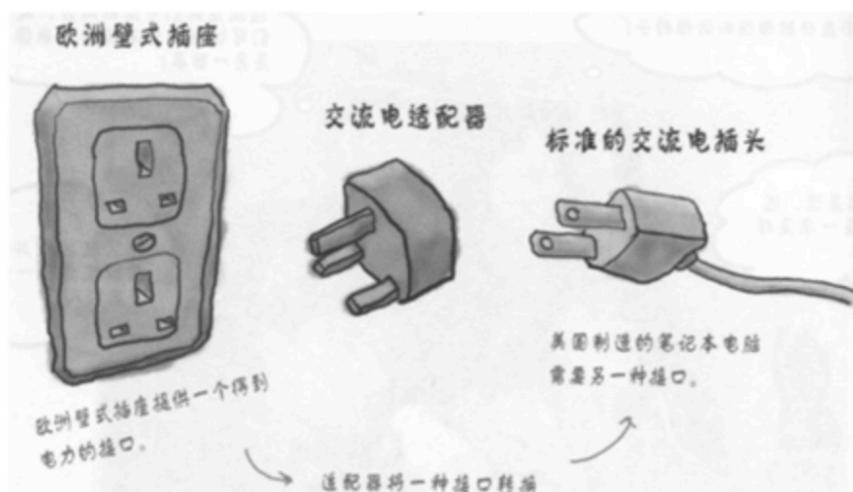


图1. 图片来源网络

举例：

```
//file 为已定义好的文件流
FileInputStream fileInput = new FileInputStream(file);
InputStreamReader inputStreamReader = new InputStreamReader(fileInput);
```

以上就是适配器模式的体现，`FileInputStream`是字节流，而并没有字符流读取字符的一些API，因此通过`InputStreamReader`将其转为Reader子类，因此有了可以操作文本的文件方法。

- 类的适配器模式

原理：通过继承特性来实现适配器功能

- 对象的适配器模式

原理：通过组合方式来实现适配器功能

- 接口的适配器模式

原理：借助抽象类来实现适配器功能

适配器模式与装饰器模式有什么区别

虽然适配器模式和装饰器模式的结构类似，但是每种模式的出现意图不同。适配器模式被用于桥接两个接口，而装饰模式的目的是在不修改类的情况下给类增加新的功能。

装饰者模式举例：

```
BufferedReader bufferedReader=new BufferedReader(inputStreamReader);
```

构造了缓冲字符流，将FileInputStream字节流包装为BufferedReader过程就是装饰的过程，刚开始的字节流FileInputStream只有一个字节方法，包装为inputStreamReader后，就有了读取一个字符的功能在包装为BufferedReader后，就拥有了read一行字符的功能。

数据库索引（MySQL）

- 索引的原理

1. B+树索引 二叉搜索树这棵树是平衡二叉树 N叉树为了减少树高

如果语句是 `select * from T where ID = 500`，即逐渐查询方式，则只需要搜索ID 这颗B+树

如果语句是 `select * from where k = 5`，即普通索引查询方式，则需要先搜索k索引树，ID的值为500，再到ID索引树搜索一次。这个过程称回表

也就是说，基于非主键索引的查询需要多扫描一颗索引树，因此，我们在应用中应该尽量使用主键查询。

2. 一个数据页满了，按照B+ Tree算法，新增加一个数据页，叫做页分裂，会导致性能下降。空间利用率大概50%

当相邻的两个数据页利用率很低的时候会做数据页合并，合并的过程是分裂过程的逆过程

3. 从性能和存储空间方面考量，自增主键往往是更合理的选择
4. 显然，主键长度越小，普通索引的叶子节点就越小，普通索引占用的空间也就越小。

因此，自增主键比较合适

- 索引分类

普通索引：没有任何限制

唯一索引：索引列的值必须唯一，但允许有控制（unique index）

主键索引：特殊的唯一索引，不允许有空值，建表自动创建主键索引

组合索引：多字段联合创建索引，组合索引大部分情况比单列索引块（比如我们刚好查的就是建立了组合索引的那几列）

- 如何创建索引

1. 使用create index
2. 使用create table 创建表的时候添加索引

- 为什么要创建索引

1. 毋庸置疑提高插入和查询的速度

- 索引有什么缺点

1. 降低更新速度（不仅要更新数据，还要更新索引）
2. 占磁盘空间、内存空间

- 索引覆盖

```
create table T(
    ID int primary key,
    k int Not NULL default 0,
    s varchar(16) not null default '',
    index k(k)
) engine = InnoDB;
```

`select * from T where k between 3 and 5` 这种查询K的索引搜索到主键，然后搜索主键的索引拿到具体的信息有回表。

`select ID from T where k between 3 and 5` 这时只需要查ID的值，而ID的值已经在k索引树上，因此可以直接提供查询结果，不需要回表，由于覆盖索引可以减少树的搜索次数，显著提升查询性能，所以使用覆盖索引是一个常用的性能优化手段。

如何索引覆盖？简单来说创建索引的时候，覆盖掉所有可能查询到的“列”名。

如何查看当前select操作是否使用了索引覆盖？很简单，当查询非主键列的时候，使用EXPLAIN查看语句的Extra列可以看到“Using Index”的信息。则使用了索引覆盖。

- 索引优化

1. 减少主键字段长度（非主键索引字段也是）
2. 建立联合索引
3. 覆盖索引对InnoDB尤其有用，因为InnoDB使用聚集索引组织数据，如果二级索引包含查询所需的数据，就不再需要在聚集索引中查找了

此外，经常使用explain分析索引性能：

使用：explain + sql语句

作用：

1. 表的读取顺序
2. 数据读取操作的操作类型
3. 哪些索引可以使用

- 4. 哪些索引被实际使用
- 5. 表之间的引用
- 6. 每张表有多少行被优化器查询

- 索引数据结构

B+-Tree (不是BTree) 、Hash索引、full-text全文索引、R-Tree索引

B+树与B树的不同在于：

1. 所有关键字存储在叶子节点，非叶子节点不存储真正的data
2. 为所有叶子节点增加了一个链指针

- 索引使用

如果你非要使用 like，那么 like "%aaa%" 不会使用索引，而 like "aaa%" 会

不要在单列上进行运算，否则索引失效

不要使用NOT IN和<>操作

如果查询的两个表大小相当，那么用in和exists差别不大。

如果两个表中一个较小，一个是大表，则子查询表大的用exists，子查询表小的用in；

无论哪个表大，用not exists都比not in要快。not in因为内外表都进行全表扫描，没有用到索引；而not exists的子查询依然能用到表上的索引。

案例分析 (join连接索引分析)

例：

```
select id from A where c1 = 1 and c2 > 1 order by v1 desc limit 1;
```

建立联合索引 (c1, c2, v1)，但explain时候发现type是range，extra中使用using filesort这需要优化；

产生原因：按照BTree索引工作原理，先排序c1，如果c1相同，排序c2，c2相同再排序v1，当c2字段在联合索引中处于中间位置，因此c2 > 1条件是一个范围值 (range)，MySQL无法利用索引在对后面的v1部分进行索引。所以建立 (c1, v1) 解决这个问题。

双表分析

左连接，加在右表的索引，右连接加在左表的索引；

Left Join条件用于确定如何从右表搜索行，左边数据一定有，所以右边数据一定要建索引。

总结

尽可能减少Join语句的NestedLoop的循环总次数，永远用小结果集驱动大的结果集；

优先优化NestedLoop的内层循环；

保证Join语句中被驱动表上Join条件字段已经被索引；

- 索引失效

1. 最好全值匹配
2. 最左前缀法则：如果索引了多列，查询从索引的最左前列开始，且不能跳过索引中的列；
3. 不在索引列上做任何操作（计算，函数，类型转换），会导致索引失效，而转向全表扫描
4. 存储引擎不能使用索引中范围条件右边的列，即范围之后全失效；
5. 尽量使用覆盖索引，只访问索引的查询（索引列和查询列一致），减少 `select *`；
6. MySQL在使用 `!=` 的时候，无法使用索引会导致全表扫描
7. `is null`, `is not null` 也无法使用索引；
8. `like` 以通配符开头 ('%aaa') 索引会失效，变成全表扫描
9. 字符串不加单引号 ('')，索引会失效；
10. 少用 `or`，用它来连接时候会索引失效

其他调优渠道

- 查询截取分析

1. 观察，查看慢SQL情况；
2. 开启查询日志，设置阈值；
3. `explain` 分析；
4. `show profile` 查看执行细节和生命周期情况

- `show profile`

```
set profiling = on ; # 开启
show profiles; # 查看执行过的sql
SHOW PROFILE cpu,block io FOR QUERY 87; # 查看这个执行sql的生命周期相关信息
```

关键信息解读：

`converting HEAP to MyISAM`：查询结果太大，内存不够用了往磁盘搬

`Creating tmp table`：创建临时表，拷贝数据到临时表，用完再删除

`Copying to tmp table on disk`：把内存中临时表复制到磁盘，危险！

`locked`：锁住

MySQL数据引擎区别

- MyISAM

保存成文件形式，适合跨平台

锁粒度是表级别

支持全文索引

非事务安全

- InnoDB

事务安全

支持行级锁

不支持全文索引

比MyISAM复杂

- 使用场景

1. MyISAM适合执行大量的select操作，可以选择MyISAM
2. 如果需要大量insert和update操作，可以选择InnoDB

MySQL为啥使用B+树

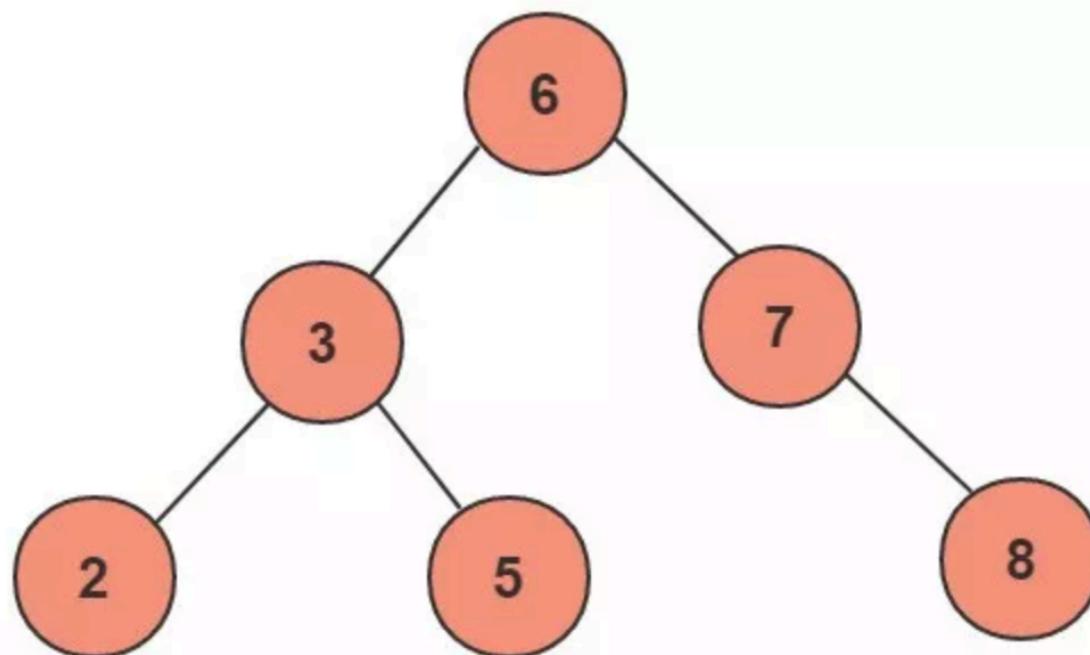
在进一步分析为什么MySQL数据库索引选择使用B+树之前，我相信很多小伙伴对数据结构中的树还是有些许模糊的，因此我们由浅入深一步步探讨树的演进过程，在一步步引出B树以及为什么MySQL数据库索引选择使用B+树！

一、二叉查找树

(1) 二叉树简介

二叉查找树也称为有序二叉查找树，满足二叉查找树的一般性质，是指一颗树具有如下性质：

1. 任意节点左子树不为空，则左子树的值均小于根节点的值；
2. 任意节点右子树不为空，则右子树的值均大于根节点的值；
3. 任意节点的左右子树也分别是二叉查找树；
4. 没有键值相等的节点



上图为一个普通的二叉查找树，按照中序遍历的方式可以从小到大的排序输出：2、3、5、6、7、8.

对上述二叉树进行查找，如查键值为5的记录，先找到根，其键值是6，6大于5，因此查找6的左子树，找到3；而5大于3，再找其右子树；一共找了3次。如果按2、3、5、6、7、8的顺序来找同样需求3次。用同样的方法在查找键值为8的这个记录，这次用来3次查找，而顺序查找需要6次。计算平均查找次数的；顺序查找的平均查找次数为 $(1+2+3+4+5+6) / 6 = 3.3$ 次，二叉查找树的平均查找次数为 $(3+3+3+2+2+1) / 6 = 2.3$ 次。二叉查找树的平均查找速度比顺序查找来得更快。

(2) 局限性及应用

一个二叉查找树是由n个节点随机构成，所以，对于某些情况，二叉查找树会退化成一个有n个节点的线性链。

如果我们的根节点选择是最小或者最大的数，那么二叉查找树就完全退化成了线性结构。平均查找次数为 $(1+2+3+4+5+5) / 6 = 3.16$ 次，和顺序查找差不多。显然这个二叉树的查询效率就很低，因此若想最大性能的构造一个二叉查找树，需要这个二叉树是平衡的（这里的平衡从一个显著的特点可以看出，这一棵树的高度比上一个树的高度要大，在相同节点的情况下也就是不平衡），从而引出一个新的定义-平衡二叉树AVL。

二、AVL树

(1) 简介

AVL树是带有平衡条件的二叉查找树，一般是用平衡因子差值判断是否平衡，并通过旋转来实现平衡，左右子树树高不超过1，和红黑树相比，它是严格的平衡二叉树，平衡条件必须满足（所有节点的左右子树高度相差不超过1）。

不管我们是执行插入还是删除操作，只要不满足上面的条件，就要通过旋转来保持平衡，而旋转是非常耗时的，由此我们可以知道AVL树适合用于插入删除次数比较少，但查找多的情况。

从上面是一个普通的平衡二叉树，这张图我们可以看出，任意节点的左右子树的平衡因子差值都不会大于1。

(2) 局限性

由于维护这种高度平衡所付出的代价比从中获得的效率收益还大，故而实际的应用不多，更多的地方是用追求局部而不是非常严格整体平衡的红黑树。当然，如果应用场景中对插入、删除元素不频繁，只是对查找要求较高，那么AVL还是较优于红黑树。

(3) 应用

Windows NT内核中广泛存在；

三、红黑树

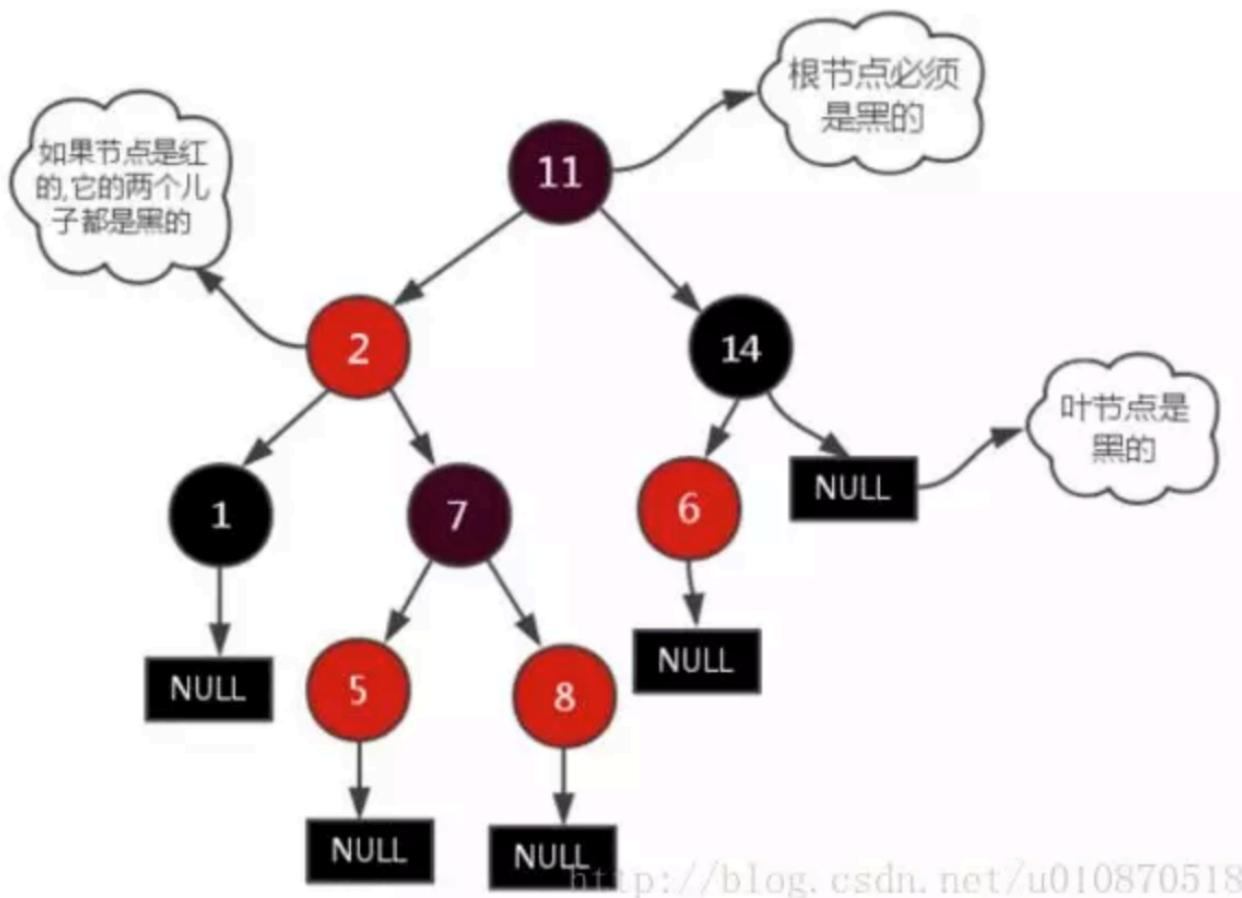
(1) 简介

一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是red或black。通过对任何一条从根到叶子的路径上各个节点着色方式的限制，红黑树确保没有一条路径会比其他路径长出两倍。它是一种弱平衡二叉树（由于是弱平衡，可以推出，相同的节点情况下，AVL树的高度低于红黑树），相对于要求严格的AVL树来说，它的旋转次数变少，所以对于搜索、插入、删除操作多的情况下，我们就用红黑树。

(2) 性质

1. 每个节点非红即黑；
2. 根节点是黑的；

3. 每个叶节点（叶节点即树末端NULL指针或NULL节点）都是黑的；
4. 如果一个节点是红的，那么它的两个儿子都是黑的；
5. 对于任意节点而言，其到叶子节点NULL指针，每条路径都包含相同数目的黑节点；
6. 每条路径都包含相同的黑节点；



(3) 应用

1. 广泛用于C++的STL中，Map和Set都是用红黑树实现的；
2. 著名的Linux进程调度Completely Fair Scheduler，用红黑树管理进程控制块，进程的虚拟内存区域都存储在一棵红黑树上，每个虚拟地址区域都对应红黑树的一个节点，左指针指向相邻的地址虚拟存储区域，右指针指向相邻的高地址虚拟地址空间；
3. IO多路复用epoll的实现采用红黑树组织管理sockfd，以支持快速的增删改查；
4. Nginx中用红黑树管理Timer，因为红黑树是有序的，可以很快的得到距离当前最小的定时器；
5. Java中TreeMap的实现；

四、B/B+树

说了上述的三种树：二叉查找树、AVL和红黑树，似乎我们还没有摸到MySQL为什么要使用B+树作为索引的实现。

首先，我们就先探讨一下什么是B树。

(1) 简介

我们在MySQL中的数据一般是放在磁盘中，读取数据的时候肯定会有访问磁盘的操作，磁盘中有两个机械运动的部分，分别是盘片旋转和磁臂移动。盘片旋转就是我们市面上所提到的多少转每分钟，而磁盘移动则是在盘片旋转到指定位置以后，移动磁臂后开始进行数据的读写。那么这就存在一个定位到磁盘中的块的过程，而定位是磁盘的存取中花费时间比较大的一块，毕竟机械运动花费的时间，要远远大于

电子运动的时间。当大规模数据存储到磁盘中的时候，显然定位是一个非常花费时间的过程，但是我们可以通过B树进行优化，提高磁盘读取时定位的效率。

为什么B类树可以进行优化呢？我们可以根据B类树的特点，构造一个多阶的B类树，然后在尽量多的在节点上存储相关的信息，保证层数尽量的少，以便后面我们可以更快的找到信息，磁盘的IO操作也少一些，而且B类树是平衡树，每个节点到叶子节点的高度都是相同，这也保证了每个查询时稳定的。

总的来说，B/B+树是为了磁盘或其他存储设备而设计的一种平衡多路查找树（相对于二叉，B树每个内节点有多个分支），与红黑树相比，在相同的节点的情况下，一颗B/B+树的高度远远小于红黑树的高度（在下面B/B+树的性能分析中会提到）。B/B+树上操作的时间通常由存取磁盘的时间和CPU计算时间这两部分构成，而CPU的速度非常快，所以B树的操作效率取决于访问磁盘的次数，关键字总数相同的情况下B树的高度越小，磁盘IO所花时间越少。

(2) B树的性质

1. 定义任意非叶子节点最多只有M个儿子，且M>2；
2. 根节点的儿子数为[2, M]；
3. 除跟几点以外的非叶子节点的儿子数为[m/2, M]；
4. 每个节点存放至少M/2 - 1（取上整）和至多M-1个关键字；（至少2个关键字）
5. 非叶子节点的关键字个数=指向儿子的指针个数-1；
6. 非叶子节点的关键字：K[1], K[2], ..., K[M-1]；且K[i] < K[i+1]；
7. 非叶子节点的指针：P[1], P[2], ..., P[M]；其中P[1]指向关键字小于K[1]的子树，P[M]指向关键字大于K[M-1]的子树，其他P[i]指向关键字属于(K[i-1], K[i])的子树；
8. 所有叶子节点位于同一层；

这里只是一个简单的B树，在实际中B树节点中关键字很多的，上面的图中比如35节点，35代表一个key（索引），而小黑块代表的是这个key所指向的内容在内存中实际的存储位置，是一个指针。

五、B+树

(1) 简介

B+树是应文件系统所需而产生的一种B树的变形树（文件的目录一级一级索引，只有最底层的叶子节点（文件）保存数据），非叶子节点只保存索引，不保存实际的数据，数据都保存在叶子节点中，这不就是文件系统文件的查找吗？

我们就举个文件查找的例子：有三个文件夹a、b、c

a包含b，b包含c

一个文件file.c

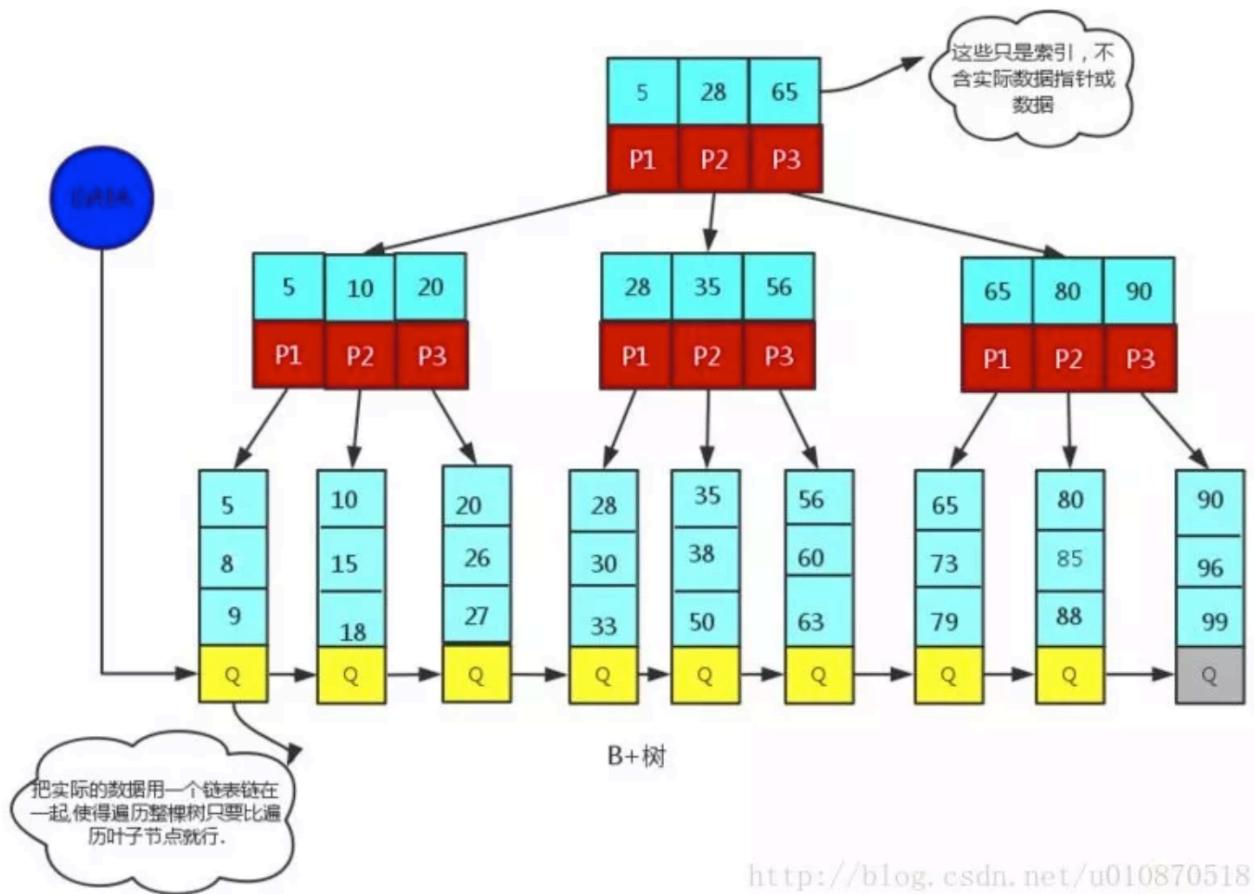
a、b、c就是索引（存储在非叶子节点），a、b、c只要找到的file.c的key，而实际的数据file.c存储在叶子节点上。

所有的非叶子节点都可以看成索引部分。

(2) B+树的性质（下面提到的都是和B树不同的性质）

1. 非叶子节点的子树指针与关键字个数相同；
2. 非叶子节点的子树指针p[i]，指向关键字值属于[k[i], k[i+1]]的子树。（B树是开区间，也就是说B树不允许关键字重复，B+树允许重复）；
3. 为所有叶子节点增加一个链指针；
4. 所有关键字都在叶子节点出现（稠密索引）。（且链表中的关键字恰好是有序的）；

5. 非叶子节点相当于是叶子节点的索引（稀疏索引），叶子节点相当于是存储（关键字）数据的数据层；
6. 更适合于文件系统；



非叶子节点（比如：5, 28, 65）只是一个key（索引），实际的数据存在叶子节点上（5, 8, 9）才是真正 的数据或指向真实数据的指针。

(3) 应用

1、B和B+树主要用在文件系统以及数据库做索引，比如MySQL；

七、为什么说B+树比B树更合适数据库索引？

1、**B+树的磁盘读写代价更低：**B+树的内部节点并没有指向关键字具体信息的指针，因此其内部节点相对B树更小，如果把所有同一内部节点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多，一次性读入内存的需要查找的关键字也就越多，相对IO读写次数就降低了。

2、**B+树的查询效率更加稳定：**由于非终结点并不是最终指向文件内容的节点，而只是叶子节点中关键字的索引。所以任何关键字的查找必须走一条从根节点到叶子节点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

3、由于B+树的数据都存储在叶子节点中，分支结点均为索引，方便扫库，只需要扫一遍叶子节点即可，但是B树因为其分支节点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引。

PS：数据库索引采用B+树的主要原因是：B树在提高了IO性能的同时并没有解决元素遍历的效率低下的问题，正是为了解决这个问题，B+树应运而生。B+树只需要去遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作或者效率太低。

所以，综合来说：

MySQL使用B+树肯定是为了提升查找效率。

所以要对查找的方式进行优化，熟悉的二分查找，二叉树可以把数据提升到 $O(\log(n, 2))$ ，查询的瓶颈在于树的深度，最坏的情况要查找到二叉树的最深层。

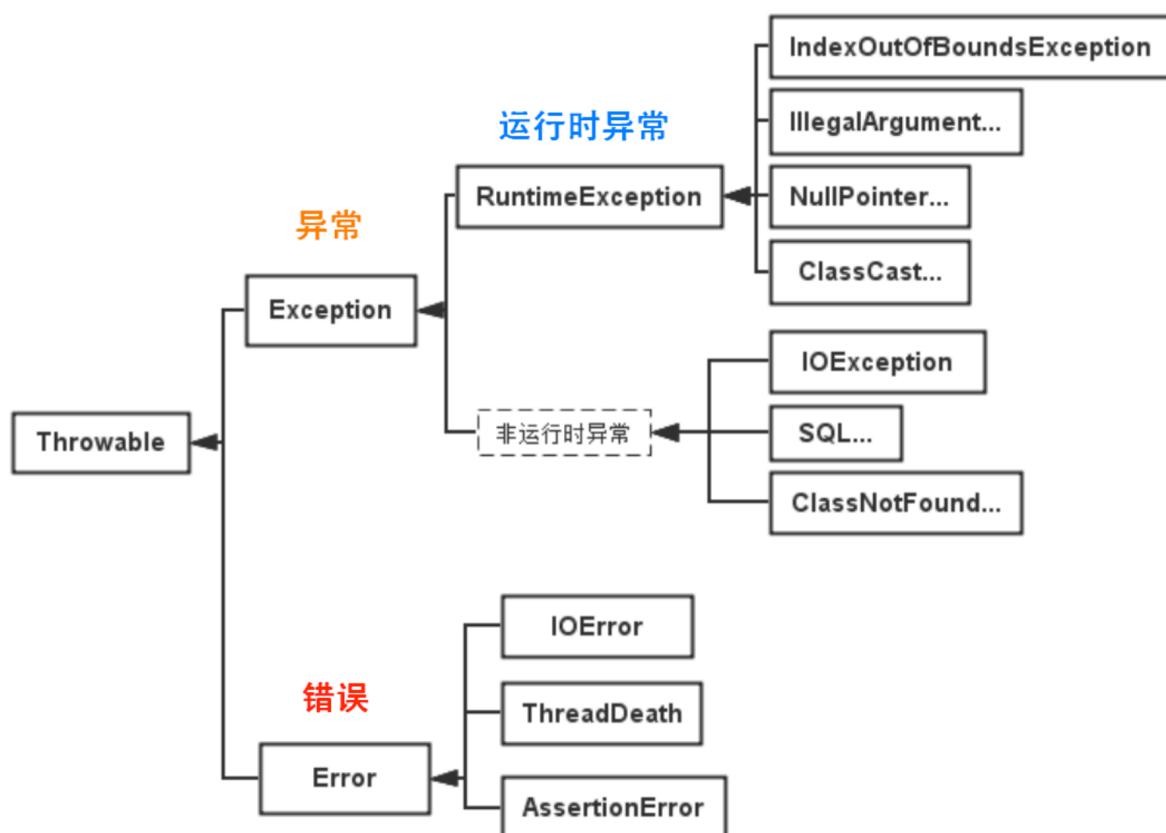
由于，没查找深一层，就要访问更深一层的索引文件。在多达数G的索引文件中，这将是很大的开销。所以，尽量把数据结构设计的更为“矮胖”一点，就可以减少访问的层数。

在众多解决方案中，B/B+树很好的合适。

MySQL中MyISAM和InnoDB都是采用的B+树结构。不同的是前者是非聚集索引，后者主键是聚集索引，所谓聚集索引是物理地址连续存放的索引，在取区间的时候，查找速度非常快，但同样的，插入的速度也会受到影响而降低。聚集索引的物理位置使用链表来进行存储。

Java异常处理

异常分类：



两个子类区别：

1. **Error**: 程序不应该捕捉的错误，应该交由JVM来处理。一般可能指非常重大的错误。这个错误我们一般获取不到，也无法处理！
2. **Exception**: 程序中应该要捕获的错误。这个异常类及它的子类是我们需要学习获取要处理的。
 - (1) `RuntimeException`: 运行时异常，也叫未检查异常，是Exception的子类，但不需捕捉的异常超类，但是实际发生异常时，还是会导至程序停止运行的，只是编译时没有报错而已。比如除数为零，数组空指针等等，这些都是在运行之后才会报错。此类异常，可以处理也可以不处理，并且可以避免。
 - (2) 在Exception的所有子类中除了`RuntimeException`类和它的子类，其他类都叫做非运行时异常，或者叫已检查异常，通常被定义为`Checked`类，是必须要处理可能出现的异常，否则编译就报错了。`Checked`类主要包含：IO类和SQL类的异常情况，这些在使用时经常要先处理异常（使用`throws`或`try catch`捕获）。

java几种常见的异常：

运行时异常：

- 1, `java.lang.ArrayIndexOutOfBoundsException` 数组索引越界异常。当对数组的索引值为负数或大于等于数组大小时抛出。
- 2, `ArithmaticException` 算术错误情形，如以零作除数，算术条件异常。
- 3 `java.lang.SecurityException` 安全性异常
- 4, `IllegalArgumentException` 方法接收到非法参数，非法参数异常！
- 5, `java.lang.ArrayStoreException` 数组中包含不兼容的值抛出的异常
- 6, `java.lang.NegativeArraySizeException` 数组长度为负异常
- 7 `java.lang.ClassNotFoundException` 找不到类异常。当应用试图根据字符串形式的类名构造类，而在遍历CLASSPATH之后找不到对应名称的class文件时，抛出该异常。
- 8 `java.lang.NullPointerException` 空指针异常。当应用试图在要求使用对象的地方使用了null时，抛出该异常。譬如：调用null对象的实例方法、访问null对象的属性、计算null对象的长度、使用throw语句抛出null等等。
- 9, `java.lang.NumberFormatException` (数字格式转换异常)
- 10, `java.lang.ClassCastException`(强制类型转换异常)

IOException

- 1, `IOException` 操作输入流和输出流时可能出现的异常
- 2, `EOFException` 文件已结束异常
- 3, `FileNotFoundException` 文件未找到异常

如何处理异常

`try`、`catch`和`finally`三个关键字，这三个关键字可以组合使用：

```
try--catch  
try--catch--finally  
try--finally
```

`throws` 关键字表示，该方法不处理异常，而由系统自动将所捕获的异常信息抛给上级调用方法。

`throw` 关键字作用是手工跑出异常类的实例化对象。`throw` 通常和 `throws` 联合使用，抛出的是程序中已经产生的异常类实例。

自定义异常方法，继承自 `Exception` 类。

抽象类和接口区别

1. 一个类可以实现多个接口，但是只可以继承一个抽象类
2. 抽象类可以包含具体方法，接口的所有方法都是抽象的
3. 抽象类可以声明和使用字段，接口则不能，但接口可以创建静态的final常量
4. 接口的方法都是public的，抽象类的方法可以是public、protected、private或者默认的package
5. 抽象类可以定义构造函数，接口却不能

总结来说，java的接口有点像C++的完全由纯虚函数构成的抽象类

除了单例模式你在生产中还用过什么模式

一般来说，依赖注入，工厂模式，装饰模式或者观察者模式。

- 抽象工厂模式

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

其实简而言之，就是在原来工厂模式的基础上，在原工厂类的基础上，添加一个抽象方法（接口），这个方法被用来在新添加的工厂类继承旧的工厂类的时候，实现该方法以实现新加入的功能。

- 观察者模式

包含四个角色：

1. 抽象被观察者角色：也就是一个抽象主题，它把所有对观察者对象的引用保存在一个集合中，每个主题都可以有任意数量的观察者。抽象主题提供一个接口，可以增加和删除观察者角色。一般用一个抽象类和接口来实现。
2. 抽象观察者角色：为所有的具体观察者定义一个接口，在得到主题通知时更新自己。
3. 具体被观察者角色：也就是一个具体的主题，在集体主题的内部状态改变时，所有登记过的观察者发出通知。
4. 具体观察者角色：实现抽象观察者角色所需要的更新接口，一边使本身的状态与制图的状态相协调。

简单来说就是设置一个接口，包含 `注册观察者`、`移除观察者`、`通知观察者` 这三个能力的接口。

然后定义一个被观察者继承这个接口。

定义一个观察者接口，观察者接口，最基本的功能就是收到 `通知消息` 并作出动作。这个抽象接口，必须被 `通知观察者` 调用

定义一个观察者继承观察者接口。

假如问到代码习惯

- 开闭原则

开闭原则就是说对扩展开放、对修改关闭。为了使程序的扩展更好，易于维护和升级。想要达到这样效果，我们需要使用接口和抽象类。

- 里氏代换原则

面向对象设计的基本原则之一。简单来说，任何基类可以出现的地方，子类一定可以出现（替换），而不影响软件的功能。

- 依赖倒置原则

这是开闭原则的基础，针对接口编程，依赖于抽象而不是依赖于具体

- 接口隔离原则

使用多个隔离的接口，比使用单个接口要好。降低耦合度，降低依赖。这个有点像Unix哲学，只做一件事情。

- 最少知道原则

一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。还是降低耦合，降低依赖

- 合成复用原则

尽量使用合成、聚合的方式（C++叫组合），而不是使用继承

为什么要使用单例

1. 核心交易，不允许许多实例运行
2. 省去new操作，降低GC压力，以及系统开销

Spring AOP

AOP是Spring框架面向切面的编程思想，AOP采用一种称为“横切”的技术，将涉及多业务流程的通用功能抽取并单独封装，形成独立的“切面”，在核实的实际将这些切面横向切入到业务流程指定的位置中。

例如：

在一个业务系统中，用户登录是基础功能，凡是涉及到用户的业务流程都要求用户进行系统登录。如果把用户登录功能代码写入到每个业务流程中，会造成代码冗余，维护也非常麻烦，当需要修改用户登录功能时，就需要修改每个业务流程的用户登录代码，这种处理方式显然是不可取的。比较好的做法是把用户登录功能抽取出来，形成独立的模块，当业务流程需要用户登录时，系统自动把登录功能切入到业务流程中。

AOP也是需要xml配置文件配置的（aop.xml），此外还需要引入jar包（aop包）

读写分离也可以用AOP来做

AOP相关概念：

- (1)横切关注点：对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点
- (2)Aspect(切面)：通常是一个类，里面可以定义切入点和通知
- (3)JointPoint(连接点)：程序执行过程中明确的点，一般是方法的调用。被拦截到的点，因为Spring只支持方法类型的连接点，所以在Spring中连接点指的就是被拦截到的方法，实际上连接点还可

以是字段或者构造器

(4)Advice(通知):AOP在特定的切入点上执行的增强处理，有before(前置),after(后置),afterReturning(最终),afterThrowing(异常),around(环绕)

(5)Pointcut(切入点):就是带有通知的连接点，在程序中主要体现为书写切入点表达式

(6)weave(织入): 将切面应用到目标对象并导致代理对象创建的过程

(7)introduction(引入): 在不修改代码的前提下，引入可以在运行期为类动态地添加一些方法或字段

(8)AOP代理(AOP Proxy): AOP框架创建的对象，代理就是目标对象的加强。Spring中的AOP代理可以使JDK动态代理，也可以是CGLIB代理，前者基于接口，后者基于子类

(9)目标对象 (Target Object) : 包含连接点的对象。也被称作被通知或被代理对象。POJO

AOP使用场景:

Authentication 权限

Caching 缓存

Context passing 内容传递

Error handling 错误处理

Lazy loading 懒加载

Debugging 调试

logging, tracing, profiling and monitoring 记录跟踪 优化 校准

Performance optimization 性能优化

Persistence 持久化

Resource pooling 资源池

Synchronization 同步

Transactions 事务

Spring ORM

对象关系映射 (Object Relation Mapping)，用于实现面向对象编程语言里不同类型系统的数据之间转换。简单点来说，就是将关系型数据库的表变为Java代码里面的类对象，便于程序操作，而且，对不同数据库是可以适配和移植的。

如何保证kafka的消息机制

Producer生产者，只负责数据生产，生产者的代码可以集成到任务系统中

数据的分发策略由producer决定，默认是defaultPartition Utils.abs(key.hashCode) % numPartitions

Broker: 当前服务器上的kafka进程，只管数据存储，不管是谁生产，不管是谁消费。在集群中每个broker都有一个唯一brokerid，不得重复

Topic: 目标发送的目的地，这是一个逻辑概念，落到磁盘上是一个partition的目录。partition的目录中有多个segment组合 (index, log)

一个topic对应多个partition[0,1,2,3]，一个partition对应多个segment组合。一个segment有默认的大小是1G。

每个partition可以设置多个副本，会从所有的副本中选取一个leader出来。所有读写操作都是通过leader来进行的。这是特别要强调的一点，mysql做主从一般都是为了读写分离。

ConsumerGroup: 数据消费组, ConsumerGroup可以有多个, 每个ConsumerGroup消费的数据都是一样的。

可以把多个consumer线程划分为一个组, 组里面所有成员共同消费一个topic的数据, 组员之间不能重复消费。

如何保证数据完全生产?

ACK机制: broker表示发来的数据已经确认接收无误, 表示数据已经保存到磁盘。

0: 不等待broker返回确认消息

1: 等待topic中某个partition leader保存成功的状态反馈

-1: 等待topic中某个partition所有副本都保存成功的状态反馈

broker如何保存数据?

在理论环境下, broker按照顺序读写的机制, 可以每秒保存600M的数据。主要通过pagecache机制, 尽可能的利用当前物理机器上的空闲内存来做缓存。

当前topic所属的broker, 必定有一个该topic的partition, partition是一个磁盘目录。partition的目录中有多个segment组合 (index, log)

consumerGroup的组员和partition之间如何做负载均衡?

最好是一一对应, 一个partition对应一个consumer

如果consumer的数量过多, 必然会产生空闲的consumer。

如何保证kafka消费者消费数据是全局有序的?

伪命题

如果要全局有序, 必须保证生产有序, 存储有序, 消费有序。

由于生产可以集群部署, 存储可以分片, 消费可以设置为一个comsumerGroup, 要保证全局有序, 就需要保证每个环节都有序。

只有一个可能, 就是一个生产者, 一个partition, 一个消费者。

为什么ack=1, 消息也会丢失?

kafka消息丢失的几种情况

网络异常:

acks设置为0时, 不和kafka集群进行消息接收确认, 当网络发生异常等情况时, 存在消息丢失的可能

客户端异常:

异步发送时, 消息并没有发送至kafka集群, 而是在client端按一定规则缓存并批量发送。在这期间, 如果客户端发生死机等情况, 都会导致消息的丢失。

缓冲区满了:

异步发送时, client端缓存的消息超出了缓冲池的大小, 也存在消息丢失的可能。

Leader副本异常:

ack=1的情况下，producer只要收到分区leader成功写入的通知就会认为消息发送成功了。如果leader成功写入后，还没来得及把数据同步到follower节点就挂了，这时候消息就丢失了。

日常应用中，我们需要结合自身的应用场景来选择不同的配置。

想要更高的吞吐量就设置：异步，ack=0；

想要不丢失消息数据就选：同步，ack=-1策略；

操作系统线程、go语言携程、actor区别

传统多数流行的语言并发是基于多线程之间的共享内存，使用同步方法防止写争夺，Actors使用消息模型，每个Actor在同一时间处理最多一个消息，可以发送消息给其他Actor，保证了单独写原则。从而巧妙避免了多线程写争夺。和共享数据方式相比，消息传递机制最大的优点就是不会产生数据竞争状态。

实现消息传递有两种常见的类型：

基于channel（golang为典型代表）的消息传递（CSP模型--Communicating Sequential Processes）

基于Actor（erlang为代表）的消息传递

二者的格言都是：“Don't communicate by sharing memory, share memory by communicating”

那么他们有什么区别呢？

- **Actor**

在Actor模型中，主角是Actor，类似一种worker，Actor彼此之间直接发送消息，不需要经过什么中介，消息是异步发送和处理的。

Actor模型描述了一组为了避免并发编程的常见问题公理：

1. 所有Actor状态是Actor本地的，外部无法访问
2. Actor必须只有通过消息传递进行通信
3. 一个Actor可以响应消息：推出新Actor，改变其内部状态，或将消息发送到一个或多个其他参与者。
4. Actor可能会堵塞自己，但Actor不应该堵塞它运行的线程。

- **Channel**

Channel模型中，worker之间不直接彼此联系，而是通过不同channel进行消息发布和侦听。消息的发送者和接受者之间通过Channel松耦合，发送者不知道自己消息被哪个接收者消费了，接受者也不知道是哪个发送者发送的消息。

Go语言的CSP模型是由携程Goroutine与通道Channel实现：

1. Go携程goroutine：是一种轻量线程，它不是操作系统的线程，而是将一个操作系统线程分段使用，通过调度器实现协作式调度。是一种绿色线程，微线程，它与Coroutine携程也有区别，能够在发现堵塞后启动新的微线程。
2. 通道channel：类似Unix的Pipe，用于携程之间通讯和同步。携程之间虽然解耦，但是他们和Channel有着耦合。

Actor之间是直接通讯的，而CSP（go携程）是通过Channel通讯，在耦合度上两者是有区别的，后者更加松耦合。

分布式系统生成唯一ID

- 数据库自增长字段
- UUID全球唯一
- UUID变种
- Redis生成ID。依赖的是redis的单线程操作。
- 雪花算法
- 利用zookeeper生成唯一ID。主要是其znode数据版本来生成序列号
- snowflake算法。一个long型ID。其核心思想是：使用41bit作为毫秒数，10bit作为机器的ID（5个bit是数据中心，5个bit的机器ID），12bit作为毫秒内的流水号（意味着每个节点在没毫秒可以产生4096个ID），最后还有一个符号位，永远是0。

无锁实现多线程实现一个唯一数

乐观锁

flink处理来自两个数据源的数据

A服务处理修改密码的请求，B服务处理放款的请求，两个数据流都输出服务处理数据到两个不同的kafka topic，但是修改密码5分钟之内不允许执行放款操作。请问怎么实现？

分别将A、B两个服务的kafka数据，接到flink的两个数据流里面，B数据stream按照时间窗口五分钟join A数据stream，如果发现五分钟内A进行过修改密码的工作，B stream业务层可以置位redis状态。B服务接收请求的时候判断redis状态，如果状态不对，则服务返回不允许操作。

不过感觉这个问题，不应该使用这种flink的操作去做，应该从服务层面去做。

比如，A收到请求后，redis记录当前操作timestamp，B请求后查询redis根据请求类型以实现频控。

乐观锁

- 什么场景下需要使用锁？

在多节点部署或者多线程执行时，同一个时间可能有多个线程更新相同数据，产生冲突，这就是并发问题。这样情况下会出现以下问题：

更新丢失：一个事物更新数据后，被另一个更新数据的事物覆盖

脏读：一个事务读取另一个事务为提交的数据，即为脏读。

其次还有幻读。

针对并发引入并发控制机制，即加锁。

加锁的谜底是在同一个时间只有一个事务在更新数据，通过锁独占数据的修改权。

- 锁的实现方式

1. 悲观锁，前提是，一定会有并发抢占资源，强行独占资源，在整个数据处理过程中，将数据处于锁定状态。
2. 乐观锁，一段执行逻辑加上乐观锁，不同线程同时执行时，可以同时进入执行，在最后更新数据的时候要检查这些数据是否被其他线程修改了（版本和执行初是否相同），没有修改则进行更新，否则放弃本次操作。

当然，还有其他的锁机制，暂时不多介绍，着重于乐观锁的实现。

乐观锁，使用版本标识来确定读到的数据与提交时的数据是否一致。提交后修改版本标识，不一致时可以采取丢弃和再次尝试的策略。

```
package com.webank.test;

public class OptimiseLock {
    public static int value = 0;

    /**
     * A线程要执行的方法
     * @param Avalue
     * @param i
     * @throws InterruptedException
     */
    public static void invoke(int Avalue, String i) throws
InterruptedException {
        Thread.sleep(1000L);
        // 判断value版本
        if (Avalue != value) {
            System.out.println(Avalue + ":" + value + "A版本不一致，不执行");
            value--;
        } else {
            Avalue++;
            value = Avalue;
            System.out.println(i + ":" + value);
        }
    }

    public static void invoke2(int Bvalue, String i) throws
InterruptedException {
        Thread.sleep(1000L);
        if (Bvalue != value) {
            System.out.println(Bvalue + ":" + value + "B版本不一致，不执行");
        } else {
            System.out.println("B:利用value运算, value=" + Bvalue);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        new Thread(new Runnable() {
```

```

        public void run() {
            try {
                for (int i = 0; i < 3; i++) {
                    int Avalue = OptimiseLock.value;      // A获取的value
                    OptimiseLock.invoke(Avalue, "A");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }).start();

    new Thread(new Runnable() {
        public void run() {
            try {
                for (int i = 0; i < 3; i++) {
                    int BValue = OptimiseLock.value;      // B获取的value
                    OptimiseLock.invoke2(BValue, "B");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }).start();
}
}

```

CAS (Compare And Swap)

只是一种建立在cpu级别的原子指令，能够保证数据的比较和交换的原子性，常见的编译语言都支持。

- C++11

C++11给我们带来的Atomic一系列原子操作类，它们提供的方法能保证具有原子性。这些方法是不可再分的，获取这些变量的值时，永远获得修改前的值或修改后的值，不会获得修改过程中的中间数值。

这些类都禁用了拷贝构造函数，原因是原子读和原子写是2个独立原子操作，无法保证2个独立的操作加在一起仍然保证原子性。

这些类中，最简单的是atomic_flag（其实和atomic相似），它只有test_and_set()和clear()方法。其中，test_and_set会检查变量的值是否为false，如果为false则把值改为true。

除了atomic_flag外，其他类型可以通过atomic获得。atomic提供了常见且容易理解的方法：

1. store
2. load
3. exchange
4. compare_exchange_weak

5. compare_exchange_strong

store是原子写操作，而load则是对应的原子读操作。

exchange允许2个数值进行交换，并保证整个过程是原子的。

而**compare_exchange_weak**和**compare_exchange_strong**则是著名的**CAS (compare and set)**。参数会要求在这里传入期待的数值和新的数值。它们对比变量的值和期待的值是否一致，如果是，则替换为用户指定的一个新的数值。如果不是，则将变量的值和期待的值交换。

weak版本的CAS允许偶然出乎意料的返回（比如在字段值和期待值一样的时候却返回了false），不过在一些循环算法中，这是可以接受的。通常它比起strong有更高的性能。

无锁队列

使用c++11实现无锁队列，无锁队列的基本原理就是CAS（交换和保存原子操作）。

以下是一个用c++实现的无锁队列版本。

```
EnQueue(x) //进队列
{
    //准备新加入的结点数据
    q = newrecord();
    q->value = x;
    q->next = NULL;

    do{
        p = tail; //取链表尾指针的快照
        }while( CAS(p->next, NULL, q) != TRUE); //如果没有把结点链上，再试

        CAS(tail, p, q); //置尾结点
}
```

我们可以看到，程序中的那个do-while的重试循环。很可能我在队列尾加入结点时，别的线程已经加入成功了，于是我的CAS返回了false，于是程序再试，直到试成功为止。这个很像我们的抢电话热线不停地重拨的情况。

你会看到，为什么我们的“置尾结点”的操作不用判断是否成功，因为：

1. 如果有一个线程T1，它的while中的CAS如果成功的话，那么其他所有的随后线程的CAS都会失败，然后就会再循环。
2. 此时，如果T1线程还没有更新tail指针，其它的线程继续失败，因为tail->next不是NULL了（因为首先拿到的tail快照没有更新）。
3. 直到T1线程更新完tail指针，于是其他的线程中的某个线程就可以得到新的tail指针，继续往下走。

细心的同学会发现一个问题——如果T1线程在利用**CAS**更新tail指针之前，线程就挂掉了，那么其他线程就进入死循环了。

下面是改良版的无锁队列：

```
EnQueue(x) //进队列改良版
```

```

{
    q = newrecord();
    q->value = x;
    q->next = NULL;

    p = tail;
    oldp = p
    do{
        while(p->next != NULL)
            p = p->next;
    }while( CAS(p.next, NULL, q) != TRUE); //如果没有把结点链上，再试

    CAS(tail, oldp, q); //置尾结点
}

```

我们让每个线程，自己fetch指针p到链表尾。但是这样的fetch会影响性能。而通常实际情况看下来，99.9%的情况不会有线程停转的情况，所以，更好的做法是，你可以结合上述的两个版本，如果retry的次数超过一个值的话（比如说3次），那么就自己fetch指针。

CAS的ABA问题

描述：

1. 进程P1在共享变量中读到值为A
2. P1被抢占了，进程P2执行
3. P2把共享变量的值从A改成了B，再改回到A，此时P1抢占
4. P1回来看到共享变量里的值没有被改变，于是继续执行

这里的核心问题就是：

1. 不违反CAS使用原则，同一时间，只有一个任务能够操作变量
2. P2拿着变量做了其他事情，P1并不知道

TC要点

C++的stl里面队列和链表区别

队列是一个逻辑层面的数据结构，链表是一个实现层面的数据结构

队列可以用链表实现，也可以用数组实现

C++ STL中也有queue，属于container adapter。它内部可以使用序列式容器来实现，比如list。
这样queue的各项操作，实际上就是转换为对内部容器的对应调用。

go语言问了interface使用场景及与java接口区别

go语言的interface概念相对于C++中的基类，通过interface来实现多态功能

C++中，当需要实现多态功能时，步骤是首先定义一个基类，该基类使用虚函数或者纯虚函数抽象了所有子类会用到的共同的最基本的成员函数，之后子类继承该基类，然后每个派生类自定义自己的虚函数实现。最后在使用基类指针或者引用的地方传入派生类，程序根据具体传入的对象来调用对象的函数。

Go中，定义一个interface类型，该类型说明了它有哪些方法，这就完成了类似C++中的基类定义，然后在其他的函数中，将该interface类型作为函数的形参，任意一个实现了interface类型的实参都作为该interface的实例对象。interface类型和作为interface类型的实参对象之间就相当于存在继承关系，或者说交实现接口（Java说法），但这种继承关系（实现接口）是隐式的（自动的），也即我们无需主动说明（显示implements实现）该实参类型是某interface类型的派生类。

Go语言实现interface中的所有方法，即该类、对象就实现了该接口。

interface可以被任意对象实现，一个类型、对象也可以实现多个interface

interface的变量可以持有任意实现该interface类型的对象

interface可以是一个空接口，不包含任何method，正因为如此，所有类型都实现了空interface，空interface在我们需要存储任意类型的数据的时候相当有用，因为它可以存储任意类型的数值，有点类似于C语言的 void* 类型。

可以使用switch测试、Comma-ok断言判断空interface里面变量实际保存了的是哪个类型的对象。

Java接口允许有默认抽象方法和属性，但是属性一定是final、public、static的。

Java也可以被多继承，但是接口只可以作用于实现它的类。我们可以使用接口该性的引用指向一个实现了该接口的对象，并且调用这个接口中的方法。这个跟C++的纯虚函数类很像。

Java接口可以继承，并且可以多继承。但是不能被实例化。

携程调度算法（Go语言）

进程、线程、携程的理解

- 进程：是计算机进行资源分配的最小单位，上下文切换开销很大，进程之间数据是隔离的
- 线程：是计算机调度的最小单位，是计算机运行的基本单元（个人理解，可以把线程理解是共享堆的进程），上下文调度相对于进程来说要轻量（因为线程之间调度不需要切换进程资源）
- 携程：是用户态的一种轻量的线程，携程的调度完全由用户控制，不需要内核参与，所以速度很快

go携程模型和特点

携程可以相互协作、抢占，go底层是基于GMP模型实现的携程调度

M：系统真正的线程。m想办法找到可以运行的g，并且运行它

P：G的集合，有个g的链表，一般会和m绑定。m不断运行p上面的g，并且p会存储m上线程的相关信息（可以理解为调度器？）

G：go携程的真正的实体。代表正在要执行的携程的逻辑和其要运行的上下文

GMP是怎么运行的

1. 启动携程：go程序内部都维护了一个gfree的list。每次我启动一个新的携程的时候，程序不会马上就new一个g对象。而是先去gfree里找，如果gfree里面有线程的g对象，就会直接拿来用。如果gfree里面没有才会去new一个对象，所有的g运行完了都不会直接销毁。而是放到了gfree里。相

当于gfree是一个g的对象池。总而言之go底层g是复用的。

2. 携程绑定p：当携程被启动后，就会优先依附于启动自己的m所绑定的p上，这样很大程度上避免了m与m之间的竞争。当m上的p的g已经超过了承受范围，就会放到全局的g队列里。与m绑定的p最多只有256个g。
3. m的运行：m就是不断地运行依附于自己p上的g。为了让全局g队列里能够被调度到。m没运行61个本地g就会去全局g队列里拿一个g运行。当m本地的p没有可以运行g可以运行的时候也会去全局队列里面拿g运行。当全局队列也没有g的时候。就会去别的m绑定的p上偷一半g来运行。如果这个时候m还是拿不到g。那么这个m就会陷入睡眠。值得注意的是睡眠m也是不会被销毁的。防止被频繁的创建销毁。

go携程调度

首先我们需要知道go的协作在什么情况下会被调度（就是被抢占）

1. 当携程运行时间过长的时候，那么这个携程在调用非内联函数的时候会被调度：

正是因为有这个特性，go的携程才具有了抢占性，才让goroutine不再是单纯的携程。首先程序会在启动的时候运行一个sysmon线程，这个线程负责整个的调度具有上帝视角。sysmon会给所有的p都记录一个已运行的g的数量。每当p运行一个g后都会+1。如果这个变量没有+1，就说明这个p一直在运行同一个g任务。如果超过了10ms，就给这个g的栈信息上加个标记。当有了这个标记，这个g在**遇到非内联函数的时候**就会把自己中断掉，移到当前p的队列尾。

2. 当携程陷入一个系统调用，没有返回的时候会被调度

当m进行系统调用的时候，这个m会被陷入到内核态，毫无疑问的被阻塞了。还是sysmon会定期检测，当一个p上的g执行时间过长，且这个g还是在进行系统调用。那么sysmon会把p与当前m解绑，然后唤醒一个之前没事做的睡眠的m进行绑定运行。如果没有睡眠的m就会重新创建一个m来绑定，之前的m运行着进行系统调用的g返回后就会尝试找没有m的p。如果没有找到，m就会把g放到全局对了，然后自己进入睡眠。

3. 当携程被channel阻塞时会被调度

这个不需要多说，当g被channel阻塞后，这个g会被放到相应的wait队列，该g的状态由_Gruning变成_Gwaiting，g会被别的g的channel操作唤醒，那么这个g就会尝试加入那个g所在的p的runnext，如果没有成功，就会加入本地p队列，全局队列，等待被m运行。

4. 当携程进行网络io阻塞的时候会被调度

网络io虽然也算是系统调用，但是go原生的网络包底层全是有epoll类似的驱动的，并不走之前系统调用那一套。sysmon线程会进行epoll_wait，然后对程序每个socket进行标记，哪些socket是可写的，哪些socket是可读的，然后尝试唤醒因为socket不可写或者不可读的g，而g在对socket进行读写的时候就判断sysmon有没有对其标记过，如果没有标记过就进入wait状态，如果标记过了就说明可用就继续操作，操作完了，再把这个标记去除，等待下次标记。

首先我们需要知道go的协作在什么情况会被调度（就是别的携程抢占）

进程调度算法 (Linux Process)

进程调度CPU资源分为四种情况：

- 进程从运行态切换到等待状态时（如：IO请求，wait()调用一遍等待一个子进程结束）
- 进程从运行态切换到就绪态（如：出现了中断）
- 进程从等待状态切换到就绪态（如：IO完成）

- 进程终止

以上第一、四种情况下发生的进程调度，为非抢占的或者协作的；否则调度就是抢占的。

为了实现进程模型，操作系统维护着一张表格（一个结构数组），即进程表（process table）

每个进程占用一个进程表项，该表项包含了进程状态的重要信息，包括程序计数器、堆栈指针、内存分配状态、所打开文件的状态、账号和调度信息。

CPU利用率=1 - pⁿ

一个进程等待IO操作的时间与其停留在内存中的时间比为p

内存中同时有n个进程

先到先服务调度（FCFS）

先请求CPU的进程首先分配到CPU。FCFS策略可以通过FIFO队列容易的实现，当一个进程进入就绪队列时，他的PCB被链接到队列尾部。当CPU空闲时，他会分配给位于就绪队列头部的进程，并将这个进程从队列中移除。

进程	执行时间
P1	24
P2	3
P3	3

如果上面三个进程顺序到达，且按FCFS顺序处理，那么平均等待时间为 $(0+24+27) / 3 = 17$ (ms)

如果是：

进程	执行时间
P2	3
P3	3
P1	24

那额平均等待时间就是 $(6+0+3) / 3 = 3$ (ms)，所以进程的CPU执行时间变化波动很大的话，那么采用这种调度策略的平均等待时间变化也会很大。

最短作业优先调度（SJF）

该算法将每个进程与下次CPU执行的长度关联起来。当CPU空闲时，他会被赋给具有最短CPU执行的进程。如果两个进程具有同样长度的CPU执行，可以由FCFS来处理。

SJF算法可以是抢占的也可以是非抢占的。新进程的下次CPU执行，与当前运行进程的尚未完成的CPU执行相比，可能还要小，抢占SJF算法会抢占当前运行进程，而非抢占SJF算法会允许当前运行进程以先完成CPU执行。抢占SJF调度有时称为**最短剩余时间优先调度**

优先级调度

SJF算法就是通过优先级调度算法的一个特例，每个进程都有一个优先级与之关联，而具有最高优先级的进程会被分配到CPU，具有相同优先级的进程按照FCFS顺序调整。

优先级调度算法的一个主要问题是**无穷阻塞、饥饿**。即让某个低优先级进程无穷等待CPU。常见的解决方式是**老化**，老化逐渐增加在系统中等待很长时间的进程的优先级。

轮转调度（RR）

轮转调度算法是专门为分时系统设计的，类似于FCFS调度，并增加了抢占以切换进程。将一个较小的时间单元定义为时间量或时间片。时间片的大小通常为10ms-100ms。就绪队列作为一个循环队列，CPU调度程序循环整个就绪队列，为每个进程分配不超过一个时间片的CPU。

在RR调度算法中，没有进程会被连续分配超过一个时间片的CPU（除非是唯一可运行的进程），如果进程的CPU执行超过一个时间片，那么该进程会被抢占，并被放回就绪队列中，所以RR调度算法是抢占的。RR调度算法的时间片切换，是需要时间的，一般少于10ms。

多级队列调度

多级队列调度算法就是将就绪队列分成多个单独队列，根据进程属性（如内存大小，进程优先级等）将每个进程永久分到一个队列中，每个队列有自己的调度算法。因此，队列之间也存在调度，通常采用固定优先级抢占调度。

多级反馈队列调度

在多级队列调度算法中，进程被永久的分配到一个队列，这种设置的优点是调度开销小，缺点是不够灵活。多级反馈队列调度算法允许进程在队列之间迁移。

例如：根据不同CPU执行的特点来区分进程，如果进程使用过多的CPU时间，那么将会被迁移到更低优先级的队列。这种方案将IO密集型和交互进程放在更高优先级队列上。

通常情况下，多级反馈调度程序可由下列参数来进行定义：

队列的数量

每个队列的调度算法

用以确定合适升级到更高优先级队列的方法

用以确定何时降级到更低优先级队列的方法

用以确定进程在需要服务时将会进入哪个队列中的方法

线程调度算法（Linux Thread）

线程模型：

- 多个线程共享同一个地址空间和其他资源，比如共享全局变量
- 进程中的不同线程不像不同进程之间那样存在很大的独立性
- 严格来说，同一时刻只有一个线程占用CPU，但高速切换给人带来并行运行的假象
- 线程和进程一样，也具有三种状态，运行态、就绪态、阻塞态，并且转化关系也一样
- 每个线程都有其自己的堆栈，其中有一帧，供各个被调用但还没返回的过程中使用，在该帧存放了相应过程的局部变量以及过程调用完成之后使用的返回地址
- 常见的线程调用：`thread_yield`，它允许线程自动放弃CPU从而让另一个线程运行，因为不同于

进程，线程无法利用时钟中断强制让出CPU

调用堆栈的弊端：

调用堆栈不能跨线程，每一个线程都有自己的堆栈，所以不能进行线程的异步调用。

工作线程发生了错误，但是其自身却无法恢复。比如一个由bug引起的内部错误导致了线程的关闭，那么会导致一个问题，到底应该由谁来重启线程并且保存线程之前的状态呢？

在支持线程的操作系统上，内核级线程才是操作系统所调度的。用户级线程是由线程库来进行管理的，而内核并不知道他们。用户级线程为了运行在CPU上最终映射到相关的内核级线程上，这种映射不是直接的，可能采用轻量级进程（LWP）

用户级线程和内核级线程的一个区别就是他们如何调度的。

为了决定哪个内核级线程调度到一个处理器上，内核采用系统竞争范围（SCS）。采用SCS调度来竞争CPU，发生在系统内所有线程之间。采用一对一模型的系统，如Windows、Linux、Solaris，只采用SCS调度。

Pthreads调度

在通过POSIX Pthreads来创建线程时，允许指定PCS或SCS。Pthreads采用如下竞争范围的值：

PTHREAD_SCOPE_PROCESS：按PCS来调度线程

PTHREAD_SCOPE_SYSTEM：按SCS来调度线程

Actor调度算法（AKKA）

Actor执行的时候会发生以下行为：

1. Actor将消息加入到消息队列的尾部
2. 假如一个Actor并未被调度执行，则将其标记为可执行
3. 一个（对外部不可见）调度器对Actor的执行进行调度
4. Actor从消息队列头部选择一个消息进行处理
5. Actor在处理过程中修改自身的状态，并发送消息给其他的Actor

为了实现这些行为，Actor必须有以下特性：

- 邮箱（作为一个消息队列）
- 行为（作为Actor的内部状态，处理消息逻辑）
- 消息（请求Actor的数据，可看成方法调用时的参数数据）
- 执行环境（比如线程池，调度器，消息分发机制等）
- 位置信息（用于后续可能会发生的行为）

因为Actor不需要在任何地方使用到锁，所以，不会阻塞发送者。

Actor A要向Actor B发送一条消息，大致流程是这样的。

- 假设需要通信的Actor是在同一进程中，那么Actor是可以直接把消息投递到目标Actor的mq。调度流程：A处于工作线程的调度中，向B的mq投递一条消息，并将B加入gmq队尾，然后A结束调度。

工作线程会循环执行scheduler函数，直到从gmq中取出B，取出第一条消息并进行任务调度。原理就这么简单。

- 又假设需要通信的Actor不再同一个进程中，Actor不可以直接把消息投递到目标Actor的mq，此时就需要借助socket进行通信的中转。

准备工作：以epoll为例，Actor之间建立连接后，每个Actor都分配一个socket，那么可以在socket对应的epoll_event.data.ptr上挂载一个结构体，主要记录消息到来时，会转发到哪个Actor的mq。这里epoll单独开辟一个IO线程来处理网络消息相关的逻辑。

调度流程：A处于工作线程的调度中，通过socket发送消息到B端，A结束调度。B端的socket收到消息后，通过其epoll_event.data.ptr获取Actor的标识，并找到B，将消息投递到B的mq，并将B假如gmq队尾，IO线程对B的消息中转工作完成。工作线程会玄幻执行scheduler函数，直到从gmq中取出B，取出第一条消息并进行任务调度。

缓存一致性

缓存一致性是指CPU在主内存和其他CPU之间传输缓存。

http和https的原理，1.0和2.0差别和实现原理

http协议是一种使用明文数据传输的网络协议。一直以来http协议都是最主流的网络协议，http协议的明文传输会让用户存在一个非常大的安全隐患。试想一下，假如你在一个http协议的网站上面购物，你需要在页面输入你的银行卡号和密码，然后你把数据提交到服务器实现购买。假如这个合适，你的传输数据被第三者给截获了，由于http明文数据传输的原因，你的银行卡号和密码，将会被这个截获人所得到。

https协议可以理解为http协议的升级，就是在http的基础上增加了数据加密。在数据进行传输之前，对数据进行加密，然后再发送到服务器。这样，就算数据被第三者所截获，但是由于数据是加密的，所以你的个人信息仍然是安全的。这就是http和https说的最大区别。

http特点：

1. 无状态：协议对客户端没有状态存储，对事物处理没有“记忆”能力，比如访问一个网站需要反复进行登录操作。
2. 无连接：HTTP/1.1之前，由于无状态特点，每次请求需要通过TCP三次握手四次挥手，和服务器重新建立连接。比如某个客户机在短时间多次请求同一个资源，服务器并不能区别是否已经响应过用户的请求，所以每次需要重新响应请求，需要耗费不必要的时间和流量。通过Cookies/Session技术解决无状态的问题，通过keep-alive方法，只要任意一端没有明确提出断开连接，则保持TCP连接状态，在请求首部字段中的Connection：keep-alive即为表明使用了持久连接。
3. 基于请求和响应：基本的特性，由客户端发起请求，服务端响应
4. 简单快速、灵活
5. 通信使用明文、请求和响应不会对通信方进行性确认、无法保护数据的完整性

https特点：

1. 内容加密：采用混合加密技术，中间者无法直接查看明文内容
2. 验证身份：通过证书认证客户端访问的是自己的服务器
3. 保护数据完整性：防止传输的内容被中间人冒充或者篡改

混合加密: 结合非对称加密和对称加密技术。客户端使用对称加密生成秘钥对传输数据进行加密，然后使用非对称加密的公钥再对秘钥进行加密，所以网络上传输的数据是被秘钥加密的秘闻和用公钥加密后的秘密秘钥，因此即使被黑客截取，由于没有私钥，无法获取到加密明文的秘钥，便无法获取到明文数据。

数字摘要: 通过单向hash函数对原文进行hash，需要加密的明文“摘要”成一串固定长度（如128bit）的密文，不同的明文摘要成的密文其结果总是不相同，同样的明文其摘要必定一致，并且即使知道了摘要也不能反推出明文。

数字签名技术: 数字签名建立在公钥加密体制基础上，是公钥加密技术的另一类应用。它把公钥加密技术和数字摘要结合起来，形成了实用的数字签名技术。

单机性能如何优化

- 程序优化
 - 1. 表单压缩
 - 2. 算法优化
 - 3. 批处理：对于大批量的数据处理，最好能够批处理，不会因为单词操作而影响系统的正常使用
 - 4. 索引：编写合理的SQL，尽量利用索引
 - 5. 防止死锁
 - 6. 防止内存泄露
 - 7. 并行：使用多个进程和线程来处理任务
 - 8. 异步：比如用MQ（消息中间件）来解耦系统之间的依赖关系，减少阻塞
- 配置优化
 - 1. jvm配置优化：合理分配堆与非堆的内存，配置合适内存的会所算法，提高系统服务能力
 - 2. 连接池：数据库连接池可以节省建立连接与关闭连接的资源消耗
 - 3. 线程池
 - 4. 缓存机制
- 数据库连接池优化
 - 1. 线程池的参数配置
 - 2. 连接池配置多少连接
 - 3. 监控线程池
- DB优化
 - 1. 优化物理结构，数据逻辑设计与物理设计要科学高效，比如分区、索引建立、字段类型及长短、冗余设计等
 - 2. 查询器优化，指定的执行几哈加快查询查找速度。比如连接查询时指定驱动表，减少表的扫描次数
 - 3. 单条SQL语句优化，对单SQL进行优化分析，比如查询条件选择索引列
 - 4. 并行SQL，对数据量巨大的表的数据遍历，用多个线程分块处理。
 - 5. 减少资源争用，可以提高IO效率减少响应时间从而提高吞吐量来缓解争用，比如物理拆分把热点数据分布在不同表空间。
 - 6. 优化内存、减少物理IO访问。
 - 7. 优化IO，进行条带化、读写分离、减少热点等

可用性相关的需要注意有哪些问题？

解决方案需要考虑什么？

http请求在浏览器发出后如何工作

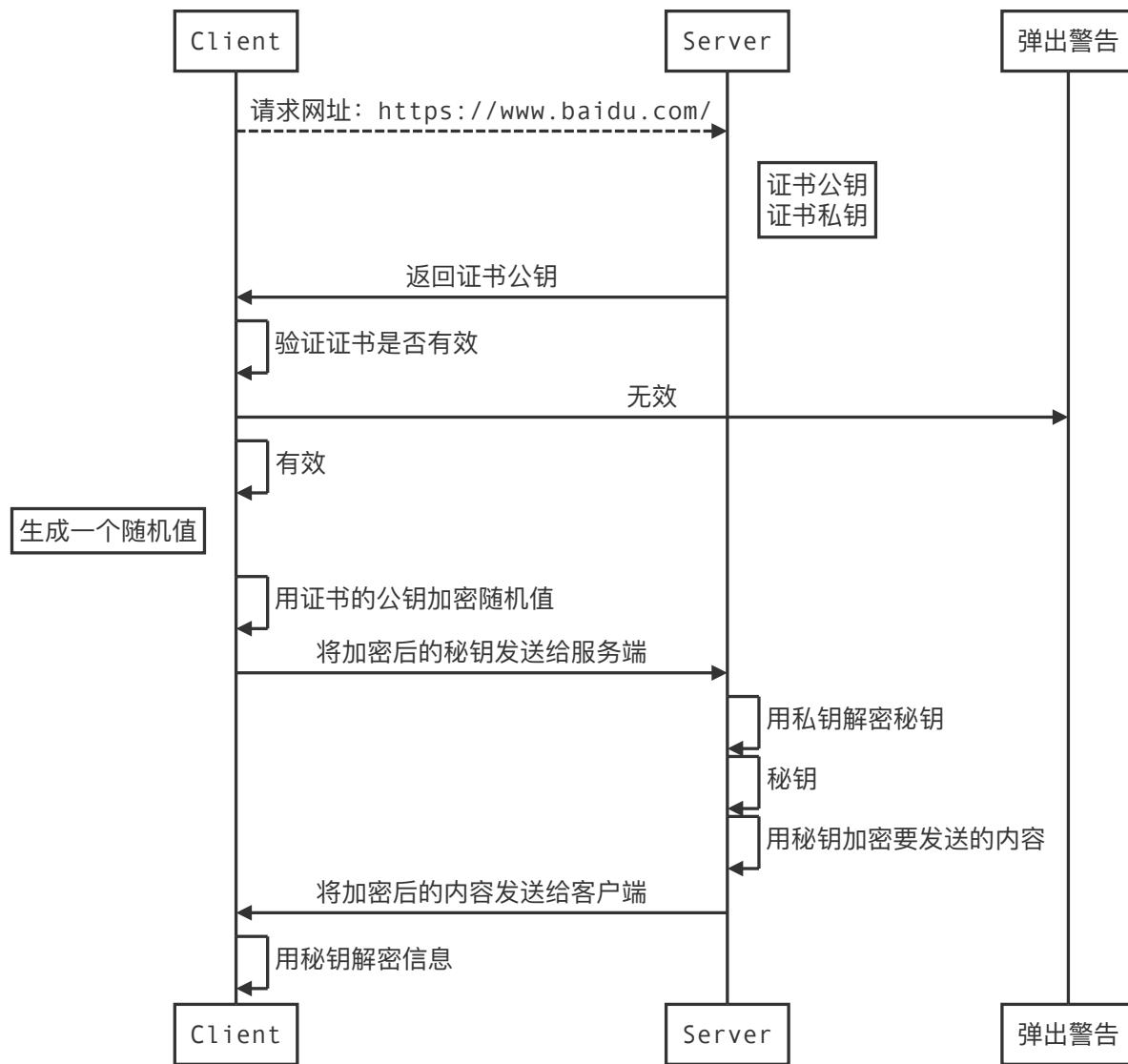
http加密（握手）过程

- http：直接通过明文在浏览器和服务器之间传递
- https：采用对称加密和非对称加密结合的方式来保护浏览器和服务器之间的通信安全。对称加密算法加密数据+非对称加密算法交换秘钥+数字证书验证身份=安全

https其实是有两部分组成：http+ssl/TLS，也就是在http上又加了一层处理加密信息的模块。服务端和客户端的信息传输都会通过TLS进行加密，所以传输的数据都是加密后的数据。

1. 传统的HTTP协议通信：传统的HTTP报文是直接将报文信息传输到TCP然后TCP再通过TCP套接字发送给目的主机上。
2. https协议通信：https是http报文直接将报文信息传输给ssl套接字进行加密，ssl加密后将加密后的报文发送给tcp套接字，然后tcp套接字再将加密后的报文发送给目的主机，目的主机将通过tcp套接字获取加密后的报文给ssl套接字，ssl解密后交给对应进程。

HTTPS时序图



HTTPS加密请求（一次握手）过程

- 首先，客户端发起握手请求，以明文传输请求信息，包含版本信息，加密-套件候选列表，压缩算法候选列表，随机数，扩展字段等信息(这个没什么好说的，就是用户在浏览器里输入一个HTTPS网址，然后连接到服务端的443端口。)
- 服务端的配置，采用HTTPS协议的服务器必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面。这套证书其实就是一对公钥和私钥。如果对公钥不太理解，可以想象成一把钥匙和一个锁头，只是世界上只有你一个人有这把钥匙，你可以把锁头给别人，别人可以用这个锁把重要的东西锁起来，然后发给你，因为只有你一个人有这把钥匙，所以只有你才能看到被这个锁锁起来的东西。
- 服务端返回协商的信息结果，包括选择使用的协议版本 version，选择的加密套件 cipher suite，选择的压缩算法 compression method、随机数 random_S 以及证书。(这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等等。)
- 客户端验证证书的合法性，包括可信性，是否吊销，过期时间和域名。(这部分工作是由客户端的SSL/TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警示框，提示证书存在的问题。如果证书没有问题，那么就生成一个随机值。然后用证书(也就是公

钥) 对这个随机值进行加密。就好像上面说的, 把随机值用锁头锁起来, 这样除非有钥匙, 不然看不到被锁住的内容。)

- 客户端使用公匙对对称密匙加密, 发送给服务端。(这部分传送的是用证书加密后的随机值, 目的是让服务端得到这个随机值, 以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。)
- 服务器用私钥解密, 拿到对称加密的密匙。(服务端用私钥解密后, 得到了客户端传过来的随机值, 然后把内容通过该随机值进行对称加密, 将信息和私钥通过某种算法混合在一起, 这样除非知道私钥, 不然无法获取内容, 而正好客户端和服务端都知道这个私钥, 所以只要加密算法够彪悍, 私钥够复杂, 数据就够安全。)
- 传输加密后的信息, 这部分信息就是服务端用私钥加密后的信息, 可以在客户端用随机值解密还原。
- 客户端解密信息, 客户端用之前生产的私钥解密服务端传过来的信息, 于是获取了解密后的内容。整个过程第三方即使监听到了数据, 也束手无策。

视频解码项目中的优化措施

tcp拥塞控制

const的实现机制

在c++中, 被const修饰的变量, 可能为其分配存储空间, 也可能不分配存储空间。

有下面两种情况, 会为这个变量分配存储空间:

- 1、当const常量为全局, 并且需要在其他文件中使用时 (extern)
- 2、当使用取地址符 (&) 取const常量的地址时

局部const常量, 存储在函数的栈区, 是可以被修改的。使用`const_cast<int*>`类似这种。

而常量引用会被编译器进行“常量替换”。

全局const变量根本没有被分配内存, 所以无法获取全局const变量的内存。变量的定义实际被保存在符号表里面, 所以符号表中一定有全局const变量的信息。全局const变量可能占用内存空间这个是否占用内存空间是由编译器的优化策略决定的。比如没有对const进行取地址等操作, 那么斤斤需要将const信息保存在符号表中就可以了。如果有取地址等操作, 那么全局const就会被分配存储空间, 这个存储空间肯定是在常量区。

全局const变量是没有办法修改的。

技术基础知识点和算法题 (链表反转, 全部反转和部分反转)

聊做的项目, 一步一步分析, 怎么实现, 可优化的地方。

redis缓存相关技术点

redis作为缓存的作用就是减少对数据库的访问压力，当我们访问一个数据的时候，首先我们从redis中查看是否有该数据，如果没有，则从数据库中读取，将从数据库中读取的数据存放到缓存中，下次访问同样的数据，先判断redis中是否存在该数据，如果有，则从缓存中读取，不访问数据库了。

在大并发的情况下，所有的请求直接访问数据库，数据库会出现连接异常。这个时候，就需要使用redis做一个缓冲操作，让请求先访问到redis，而不是直接访问数据库。

缓存同步原理：就是将redis中的key进行删除，下次访问的时候，redis没有该数据，则从DB进行查询，再次更新到redis中。

缓存使用主要面对四个问题：

- 缓存和数据库双写一致问题

答这个问题，先明白一个前提。就是如果数据有强一致性要求，不能放缓存。我们所做的一切，只能保证最终一致性。另外，我们所做的方案其实从根本上来说，只能说降低不一致发生的概率，无法完全避免。因此，有强一致性要求的数据，不能放缓存。

首先，采取正确的缓存更新策略，先更新数据库，再删缓存。其次，因为可能存在删除缓存失败的问题，提供一个补偿措施即可，例如利用消息队列。

- 缓存雪崩问题（集体失效）

缓存雪崩，即缓存同一时间大面积失效，这个时候又来了一拨请求，结果请求都堆到数据库上，从而导致数据库连接异常。

解决方案：

1. 给缓存的失效时间，加上一个随机值，避免集体失效。
2. 使用互斥锁，但是该方案吞吐量明显下降了。
3. 双缓存。我们有两个缓存，缓存A和缓存B。缓存A的失效时间为20分钟，缓存B不设失效时间。自己做缓存预热操作。然后细分以下几个小点：

从缓存A读取数据库，有则直接返回

A没有数据，直接从B读数据，直接返回，并且异步启动一个更新线程

更新线程同时更新缓存A和缓存B

- 缓存击穿问题（访问不存在的key，导致缓存失效，直接击穿到数据库）

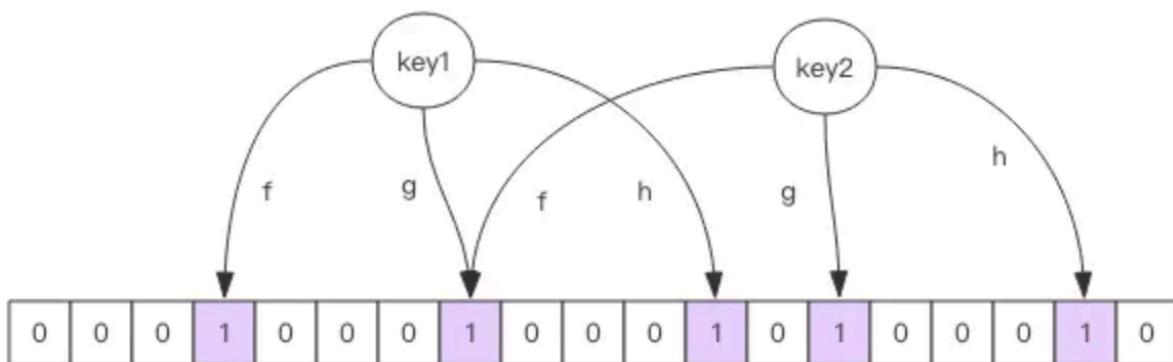
缓存穿透，即黑客故意去请求缓存中不存在的数据，导致所有的请求都堆到数据库上，从而导致连接异常。

解决方案：

1. 利用互斥锁，缓存失效的时候，先去获得锁，得到锁了，再去请求数据库。没得到锁，则休眠一段时间重试
2. 采用异步更新策略，无论key是否取到值，都直接返回。value值中维护一个缓存失效时间，缓存如果过期，异步起一个线程去读数据库，更新缓存。需要做缓存预热（项目启动前，先加载缓存）操作。
3. 提供一个迅速判断请求是否有效的拦截机制，比如，利用布隆过滤器，内部维护一系列合法有效的key。迅速判断出，请求所携带的key是否合法有效。如果不合法，则直接返回。

布隆过滤器

布隆过滤器是一个不精确的 set 结构，用于判断某个元素是否存在过，存在一定误判。内部实现是一个大型的位数组，通过多次哈希进行散落。效果与空间向量类似，同样的元素散落结果一定相同。



- 缓存的并发竞争问题

同时有多个子系统去set一个key。这个时候要注意什么呢？大部分人都推荐使用redis事务机制。但是redis集群环境，做了数据分片操作。一个事务中有涉及到多个key操作的时候，这多个key不一定都存储在同一个redis-server上。因此，redis的事务机制，十分鸡肋。

1. 如果对这个key操作，不要求顺序。这种情况下，准备一个分布式锁，大家去抢锁，抢到锁就做set操作即可，比较简单。
2. 如果对这个key的操作，要求顺序。假设有一个key1，系统A需要将key1设置为valueA，系统B需要将key1设置为valueB，系统C需要将key1设置为valueC。期望按照key1的value值按照valueA--valueB--valueC的顺序变化。这种时候我们在数据写入数据库的时候，需要保存一个时间戳。假设时间戳如下：

系统A key1 「valueA 3: 00」

系统B key1 「valueB 3: 05」

系统C key1 「valueC 3: 10」

那么，假设系统B先抢到锁，将key1设置为「valueB 3: 05」。接下来系统A抢到锁，发现自己的valueA的时间戳早于缓存中的时间戳，那就不做set操作了。以此类推。

其他方法，比如利用队列，将set方法编程串行访问也可以。总之，灵活变通。

redis为什么那么快？

主要就是以下三点：

1. 完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于 hashmap，hashmap的优势就是查找和操作的时间复杂度都是 $O(1)$
2. 采用单线程，避免不必要的cpu上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗cpu，不用去考虑各种锁问题，不存在加锁释放锁的操作，没有因为可能出现死锁而导致性能消耗。

redis设计思路：

- IO模型

单线程程序，使用非阻塞IO：能读时读，能写时写，无线程阻塞

非阻塞IO基于操作系统底层的时间轮询API，注册监听基于回调。

- 持久化

redis提供两种持久化方式：

1. RDB--一段时间内生成指定时间点生成数据集快照（snapshot）

优点：

1. rdb多文件存储，文件方式方便存储和传输，非常适合做冷备。
2. rdb对正常对外提供读写服务，影响很小，可以让redis保持高性能，因为可以fork子进程做这件事情
3. 相对于AOF来说，基于rdb文件的重启和恢复，更加快速

缺点：

1. 如果想尽可能减少数据丢失，rdb没有aof好。因为rdb快照文件，是间隔5分钟更新一次，更新时间较短。
2. rdb如果每次fork进程来处理快照文件，文件过大的时候，可能会导致客户端失去响应数毫秒，甚至数秒。搞不好，会出现大面积redis连接出错。

2. AOF--增量日志

redis提供了save和bgsave两个命令来生成rdb文件，前者是阻塞的，后者是后台fork子进程。

AOF持久化实现可以分为命令追加（append）、文件写入（write）、文件同步（fsync）三个步骤。

Append追加命令到AOF缓冲区，Write将缓冲区的内容写入到程序缓冲区，Fsync将程序缓冲区的内容写入到文件。

由于AOF比RDB文件更加完整，server启动时优先采用AOF文件进行恢复。

优点：

1. 数据丢失少，一般aof间隔1秒，后台执行一个fsync操作。
2. 日志文件已append-only模式写入，没有磁盘寻址开销，写入性能高。数据不容易损坏。
3. AOF日志可读性强，非常适合做灾难性的误删除的紧急恢复。比如某人不小心用flushall命令清空了所有数据，只要这个时候后台rewrite还没有发生，那么就可以立即拷贝AOF文件，将最后一条flushall命令给删了，然后再将该AOF文件放回去，就可以通过恢复机制，自动恢复所有数据。

缺点：

1. AOF文件一般都比rdb文件更占空间
2. aof会大大降低实时写入的qps

Redis支持同时开启rdb和aof。

- 主从复制

主从复制是指一个server复制另一个server的数据

主从复制的对象：master和多slaver，和集群的概念区分开

主从复制的三个阶段：复制初始化（建立连接）、数据同步（同步现有数据）和命令传播（同步增量数据）

主从复制的策略：从未同步的slaver全量同步，同步过的slaver部分同步

- 集群部署

大规模数据存储系统都会面临水平扩展的问题。水平扩展可以通过数据分片来解决，通过一致性hash来避免大量的rehash数据迁移。但是数据分片如何屏蔽业务，在redis3.0支持集群以前。

- 哈希槽

redis cluster的数据分片依赖hash槽（slot）实现，集群预先划分16384个slot，每个分片负责一部分slot，请求时先通过key计算hash值，然后映射到唯一的slot，最后直连对应的数据分片。

- 分片

redis cluster的数据分片通常有多态服务组成，一主多从（master and slaves）。master负责写（或读），slaver负责读（或转发写请求）。主从服务使用raft协议完成leader选举（故障转移），使用epoch的概念来给事件增加版本号。

项目的架构的设计，实现的逻辑是什么样的。

分布式架构，优化方案

一面：

- 1.缓存穿透怎么解决
- 2.缓存雪崩怎么解决
- 3.怎么解决大量请求对同一个redis 热点key的访问
- 4.应用进程cpu100%了，排查解决的思路是什么样
- 5.redis中间件整体设计思路是怎么样，解决了哪些问题
- 6.稳定性治理涵盖哪些方面，全链路自动化压测涵盖了哪些具体技术，推动过程中遇到了哪些阻力，怎么解决
- 7.微服务模式下串联整个链路的思路是怎么样的
- 8.长链接底层设计思路，涵盖协议位，数据结构，优化措施，全网下发策略，手动重启策略，异常coredump容灾，跨机房策略
- 9.im具体设计思路是怎么样的，底层数据结构设计，写扩散和读扩散，怎么解决消息漫游，多端同时在线数据一致性，重装/切换客户端后数据同步
- 10.编程及算法题：1.ipv4转整形 2.[8,9,10,0,1,2,3,4,5,6,7]局部翻转有序数组，输入一个数，输出下标。时间复杂度尽可能小，绝对不能是O（n）。

char *p = "192.168.001.001";

```
int ipv4_to_int(char *ip)
{
    int tmp = 0;
    char ip1, ip2, ip3, ip4;

    ip1 = atoi(ip);
    ip = strchr(ip, '.');
    if(!ip)
        return -1;
    ip2 = atoi(++ip);
    ip = strchr(ip, '.');
```

```

if(!ip)
    return -1;
ip3 = atoi(++ip);
ip = strchr(ip, '.');
if(!ip)
    return -1;
ip4 = atoi(++ip);

tmp |= ip4 & 0xff;
tmp = (tmp << 8) | (ip3 & 0xff);
tmp = (tmp << 8) | (ip2 & 0xff);
tmp = (tmp << 8) | (ip1 & 0xff);

return tmp;
}

```

整形转ipv4字符串：

```

void int_to_ipv4(int tmp, char *ip)
{
    unsigned short ip1, ip2, ip3, ip4;
    ip1 = tmp & 0xff;
    ip2 = (tmp >> 8) & 0xff;
    ip3 = (tmp >> 16) & 0xff;
    ip4 = (tmp >> 24) & 0xff;
    sprintf(ip, "%03d.%03d.%03d.%03d", (int)ip1, (int)ip2, (int)ip3,
    (int)ip4);
}

```

11.为什么从php切换到go, go有哪些优势

12.怎么解决微服务调用链路上一个依赖服务拖垮整个服务

二面：

1.redis 备份机制有哪些，分别的底层实现机制是怎么样的

2.redis cluster模式数据分片的思路是怎么样的

3.redis cluster模式和哨兵模式分别应用场景是什么样

4.redis cluster怎么解决mget问题

Redis怎么解决部分热点key访问过大问题

一个key成为热点key，例如一个重要的新闻，一个热门的八卦新闻等等，所以这种key的访问量可能非常大。

缓存的构建是需要一定时间的。可能是一个复杂的计算，例如复杂的sql、多次IO、多个依赖（各种接口调用）等。

于是就出现一个致命问题：在缓存失效的瞬间，有大量的线程来构建缓存，造成后端负载加大，甚至可能会让系统崩溃。

解决思路有两种：

1. 使用“互斥锁”，这样保证数据在写入的时候，读取操作无法进行。
2. 在redis层面，设置数据不过期，但是，把key的过期时间存储在value里面，如果发现要过期了，通过一个后台的异步线程进行缓存的构建，也就是“逻辑”过期。

其实这个问题的本质就是访问者不知道数据即将过期，所以，逻辑过期可以让用户永远有时间知道这个数据是否即将过期了。

6.多数据中心架构下，解决数据读写一致性的思路有哪些

7.im技术细节，问的非常细，单单im一个项目聊了整整一小时

8.在长链接场景下，用户连在哪台server上存在redis上。基于上面前提，在海量下发通知调用，怎么防止redis查询过多被打挂

1、亿级表如何添加索引，不影响线上

2、profile的使用

腾讯一面（20200404）：

mysql：索引的数据结构，红黑树支持范围索引吗？

如何存储10亿个qq号在线/离线状态，以及如何快速索引其中一个qq号

TCP拥塞控制算法

操作系统缺页中断什么时候发生

golang携程，操作系统线程区别，携程是否需要内核参与

堆数据结构

堆就是用数组实现的二叉树，所以它没有使用父指针或者子指针。堆根据“堆属性”来排序，“堆属性”决定了树中节点的位置。

堆的常用场景：

- 构建优先队列
- 支持堆排序
- 快速找出一个集合中的最小值（或者最大值）

堆分为两种：最大堆和最小堆，两者的差别在于节点的排序方式。

在最大堆中，父节点的值比每一个子节点的值都要大。在最小堆中，父节点的值比每一个子节点的值都要小。这就是所谓的“堆属性”，并且这个属性对堆中的每一个节点都成立。

可以用来解决以下问题：

给定一个数组（浮点数），如何快速找出第k大的数

比较简单的方法就是使用快速排序，排序完毕找出第k个数。时间复杂度是 $n \cdot \log(n)$ 。

思考：

1. 直接从大到小排序，排好序后，第k大的数就是 $\text{arr}[k-1]$ 。
2. 只需找到第k大的数，不必把所有的数排好序。我们借助快速排序中partition过程，一般情况下，

在把所有数都排好序前，就可以找到第k大的数。我们依据的逻辑是，经过一次partition后，数组被pivot分成左右两部分：S左、S右。当S左的元素个数 $|S\text{左}|$ 等于 $k-1$ 时，pivot即是所找的数；当 $|S\text{左}|<k-1$ ，所找的数位于S右中；当 $|S\text{左}|>k-1$ ，所找的数位于S左中。显然，后两种情况都会使搜索空间缩小。

算法的时间复杂度为： $O(N)$ --详情参考算法导论。

如果N很大，100亿？甚至更多，这个时候数据不能够全部放入内存，所以要求尽可能少遍历数据。不妨设 $N>K$,考虑前K个数中的最大K个数的一个退化的情况：所有K个数就是最大的K个数。

如果考虑第 $K+1$ 个数X呢？如果X比最大的K个数中的最小的数Y小，则最大的K个数保持不变。如果X比最大的K个数中的最小的数Y大，则最大的K个数要除去Y，加入X。如果用一个数组来保存前K大的数，每加入一个数X，就扫描一遍数组。得到数组中最小的数Y，用X代替Y或者保持不变。这种方法消耗的时间 $O(N*K)$

进一步，可以用容量为K的**最小堆**来存储最大的K个数。最小堆的堆顶元素就是K个数中最小的一个。每次考虑一个数X，如果X比堆顶元素Y小，则保持最小堆不变，因为这个元素比最大的K个数小。如果X比堆顶元素Y大，那么用X替换原来的堆顶元素Y，X可能破坏原来的最小堆结构（每个结点比它的父节点大），需要更新堆来维持堆的性质。更新堆时间复杂度为 $O(\log_2 K)$.总的算法复杂度为 $O(N*\log_2 k)$

如何做客户端鉴权

为什么使用ip白名单，而不是token鉴权（项目相关）

动态添加IP黑/白名单

目前服务端主要有两种ip白名单实现机制：nginx和iptables配置实现

- 手动封IP步骤

1. Nginx手动封IP

获取各个IP访问次数

```
awk '{print $1}' nginx.access.log | sort | uniq -c | sort -n
```

新建一个黑名单文件 `blacklist.conf`，放在 `nginx/conf` 下面

添加一个IP，`deny 192.168.1.1;`

在http或者server模块引入

```
include blacklist.conf;
```

需要重启服务器，设置生效

```
nginx -s reload;
```

2. iptables手动封IP

单个IP的命令是

```
iptables -I INPUT -s 124.115.0.119 -j DROP
```

封IP段的命令是

```
iptables -I INPUT -s 124.115.0.0/16 -j DROP
```

封整个段的命令是

```
iptables -I INPUT -s 194.42.0.0/8 -j DROP
```

封几个段的命令是

```
iptables -I INPUT -s 61.37.80.0/24 -j DROP  
iptables -I INPUT -s 61.37.81.0/24 -j DROP
```

限制并发访问(限制211.1.0.1访问80端口的并发数不超过10)

```
iptables -I INPUT -p tcp --dport 80 -s 211.1.0.1 -m connlimit --connlimit-above 10 -j REJECT
```

解封

```
iptables -F
```

清空 (iptables -L --line-numbers 查看id)

```
iptables -D INPUT $ID
```

生效

```
service iptables save  
service iptables restart  
iptables -L -n
```

- Nginx自动封IP

1. 示例：覆盖

```
#!/bin/sh
tail -n500000 /usr/local/tengine/logs/access.log | awk '{print $1,$7}' | grep
-i -E "payments|smsSdk|reportErrorLog|errorPay" | awk '{print $1}' | sort |
uniq -c | sort -rn | awk '{if($1>100)}{print "deny \"$2\";"}' >
/usr/local/tengine/conf/ip.blacklist.auto.conf
/usr/local/tengine/sbin/nginx -s reload
```

简而言之，查询访问日志，并统计

2. 示例：追加

```
#!/bin/sh
tail -n500000 /usr/local/tengine/logs/access.log | awk '{print $1,$7}' | grep
-i -E "payments|smsSdk|reportErrorLog|errorPay" | awk '{print $1}' | sort |
uniq -c | sort -rn | awk '{if($1>500)}{print "deny \"$2\";"}' >>
/usr/local/tengine/conf/ip.blacklist.auto.append.conf
/usr/local/tengine/sbin/nginx -s reload
```

3. nginx中配置

```
local / {
    ...
    limit_req zone=one burst=5 nodelay;

    include ip.blacklist.auto.append.conf;
    include ip.blacklist.auto.conf;
}
```

其实可以看到，这里所谓的**自动封**，其实还是要重启nginx，期间只是利用脚本扫描了访问日志里面高频访问IP，省去了手动查询而已。并不算是非常完美的解决方案。

接下来还有一个方案，是利用crontab自动执行，但是假如还是要重启nginx的话，在高并发场景，是并不能被接受的。

思考：

假如，需要配置ip白名单，又不想完全重启iptables和nginx，那么只有通过业务方维护IP白名单，并且定期通过请求服务的方式更新白名单列表。但是这样子做的风险就是：

1. 多了一层流量转发，需要添加前置机器
2. 此外，依然不能阻止请求进入前置机器

优点就是：不需要重启nginx了

个人感觉，小型项目，直接启动iptables应该是比较靠谱的。

分布式系统发生雪崩如何处理，要求重点讲一下超时机制

这里我犯了一个错误，首先https和tcp的握手设置超时时间，这个是客户端的超时设置，由客户端代码对这个超时的异常进行捕捉，这个跟服务端超时，没有任何关系。

具体关于超时的解决方案描述，见前文

Https协议是如何生成随机值

Https协议是如何判断证书有效

这就要从 CA 证书讲起了。CA 证书其实就是数字证书，是由 CA 机构颁发的。至于 CA 机构的权威性，那是毋庸置疑的，所有人都是信任它的。CA 证书内一般会包含以下内容：

- 证书的颁发机构、版本
- 证书的使用者
- 证书的公钥
- 证书的有效时间
- 证书的数字签名 Hash 值和签名 Hash 算法
- ...

正好我们把客户端如何校验 CA 证书的步骤说下吧。

CA 证书中的 Hash 值，其实是用证书的私钥进行加密后的值（证书的私钥不在 CA 证书中）。然后客户端得到证书后，利用证书中的公钥去解密该 Hash 值，得到 Hash-a；然后再利用证书内的签名 Hash 算法去生成一个 Hash-b。最后比较 Hash-a 和 Hash-b 这两个的值。如果相等，那么证明了该证书是对的，服务端是可以被信任的；如果不相等，那么就说明该证书是错误的，可能被篡改了，浏览器会给出相关提示，无法建立起 HTTPS 连接。除此之外，还会校验 CA 证书的有效时间和域名匹配等。

接下来我们就来详细讲一下 HTTPS 中的 SSL 握手建立过程，假设现在有客户端 A 和服务器 B：

1. 首先，客户端 A 访问服务器 B，比如我们用浏览器打开一个网页 <https://www.baidu.com>，这时，浏览器就是客户端 A，百度的服务器就是服务器 B 了。这时候客户端 A 会生成一个随机数1，把随机数1、自己支持的 SSL 版本号以及加密算法等这些信息告诉服务器 B。
2. 服务器 B 知道这些信息后，然后确认一下双方的加密算法，然后服务端也生成一个随机数 B，并将随机数 B 和 CA 颁发给自己的证书一同返回给客户端 A。
3. 客户端 A 得到 CA 证书后，会去校验该 CA 证书的有效性，校验方法在上面已经说过了。校验通过后，客户端生成一个随机数3，然后用证书中的公钥加密随机数3 并传输给服务端 B。
4. 服务端 B 得到加密后的随机数3，然后利用私钥进行解密，得到真正的随机数3。
5. 最后，客户端 A 和服务端 B 都有随机数1、随机数2、随机数3，然后双方利用这三个随机数生成一个对话密钥。之后传输内容就是利用对话密钥来进行加解密了。这时就是利用了对称加密，一般用的都是 **AES 算法**。
6. 客户端 A 通知服务端 B，指明后面的通讯用对话密钥来完成，同时通知服务器 B 客户端 A 的握手过程结束。
7. 服务端 B 通知客户端 A，指明后面的通讯用对话密钥来完成，同时通知客户端 A 服务器 B 的握手过程结束。
8. SSL 的握手部分结束，SSL 安全通道的数据通讯开始，客户端 A 和服务器 B 开始使用相同的对话密钥进行数据通讯。

到此，SSL 握手过程就讲完了。可能上面的流程太过于复杂，我们简单地来讲：

1. 客户端和服务端建立 SSL 握手，客户端通过 CA 证书来确认服务端的身份；
2. 互相传递三个随机数，之后通过这随机数来生成一个密钥；

3. 互相确认密钥，然后握手结束；
4. 数据通讯开始，都使用同一个对话密钥来加解密；

我们可以发现，在 HTTPS 加密原理的过程中把对称加密和非对称加密都利用了起来。即利用了非对称加密安全性高的特点，又利用了对称加密速度快，效率高的好处。真的是设计得非常精妙，令人赞不绝口。

解决高并发的技术手段

主要从技术而非业务解析

- 悲观锁
- 乐观锁（但是像actor模型基本不需要使用锁--无锁化）
- Actor/携程无锁化
- 并行/异步处理耗时程序（如写kafka，写文件，写log）/异步、回调
- FIFO缓存队列
- 入口层反作弊
- 入口层熔断（regisry）+业务逻辑层超时（timeout）/服务降级
- epoll（IO多路复用，nginx/redis方案）

Zookeeper一致性协议（ZAB）

什么是ZAB协议？

ZAB协议，全称是Zookeeper Atomic Broadcast（Zookeeper原子广播）

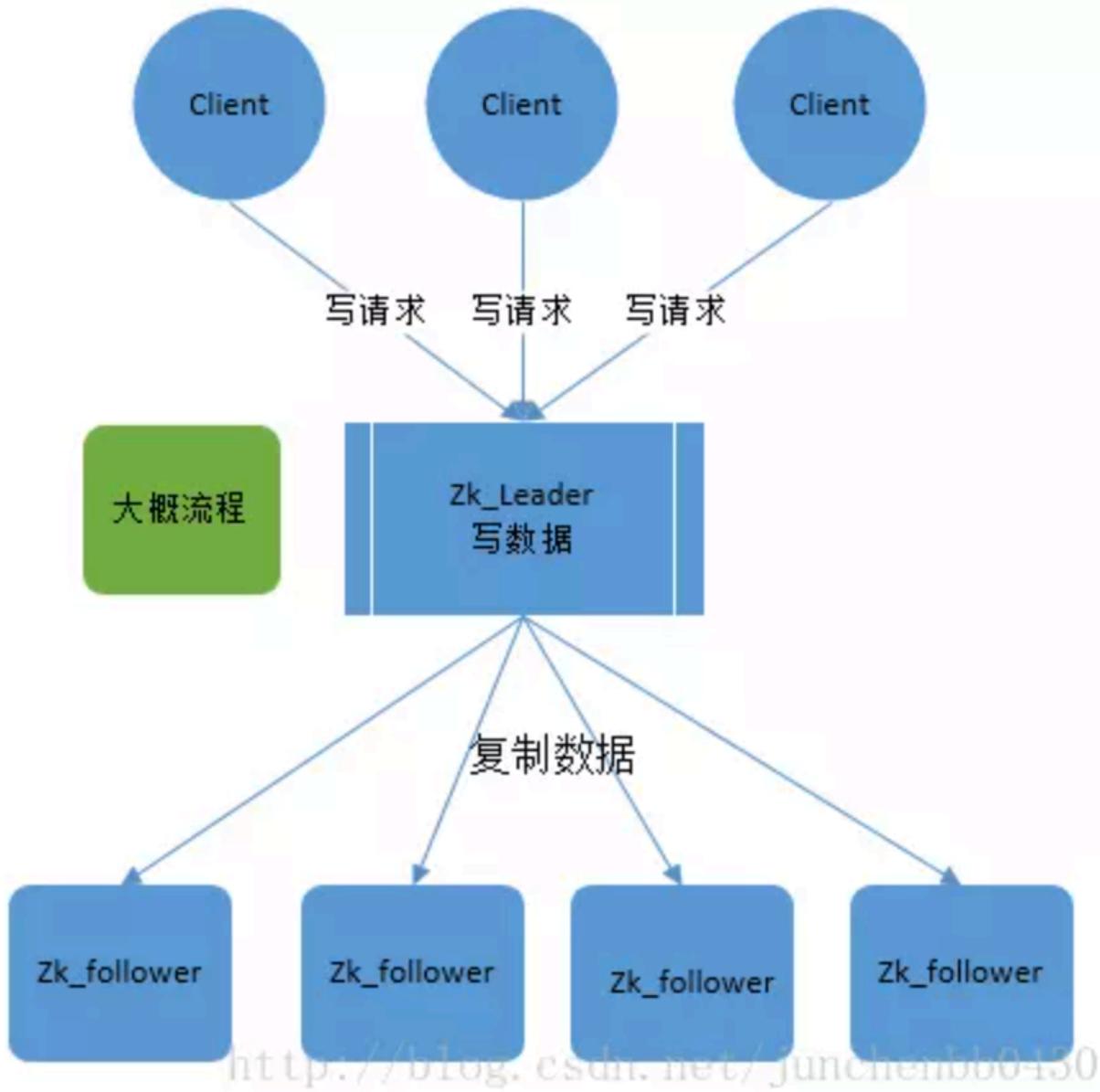
Zookeeper是通过Zab协议来保证分布式事务的最终一致性

1. ZAB协议是为分布式协调服务Zookeeper专门设计的一种支持崩溃恢复的原子广播协议，是Zookeeper保证数据一致性的核心算法。ZAB借鉴了Paxos算法，但又不像Paxos那样，是一种通用的分布式一致性算法。它是特别为Zookeeper设计的支持崩溃恢复的原子广播协议。
2. 在Zookeeper中主要依赖ZAB协议来实现数据一致性，基于该协议，zk实现了一种主备模型（即Leader和Follower模型）的系统架构来保证集群中各个副本之间数据的一致性。这里的主备系统架构模型，就是指只有一台客户端（Leader）负责处理外部的写事务请求，然后Leader客户端将数据同步到其他Follower节点。

Zookeeper客户端会随机的链接到zookeeper集群中的一个节点，如果是读请求，就直接从当前节点中读取数据；如果是写请求，那么节点就会向Leader提交事务，Leader接收到事务提交，会广播该事务，只要超过半数节点写入成功，该事务就会被提交。

ZAB协议的特性

- 1) ZAB协议需要确保那些已经在Leader服务器上提交（commit）的事务最终被所有的服务器提交。
- 2) ZAB协议需要确保丢弃那些只在Leader上被提出而没有被提交的事务。



ZAB协议实现的作用

- 1) 使用一个单一的主进程（Leader）来接收并处理客户端的事务请求（也就是写请求），并采用了ZAB的原子广播协议，将服务器数据的状态变更以事务proposal（事务提议）的形式广播到所有的副本（Follower）进程上去。
- 2) 保证一个全局的变更序列被顺序引用。

Zookeeper是一个树形结构，很多操作都要先检查才能确定是否可以执行，比如P1的事务t1可能是创建节点“/a”，t2可能是创建节点“/a/bb”，只有先创建了父节点“/a”，才能创建子节点“/a/bb”。

为了保证这一点，ZAB要保证同一个Leader发起的事务要按顺序被apply，同时还要保证只有先前Leader的事务被apply之后，新选举出来的Leader才能再次发起事务。

- 3) 当主进程出现异常的时候，整个zk集群依旧能正常工作。

ZAB协议原理

ZAB协议要求每个Leader都要经历三个阶段：发现、同步、广播。

- **发现**：要求zookeeper集群必须选举出一个Leader进程，同时Leader会维护一个Follower可用客户端列表。将来客户端可以和这些Follower节点进行通信。
- **同步**： Leader要负责将本身的数据与Follower完成同步，做到多副本存储。这样也是体现了CAP中的高可用和分区容错。Follower将队列中未处理完的请求消费完成后，写入本地事务日志中。
- **广播**： Leader可以接受客户端新的事务Proposal（建议）请求，将新的Proposal请求广播给所有的Follower。

ZAB协议核心

ZAB协议的核心：定义了事务请求的处理方式

- 1) 所有的事务请求必须由一个全局唯一的服务器来协调处理，这样的服务器被叫做**Leader服务器**。其他剩余的服务器则是**Follower服务器**。
- 2) Leader服务器，负责将一个客户端事务请求，转换成一个**事务Proposal**，并将该Proposal分发给集群中的所有Follower服务器，也就是向Follower节点发送数据广播请求（或数据复制）
- 3) 分发之后Leader服务器需要等待所有Follower服务器的反馈（Ack请求），在**ZAB协议中，只要超过半数的Follower服务器进行了正确的反馈后**（也就是收到半数以上的Follower的Ack请求），那么Leader就会再次向所有的Follower服务器发送Commit消息，要求其将上一个事务proposal进行提交。

ZAB协议内容

ZAB协议包括两种基本的模式：崩溃恢复和消息广播

协议过程

当整个集群启动过程中，或者当Leader服务器出现网络中断、崩溃退出或重启等异常时，ZAB协议就会进入**崩溃恢复模式**，选举产生新的Leader。

当选举产生了新的Leader，同时集群中有过半的机器与该Leader服务器完成了状态同步（即数据同步）之后，ZAB协议就会退出崩溃恢复模式，进入**消息广播模式**。

这时，如果有一台遵守ZAB协议的服务器加入集群，因此此时集群中已经存在一个Leader服务器在广播消息，那么该新加入的服务器自动进入恢复模式；找到Leader服务器，并且完成数据同步。同步完成后，作为新的Follower一起参与到消息广播流程中。

协议状态切换

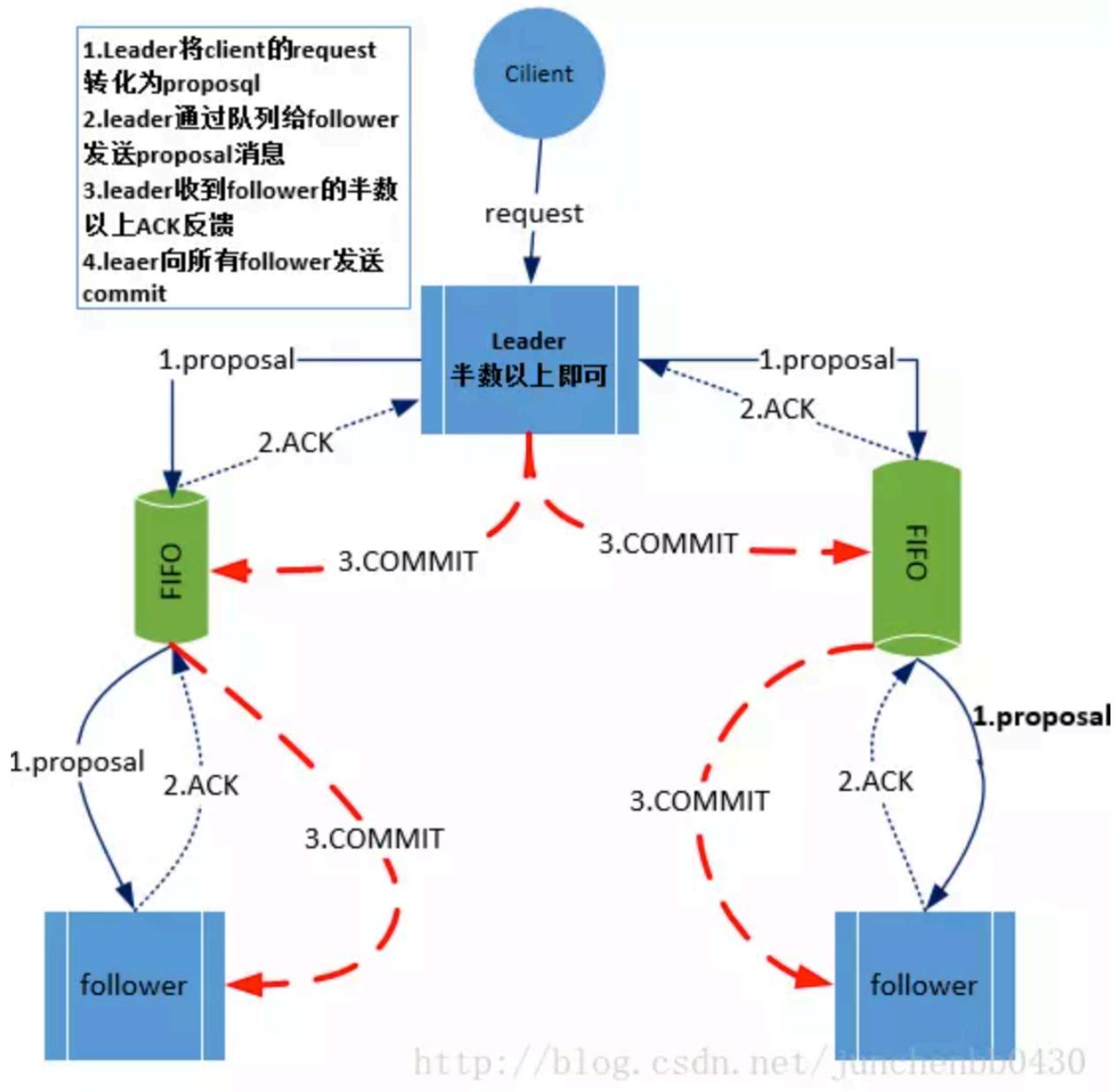
当Leader出现崩溃退出或者机器重启，亦或是集群中不存在超过半数的服务器与Leader保存正常通信，ZAB就会再一次进入**崩溃恢复**，发起新一轮Leader选举并实现数据同步。同步完成后又会进入**消息广播模式**，接收事务请求。

保证消息有序

在整个消息广播中，Leader会将每一个事务请求转换成对应的proposal来进行广播，并且在广播事务proposal之前，Leader服务器会首先为这个事务proposal分配一个全局单递增的唯一ID，称之为**事务ID**（即zid），由于ZAB协议需要保证每一个消息的严格的顺序关系，因此，必须将每一个proposal按照其zid的先后顺序进行排序和处理。

消息广播

- 1) 在zookeeper集群中，数据副本的传递策略就是采用消息广播模式。zookeeper中数据副本的同步方式与二段提交相似，但是却又不同。二段提交要求协调者必须等到所有的参与者全部反馈ACK确认消息后，再发送commit消息。要求所有的参与者要么全部成功，要么全部失败。二段提交会产生严重的阻塞问题。。
- 2) ZAB协议中Leader等待Follower的ACK反馈消息是指“只要半数以上的Follower成功反馈即可，不需要收到全部Follower反馈”



消息广播具体流程

1. 客户端发起一个写操作请求
2. Leader服务器将客户端的请求转化为事务proposal提案，同时为每个proposal分配一个全局的ID，即zxid。
3. Leader服务器为每个Follower服务器分配一个单独的队列，然后将需要广播的proposal依次放到队列中去，并且根据FIFO策略进行消息发送。

4. Follower接收到proposal后，会首先将其以事务日志的方式写入本地磁盘中，写入成功后向Leader反馈一个ack响应消息。
5. Leader接收到超过半数以上Follower的ack响应消息后，即认为消息发送成功，可以发送commit消息。
6. Leader向所有的Follower广播commit消息，同时自身也会完成事务提交。Follower接收到commit消息后，会将上一条事务提交。

zookeeper采用zab协议的核心，就是要有一台服务器提交了proposal，就要确保所有的服务器最终都能正确提交proposal。这也是CAP/BASE实现最终一致性的一个体现。

Leader服务器与每一个Follower服务器之间都维护了一个单独的FIFO消息队列进行收发消息，使用队列消息可以做到异步解耦。Leader和Follower之间只需要往队列中发送消息即可。如果使用同步的方式会引起阻塞，性能要下降很多。

崩溃恢复

一旦Leader服务器出现崩溃或者由于网络原因导致Leader服务器失去了与过半Follower的联系，那么就会进入崩溃恢复模式。

在ZAB协议中，为了保证程序的正确运行，整个恢复过程结束后需要选举出一个新的Leader服务器。因此ZAB协议需要一个高效且可靠的Leader选举算法，从而确保能够快速选举出新的Leader。

Leader选举算法不仅仅需要让Leader自己知道自己已经被选举为Leader，同时还需要让集群中的所有其他机器也能够快速感知到选举产生的新Leader服务器。

崩溃恢复主要包括两部分：**Leader选举**和**数据恢复**。

ZAB协议如何保证数据一致性

假设两种异常情况：

1. 一个事务在Leader上提交了，并且过半的Follower都响应ack了，但是Leader在commit消息发出之前挂了。
2. 假设一个事务在Leader提出之后，Leader挂了。

要确保如果发生上述两种情况，数据还能保持一致性，那么ZAB协议选举算法必须满足以下要求：

ZAB协议崩溃恢复要求满足一下两个要求：

- 1) 确保已经被Leader提交的proposal必须最终被所有的Follower服务器提交。
- 2) 确保丢弃已经被Leader提出的但是没有被提交的proposal。

根据上述要求：

ZAB协议需要保证选举出来的Leader需要满足以下条件：

- 1) 新选举出来的Leader不能包含未提交的proposal

即新选举的Leader必须都是已经提交了proposal的follower服务器节点。

- 2) 新选举的Leader节点中含有最大的zxid

这样做的好处是可以避免Leader服务器检查proposal的提交和丢弃工作。

ZAB如何数据同步

- 1) 完成Leader选举后（新Leader具有最高的zxid），在正式开始工作之前（接收事务请求，然后提出新的proposal），Leader服务器会首先确认事务日志中的所有的proposal是否已经被集群中过半的服务器commit。
- 2) Leader服务器需要确保所有的follower服务器能够接收到每一条事务的proposal，并且能将所有已经提交的事务proposal应用到内存数据中。等到follower将所有尚未同步的事务proposal都从Leader服务器上同步过来，并且应用到内存数据中以后，Leader才会把该follower加入到真正可用的follower列表中。

ZAB丢弃的proposal

在ZAB数据同步过程中，如何处理需要丢弃的proposal

在ZAB的事务编号zxid设计中，zxid是一个64位的数字。

其中低32位可以看成一个简单的单增计数器，针对客户端每一个事务请求，Leader在产生新的proposal事务时，都会对该计数器+1。

而高32位则代表了Leader周期的epoch编号。

epoch编号可以理解为当前集群所处的年代，或者周期。

每次Leader变更之后都会在epoch的基础上+1，这样旧的Leader崩溃恢复之后，其他Follower也不会听它的了，因为Follower只服从epoch最高的Leader命令。

每当选举产生一个新的Leader，就会从这个Leader服务器上取出本地事务日志中最大编号proposal的zxid，并从zxid中解析得到对应的epoch编号，然后在其+1，之后该编号就作为新的epoch值，并将低32位数字归零，由0开始重新生成zxid。

ZAB协议通过epoch变化来区分Leader变化周期，能够有效避免不同的Leader错误的使用了相同的zxid编号提出了不一样的proposal的异常情况。

基于以上策略

当一个包含了上一个Leader周期中尚未提交过的事务proposal的服务器启动时，当这台机器加入集群中，以Follower角色连上Leader服务器后，Leader服务器会根据自己服务器上最后提出的proposal来和follower服务器的proposal进行比对，比对的结果肯定是Leader要求follower进行一个回退操作，回退到一个确实已经被集群中过半机器commit的最新proposal。

实现原理

ZAB节点（实例）有三种状态：

- Following：当前节点是跟随者，服从Leader节点的命令。
- Leading：当前节点是Leader，负责协调事务。
- Election/Looking：节点处于选举状态，正在寻找Leader。

代码实现中，多了一种状态：Observing状态

这是Zookeeper引入Observer之后加入的，Observer不参与选举，是只读节点，跟ZAB协议没有关系。

节点的持久状态：

- history：当前节点接收到事务proposal的Log
- acceptedEpoch：Follower已经接受的Leader更改epoch的newEpoch提议。
- currentEpoch：当前所处的Leader年代
- lastZxid：history中最近接收到的proposal的zxid（最大zxid）

ZAB的四个阶段

选举阶段（Leader Election）

节点一开始都处于选举节点，只要有一个节点得到超过半数节点的票数，它就可以当选准Leader，只有到达第三阶段（也就是同步阶段），这个准Leader才会成为真正的Leader。

Zookeeper规定所有有效的投票都必须在同一个轮次中，每个服务器在开始新一轮投票时，都会对自己维护的LogicalClock进行自增操作。

每个服务器在广播自己的选票前，会将自己的投票箱清空。该投票箱记录了所收到的选票。

例如：server2投票给server3，server3又投给了server1，那么数据结构就是那么server1的投票箱为(2, 3)、(3, 1)、(1, 1)。默认给自己投票。

前一个数字表示投票者，后一个数字表示被选举者。票箱中只会记录每一个投票者的最后一次投票记录，如果投票者更新自己的选票，则其他服务器收到该新选票后会在给自己的票箱中更新该服务器的选票。

这个阶段的目的就是为了选出一个准Leader，然后进入下一个阶段。

发现阶段（Discovery）

在这个阶段，follower和上一轮选举出的准Leader进行通信，同步follower最近接收的事务proposal。

一个follower只会连接一个leader，如果一个follower节点认为另一个follower节点，则会在尝试连接时被拒绝。被拒绝之后，该节点就会进入Leader Election阶段。

这个阶段的主要目的是发现当前大多数节点接收的最新proposal，并且准Leader生成新的epoch，让follower接收，更新他们的acceptedEpoch。

同步阶段（Synchronization）

同步阶段主要是利用Leader前一阶段获得的最新proposal历史，同步集群中所有的副本。

只有当quorum（超过半数的节点）都同步完成，准Leader才会成为真正的Leader。follower只会接收zxid比自己lastZxid大的proposal。

广播阶段（Broadcast）

到了这个阶段，zookeeper集群才能正式对外提供事务服务，并且Leader可以进行消息广播。

同时，如果有新的节点假如，还需要对新节点进行同步。

需要注意的是，ZAB提交事务并不像2PC一样需要全部follower都ack，只需要得到quorum（超过半数的节点）的ack就可以。

分布式事务

网上购物，明明已经扣款，但是却告诉我没有发生交易。这一系列情况都是因为没有事务导致的。这说明了事务在生活中的一些重要性。

有了事务，你去小卖铺买东西，那就是一手交钱一手交货。有了事务，你去网上购物，扣款即产生订单交易。

事务的具体定义

事务提供一种机制将一个活动设计的所有操作纳入到一个不可分割的执行单元，组成事务的所有操作只有在所有操作均能正常执行的情况下方能提交，只要其中任一操作执行失败，都将导致整个事务的回滚。

简单地说，事务提供一种“要么什么都不做，要么做全套（All or Nothing）”机制。

数据库本地事务

ACID

说到数据库事务就不得不说，数据库事务中的四大特性ACID：

A：原子性（Atomicity），一个事务（transaction）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。

事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

就像你买东西要么交钱收货一起都执行，要么发不出货，就退钱。

C：一致性（Consistency），事务的一致性指的是在一个事务执行之前和执行之后数据库都必须处于一致性状态。

如果事务成功地完成，那么系统中所有变化将正确地应用，系统处于有效状态。

如果在事务中出现错误，那么系统中的所有变化将自动地回滚，系统返回到原始状态。

I：隔离性（Isolation），指的是在并发环境中，当不同的事务同时操纵相同的数据时，每个事务都有各自的完整数据空间。

由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。事务查看数据更新时，数据所处的状态要么是另一事务修改它之前的状态，要么是另一事务修改它之后的状态，事务不会查看到中间状态的数据。

打个比方，你买东西这个事情，是不影响其他人的。

D：持久性（Durability），指的是只要事务成功结束，它对数据库所做的更新就必须保存下来。

即使发生系统崩溃，重新启动数据库系统之后，数据库还能恢复到事务成功结束时的状态。

打个比方，你买东西的时候需要记录在账本上，即使老板忘记了，那也有据可查。

InnoDB事务实现

InnoDB是MySQL的一个存储引擎，大部分人对MySQL都比较熟悉，这里简单介绍一下数据库事务实现的一些基本原理。

在本地事务中，服务和资源在事务的包裹下可以看做是一体的，如下图：我们的本地事务由资源管理器进行管理：

而事务的ACID是通过InnoDB日志和锁来保证。事务的隔离性是通过数据库锁的机制来实现的，持久性通过Redo Log（重做日志）来实现，原子性和一致性通过Undo Log实现。

Undo Log的原理很简单，为了满足事务的原子性，在操作任何数据之前，首先将数据备份到一个地方（这个存储数据备份的地方称为Undo Log）。然后进行数据的修改。

如果出现了错误或者用户执行了Rollback语句，系统可以利用Undo Log中的备份将数据恢复到事务开始之前的状态。

和Undo Log相反，Redo Log记录的是新数据的备份。在事务提交前，只要将Redo Log持久化即可，不需要将数据持久化。

当系统崩溃时，虽然数据没有持久化，但是Redo Log已经持久化。系统可以根据Redo Log的内容，将所有数据恢复到原始的状态。对具体实现过程有兴趣的同学可以自行搜索扩展。

什么是分布式事务

分布式事务指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。

简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。

本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

分布式事务产生的原因

从上面本地事务来看，我们可以分为两块：

- service 产生多个节点
- resource 产生多个节点

Service多个节点

随着互联网快速发展，微服务，SOA等服务架构模式正在被大规模的使用。

举个简单的例子，一个公司之内，用户的资产可能分为好多个部分，比如余额，积分，优惠券等等。

在公司内部有可能积分功能由一个微服务团队维护，优惠券又是另外的团队维护。

这样的话就无法保证积分扣减了之后，优惠券能否扣减成功。

Resource多个节点

同样的，互联网发展得太快了，我们的MySQL一般来说装的数据就得进行分库分表。

对于一个支付宝的转账业务来说，你给朋友转钱，你有朋友转钱，有可能你的数据库是在北京，而你的朋友的钱是存在伤害，所以我们依然无法保证他们能同时成功。

分布式事务的基础

从上面来看，分布式事务是随着互联网高速发展应运而生的，这是一个必然。

我们之前说过数据库的ACID四大特性，已经无法满足我们分布式事务，这个时候又有一些新的大佬提出一些新的理论。

CAP

CAP定理，又被叫做布鲁尔定理。对于设计分布式系统（不仅仅是分布式事务）的架构师来说，CAP就是你的入门理论。

C（一致性）：对某个指定的客户端来说，读操作能返回当前的写操作。

对于数据分布在不同节点上的数据来说，如果在某个节点更新了数据，那么在其他节点如果都能读取到这个人写的数据，那么就称为强一致，如果有某个节点没有读取到，那就是分布式不一致。

A（可用性）：非故障的节点在合理的时间内返回合理的响应（不是错误和超时的响应）。可用性的两个关键一个是合理的时间，一个是合理的响应。

合理的时间指的是请求不能被阻塞，应在合理的时间给出返回。合理的响应指的是系统应该明确返回结果并且结果是正确的，这里的正确指的是比如应该返回50，而不是返回40。

P（分区容错性）：当出现网络分区后，系统能够继续工作。打个比方，这里集群有多台机器，有台机器网络出现了问题，但是这个集群仍然可以正常工作。

熟悉CAP的人都知道，三者不能共有，如果感兴趣可以搜索CAP的证明，在分布式系统中，网络无法100%可靠，分区其实是一个必然现象。

如果我们选择了CA而放弃了P，那么当发生分区现象时，为了保证一致性，这个时候必须拒绝请求，但是A又不允许，所以分布式系统理论上不可能选择CA架构，只能选择CP或者AP架构。

对于CP来说，放弃可用性，追求一致性和分区容错性，我们Zookeeper其实就是追求的强一致性。

对于AP来说，放弃一致性（这里说的一致性是强一致性），追求分区容错性和可用性，这是很多分布式系统设计时的选择，后面的BASE也是根据AP来扩展。

顺便一提，CAP理论中是忽略网络延迟，也就是当事务提交时，从节点A复制到节点B没有延迟，但是在现实中这个是明显不可能的，所以总会有一定的时间是不一致。

同时CAP中选择两个，比如你选择了CP，并不是叫你放弃A。因为P出现的概率实在是太小了，大部分的时间你仍然需要保证CA。

就算分区出现了你也要为后来的A做准备，比如通过了一些日志的手段，是其他机器恢复至可用。

BASE

BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的缩写，是对CAP中AP的一个扩展。

基本可用：分布式系统在出现故障时，允许损失部分可用功能，保证核心功能可用。

软状态：允许系统中存在中间状态，这个状态不影响系统可用性，这里指的是CAP中的不一致。

最终一致性：最终一致性是指经过一段时间后，所有节点数据都将会达到一致。

BASE解决了CAP中理论没有网络延迟，在BASE中用软状态和最终一致，保证了延迟后的一致性。

BASE和ACID是相反的，它完全不同于ACID的强一致性模型，而是通过牺牲强一致性来获得可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态。

分布式事务解决方案

在说方案之前，首先你一定要明确你是否真的需要分布式事务？

上面说过出现分布式事务的量大原因，其中有个原因是微服务过多。我见过太多团队一个人维护几个微服务，太多团队过度设计，搞得所有人疲劳不堪。

而微服务过多就会引出分布式事务，这个时候我不会建议你去采用下面任何一种方案，而是请把需要事务的微服务聚合成一个单机服务，使用数据库的本地事务。

因为不论任何一种方案都会增加你系统的复杂度，这样的成本实在是太高了，千万不要因为追求某些设计，而引入不必要的成本和复杂度。

如果你确定需要引入分布式事务可以看看下面几种常见的方案。

2PC（两阶段提交）

两阶段提交（Two-phase Commit, 2PC），通过引入协调者（Coordinator）来协调参与者的行为，并最终决定这些参与者是否要真正执行事务。

1. 运行过程

1.1 准备阶段

协调者询问参与者事务是否执行成功，参与者发回事务执行结果。

1.2 提交阶段

如果事务在每个参与者上都执行成功，事务协调者发送通知让参与者提交事务；否则，协调者发送通知让参与者回滚事务。

需要注意的是，在准备阶段，参与者执行了事务，但是还未提交。只有在提交阶段接收到细条这发来的通知后，才进行提交或者回滚。

2. 存在的问题

同步阻塞，所有事务参与者在等待其他参与者响应的时候都处于同步阻塞状态，无法进行其他操作。

单点问题，协调者在2PC中起到非常大的作用，发生故障将会造成很大影响。特别是在阶段二发生故障，所有参与者会一直等待状态，无法完成其他操作。

数据不一致，在阶段二，如果协调者只发送了部分commit消息，此时网络发生异常，那么只有部分参与者接收到commit消息，也就是说只有部分参与者提交了事务，使得系统数据不一致。

太过保守，任意一个节点失败就会导致整个事务失败，没有完善的容错机制。

TCC（补偿事务）

TCC其实就是采用的补偿机制，其核心思想：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：

- Try阶段主要是对业务系统做检测及资源预留
- Confirm阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行Confirm阶段时，默认Confirm阶段时不会出错的。即：只要Try成功，Confirm一定成功
- Cancel阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。

举个例子，假如Bob要向Smith转账，思路大概是：我们有一个本地方法，里面依次调用：

1. 首先在Try阶段，需要调用远程接口把Smith和Bob的钱给冻结起来
2. 在Confirm，执行远程调用的转账操作，转账成功进行解冻。
3. 如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法（Cancel）。

优点：跟2PC比起来，实现以及流程相对简单一些，但数据的一致性比2PC也要差一些

缺点：缺点还是比较明显的，在2，3步中都有可能失败。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理。

本地消息表（异步确保）

本地消息表与业务数据表处于同一个数据库中，这样就能利用本地事务来保证在对这两个表的操作满足事务特性，并且使用了消息队列来保证最终一致性。

1. 在分布式事务操作的一方完成写业务数据的操作，之后向本地消息表发送一个消息，本地事务能保证这个消息一定会被写入本地消息表中
2. 之后将本地消息表中的消息转发到kafka等消息队列中，如果转发成功则将消息从本地消息表中删除，否则继续重新转发。
3. 在分布式事务操作的另一方，从消息队列中读取一个消息，并执行消息中的操作。

优点：一种非常经典的实现，避免了分布式事务，实现了最终一致性。

缺点：消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。

MQ事务消息

有一些第三方的MQ是支持事务消息的，比如RocketMQ，他们支持事务消息的方式也是类似于采用二阶段提交，但是市面上一些主流的MQ都是不支持事务消息的，比如RabbitMQ和kafka都不支持。

以阿里的RocketMQ中间件为例，其思路大致为：

第一阶段Prepared消息，会拿到消息的地址。第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。

也就是说在业务方法内，想要消息队列提交两次请求：一次发送消息和一次确认消息。

如果确认消息发送失败了，RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发送者确认，所以生产方需要实现一个check接口，RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。

优点：实现了最终一致性，不需要依赖本地数据库事务。

缺点：实现难度大，主流MQ不支持，RocketMQ事务消息部分代码也未开源。

总结

从以上论述，我们总结对比了集中分布式分解方案的优缺点，分布式事务本身是一个技术难题，是没有一种完美的方案应对所有场景的，具体还是要根据业务场景去抉择吧。阿里RocketMQ去实现的分布式事务，现在也出了很多分布式事务的协调器，比如LCN等。

Kafka保证高可用的原因

数据可靠性

kafka作为一个商业级消息中间件，消息可靠性的重要性可想而知。

Topic分区副本

kafka从0.8.0版本，开始引入分区副本。也就是说每个分区可以认为的配置几个副本（比如创建主题的时候replication-factor，也可以在broker级别进行配置default.replication.factor），一般会设置为3。

kafka可以保证单个分区里的时间是有序的，分区可以在线（可用），也可以离线（不可用）。在众多的分区副本里面有一个副本是Leader，其余的副本是follower，所有的读写操作都是经过Leader进行的，同时follower会定期地从leader上复制数据。当Leader挂了的时候，其中一个follower会重新成为新的Leader。通过分区副本，引入了数据冗余，同时也提供了kafka的数据可靠性。

kafka的分区多副本架构是kafka可靠性保证的核心，把消息写入多个副本可以使kafka在发生崩溃时仍能保证消息的持久性。

Producer往Broker发送消息

如果我们要往kafka对应的主题发送消息，我们需要通过producer完成。前面我们讲过kafka主题对应了多个分区，每个分区下面又对应了多个副本；为了让用户设置数据可靠性，kafka在producer里面提供了消息确认机制。也就是说我们可以通过配置来决定消息发送到对应分区的几个副本才算消息发送成功。可以在定义producer时通过acks参数指定。（在0.8.2.X版本之前是通过request.required.acks参数设置的，详见kafka-3043）。这个参数支持以下三种值：

- acks=0；意味着如果生产者能够通过网络把消息发送出去，那么就认为消息已成功写入kafka。在这种情况下还是有可能发生错误，比如发送的对象无法被序列化或者网卡发生故障，但如果是分区离线或整个集群长时间不可用，那就不会收到任何错误。在acks=0模式下的运行速度是非常快的（这就是为什么很多基准测试都是基于这个模式），你可以得到惊人的吞吐量和带宽利用率，不过如果选择了这种模式，一定会丢失一些消息。
- acks=1；意味着若Leader在收到消息并把它写入到分区数据文件（不一定同步到磁盘上）时，会返回确认或错误响应。在这个模式下，如果发生正常的Leader选举，生产者会在选举时收到一个Leader NotAvailableException异常，如果生产者能恰当地处理这个错误，它会重试发送消息，最终消息会安全到达新的Leader那里。不过在这个模式下，仍然可能会丢失数据，比如消息已经成功写入Leader，但在消息被复制到follower副本之前Leader发生崩溃。
- acks=all（这个和request.required.acks=-1含义一样）；意味着Leader在返回确认或错误响应之前，会等待所有同步副本都收到消息。如果和min.insync.replicas参数结合起来，就可以决定在返回确认前至少有多少个副本能够收到消息，生产者会一直重试，直到消息被成功提交。不过这也是最慢的做法，因为生产者在继续发送其他消息之前，需要等待所有副本都收到当前的消息。

根据实际的应用场景，我们设置不同的acks，以此保证数据的可靠性。

另外，Producer发送消息还可以选择同步（默认，通过producer.type=sync配置）或者异步（producer.type=async）模式。

如果设置成异步，虽然会极大的提高消息发送的性能，但是这样会增加丢失数据的风险。

如果需要确保消息可靠性，必须将producer.type设置为sync。

Leader选举

在介绍Leader选举之前，先了解一下ISR (in-sync-replicas) 列表。每个分区的Leader会维护一个ISR列表，ISR列表里面就是follower副本的Broker编号，只有跟得上Leader的follower副本才能加入到ISR里面，这个是通过replica.lag.time.max.ms参数配置的。只有ISR里的成员才有被选为Leader的可能。

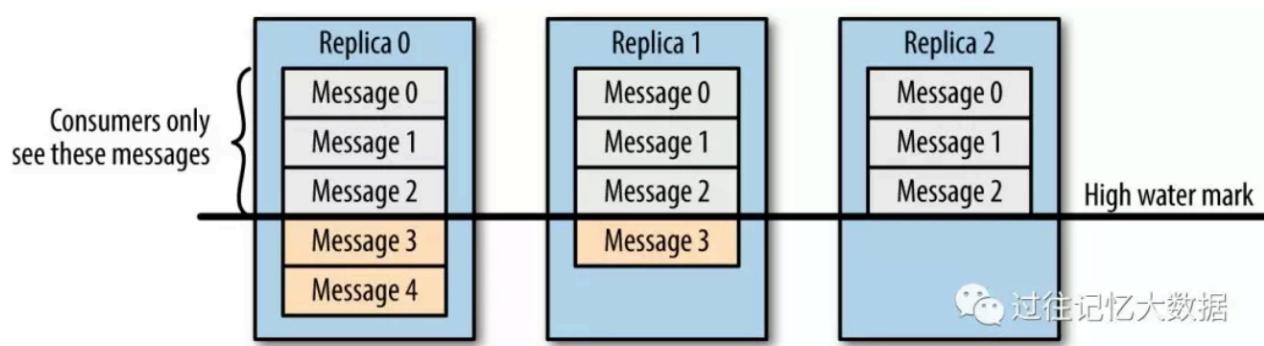
所以，当Leader挂掉了，而且unclean.leader.election.enable=false的情况下，kafka会从ISR列表中选择第一个follower作为新的Leader，因为这个分区拥有最新的已经committed的消息。通过这个可以保证已经committed的消息的数据可靠性。

综上所述，为了保证数据的可靠性，我们最少需要配置一下几个参数：

- producer级别：acks=all（或者request.required.acks=-1），同时生产模式为同步 producer.type"sync"
- topic级别：设置replication.factor>=3，并且min.insync.replicas>=2
- broker级别：关闭不完全的Leader选举，即unclean.leader.enable=false；

数据一致性

数据一致性主要是说不论是老的Leader还是新选举的Leader，Consumer都能读到一样的数据。那么kafka是如何实现的呢？



假设分区的副本为3 (replicas=3)，其中副本0是Leader，副本1和副本2是follower，并且在ISR列表里面。虽然副本0已经写入了Message4，但是Consumer只能读取到Message2。因为所有的ISR都同步了Message2，只有 High Water Mark 以上的消息才支持Consumer读取，而 High Water Mark 取决于ISR列表里面偏移量最小的分区，对应于上图的副本2，这个很类似于木桶原理。

这样做的原因是还没有被足够多副本复制的消息被认为是“不安全”的，如果Leader发生崩溃，另一个副本成为新Leader，那么这些消息很可能丢失。如果我们允许消费者读取这些消息，可能就会破坏一致性。

试想，一个消费者从当前Leader（副本0）读取并处理了Message4，这个时候Leader挂掉了，选举了副本1为新的Leader，这时候另一个消费者再去从新的Leader读取消息，发现这个消息其实并不存在，这就导致了数据不一致性问题。

当然，引入了 High Water Mark 机制，会导致 Broker 间的消息复制因为某些原因变慢，那么消息到达消费者的时间也会随之变长（因为我们会先等待消息复制完毕）。延迟时间可以通过参数 replica.lag.time.max.ms 配置，它指定了副本在复制消息时可被允许的最大延迟时间。

Redis 高可用的原因

高可用 (HA-High Availability)

是当一台服务器停止服务之后，对于业务及用户毫无影响。停止服务的原因可能由于网卡、路由器、机房、CPU 负载过高、内存溢出、自然灾害等不可预期的原因导致，在很多时候也称单点问题。

主备方式 (简单情景)

这种通常是一台主机，一台或多台备机，在正常情况下主机对外提供服务，并把数据同步到备机，当主机宕机后，备机立刻开始服务。Redis HA 中使用比较多的是 keepalive，它使主机备机对外提供同一个虚拟 IP，客户端通过虚拟 IP 进行数据操作，正常期间主机一直对外提供服务，宕机后 VIP 自动漂移到备机上。

优点：客户端毫无影响，仍然通过 VIP 操作。

缺点：在绝大多数时间内备机是一直没使用的，被浪费着的。

主从方式 (推荐)

这种采取一主多从的办法，主从之间进行数据同步。当 Master 宕机后，通过选举算法 (Paxos、Raft) 从 slave 中选举出新 Master 继续对外提供服务，主机恢复后以 slave 的身份重新加入。主从另一个目的是进行读写分离，这是当单机读写压力过高的一种通用型解决方案。

其主机的主要角色只提供写操作或少量的读，把多余的读请求通过负载均衡算法分流到单个或多个 slave 服务器上。

缺点：主机宕机后，slave 虽然被选举成 master 了，但是对外提供 IP 服务地址却发生了变化，意味着会影响到客户端。解决这种情况需要一些额外的工作，在当主机地址发生变化后，及时通知到客户端，客户端收到新地址后，使用新地址继续发送新请求。

方案选择

主备 (keepalive) 方案配置简单、人力成本小，在数据量小、压力小的情况下推荐使用。如果数据量较小，不希望过多浪费机器，还希望在宕机后，做一些自定义的措施，比如报警、记日志、数据迁移等操作，推荐使用主从方式，因为和主从搭配的一般还有个管理监控中心。

主从拓扑

Redis 的主从拓扑支持单层或者多层结构，可分为一主一从，一主多从，树状主从。

- 一主一从

用于主节点故障转移到从节点，当主节点的“写”命令并发高且需要持久化，可以只在从节点开启 AOF (主节点不需要)，这样既保证了数据的安全性，也避免持久化对主节点的影响。

- 一主多从

针对“读”较多场景，“读”由多个从节点来分担，但节点越多，主节点同步到多节点的次数也越多，影响带宽，也加重节点的负担。

- 树状主从

一主多从的缺点（主节点推送次数多压力大）可用写方案解决，主节点只推送一次数据到从节点1，再有节点2推送到11，减轻节点推送的压力。

Redis的数据同步方式

无论是主备还是主从都牵扯到数据同步的问题，这也分2种情况：

1. 同步方式；当主机收到客户端写操作后，以同步方式把数据同步到从机。
2. 异步方式：主机接收到写操作后，直接返回成功，然后再后台用异步方式把数据同步到从机上。

redis高并发跟整个系统的关系

redis，你要搞高并发的话，不可避免，要把底层的缓存搞得很好

mysql，高并发，做到了，那么也是通过一些列复杂的分库分表，订单系统，事务要求的，QPS到几万，比较高了。

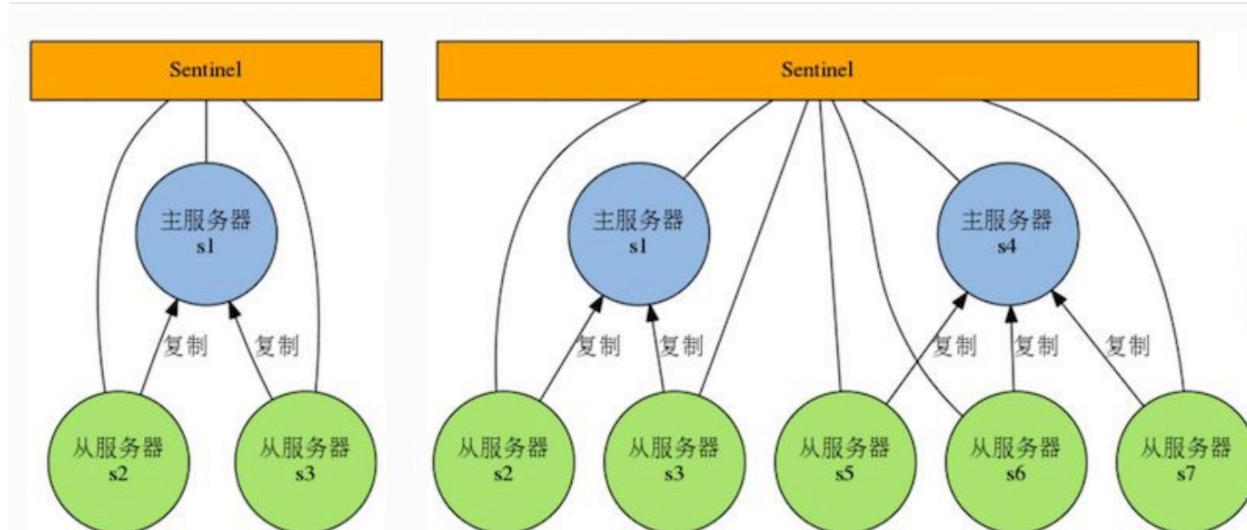
要做一些电商的商品详情页，真正的超高并发，QPS上10w，甚至百万，一秒钟百万的请求量

光是redis是不够的，但是redis是整个大型的缓存架构中，支撑高并发的架构里面，非常重要的一个环节

首先，底层的缓存中间件，缓存系统，必须能够支撑的起我们说的那种高并发，其次，再经过良好的整体的缓存架构设计（多级缓存架构、热点缓存），支撑真正的上十万，甚至上百万的高并发。

哨兵（sentinel）

Redis-Sentinel是Redis官方推荐的高可用性（HA）解决方案，当用Redis做Master-slave的高可用方案时，假如master宕机了，Redis本身（包括它的很多客户端）都没有实现自动进行主备切换，而Redis-sentinel本身也是一个独立运行的进程，它能监控多个master-slave集群，发现master宕机后能进行自动切换。



它的主要功能有以下几点：

- 监控（Monitoring），不时地监控redis（master和slave）是否按照预期良好地运行；
- 提醒（Notification），如果发现某个redis节点运行出现状况，能够通知另外一个进程（例如它

的客户端)；

- **自动故障迁移（Automatic failover）**，能够进行自动切换。当一个master节点不可用时，能够选举出master的多个slave（如果有超过一个slave的话）中的一个来作为新的master，其他的slave节点会将它所追随的master的地址改为被提升为master的slave的新地址。

哨兵原理

Redis哨兵的三个定时任务，Redis哨兵判定一个Redis节点故障不可达，主要就是通过三个定死监控任务来完成的。

- (Job1) 每隔10秒，每个哨兵节点会向主节点和从节点发送“`info replication`”命令，来获取最新的拓扑结构
- (Job2) 每隔2秒，每个哨兵节点会向Redis节点的`_sentinel_:hello`频道发送自己对主节点是否故障的判断以及自身的节点信息，并且其他的哨兵节点也会订阅这个频道来了解其他哨兵节点的信息以及对主节点的判断
- (Job3) 每隔1秒，每个哨兵会向主节点、从节点、其他的哨兵节点发送一个“`ping`”命令来做心跳检测。

如果定时任务Job3检测不到节点的心跳，会判断为“主观下线”。如果该节点还是主节点那么还会通知到其他的哨兵对该节点进行心跳检测，这时主观下线的票数超过了`<quorum>`数时，那么这个主节点确实就可能是故障不可达了，这时就由原来的主观下线变为了“客观下线”。

故障转移和Leader选举

如果主节点被判定为客观下线之后，就要选取一个哨兵节点来完成后面的故障转移工作，选举出一个Leader，这里面采用的选举算法为Raft。选举出来的哨兵Leader就要来完成故障转移工作，也就是在从节点中选出一个节点来当新的主节点，这部分的具体流程可参考引用。

主从复制的断点续传

从redis2.8开始，就支持主从复制的断点续传，如果主从复制过程中，网络连接断掉了，那么可以接着上次复制的地方，继续复制下去，而不是从头开始复制一份。

腾讯会议，工作任务：

1. 业务开发，高并发不是很多
2. 把流量转发到cdn等和下游视频编解码

高并发

OpenResty

用于做反向代理层的高并发，这个需要了解一下。

当然，每个人对高并发的理解可能不太一样，有人说1000并发就是高并发了，有人说1万的并发才是高并发，有人说并发百万才是高并发，OpenResty是可以做到百万并发的（当然需要各种调优），现在大部分业务OpenResty都可以胜任，但是像腾讯10亿用户，1亿的并发，OpenResty就搞不定了。

不同的并发量要应对的东西不一样，比如1000并发，用tomcat，springmvc框架加缓存就可以应对，1万的并发在关键节点使用内存处理也很容易，百万并发就需要linux内核调优，socket缓冲区，文件句柄数，内存池，RPS/RFS SMP等优化也可以达到。千万并发就需要考虑用户态协议dpdk了。

从深入角度来说，OpenResty是基于nginx增加了模块，我们说的其实也就是nginx的性能，而nginx是异步非阻塞的，基于事件驱动的server，相比其他的server在卡住的时候他为什么不卡？

就拿你得问题来说，mysql卡了，这条请求的上下文会被卡在这里，不管是nginx还是apache，都会卡住这条请求，但是问题关键还在于后续请求进来后会怎么办？

apache的做法是开启一个新的进程来处理后续的请求，但系统进程资源是有限的，所以面对大量请求时，进程耗尽，apache就会把所有后续的请求都卡住了。

nginx只有一个master进程和已配置个数的worker进程，master进程把请求交给worker去处理，一个worker在可能出现阻塞的地方会注册一个时间就放过去了（epoll模型），而不是干巴巴的等待阻塞被处理完，他会继续处理后续的请求（非阻塞），当这个时间处理完之后会通过callback来通知worker继续处理那条请求后续的事情（事件驱动），因此，单个worker可以处理大量请求而不会轻易让整个系统卡住。

所以，epoll模型？了解一下

Apigate

前置网关，用于服务IP黑/白名单，频控，熔断。这个也需要着重了解一下。

RedisProxy

用于做redis代理，redis动态扩容的另外一种解决方案。

Redis专题

redis扩容和缩容

redis数据结构

redis主从同步

代码调试

- 使用intelliJ debug
- 或者配置maven spring boot为开启debug模式， 其实就是java虚拟机的启动参数注入
- 使用jps显示java进程pid
- 使用jmap在线调试和观察java进程的各项指标（GC）， 也可以查到哪些对象占用内存
- 使用top -Hp pid查看哪些线程使用的cpu最多
- 使用jstack pid 可以查到具体哪个线程最繁忙（所以给线程取名称很重要）， 打印堆栈信息
- 使用jstat 查看GC
- 使用jinfo可以拿到java启动参数
- 拿到Command line后面的配置参数到perfma中验证查询， 启动参数是否合理<http://xxfox.perfma.com/jvm/check>

一般分析CPU或者内存异常情况可以通过以下几步：

1. 查看日志
2. 查看CPU情况
3. 查看TCP情况
4. 查看java线程， jstack
5. 查看java堆， jmap
6. 通过MAT分析堆文件， 寻找无法被回收的对象