# C++总结

**指针常量读作常量的指针=底层const，表示修饰的变量本身是一个常量**

**常量指针表示的是指针的指向是无法修改的=顶层const**

## -----------------------C++常规问题------------------------

## Golang和C++的区别

性能方面：golang在io密集型比较有优势，golang使用大量协程进行并发操作，协程之间的切换要更节省CPU；而C++在计算密集型比较有优势。

编译时间：C++的编译时间非常慢，而Go的编译时间明显更快。

内存管理：C++new出来的变量需要后续手动释放，否则会造成内存泄露，golang有垃圾回收机制。

局部变量的返回；在c++中，局部变量是分配在栈上的，返回一个局部变量会报错，但是Go中有逃逸分析机制，编译器可以决定是分配在堆上还是栈上，这保证返回局部变量不会报错。

## C和C++的区别

* C++中new和delete是对内存分配的运算符，取代了C中的malloc和free。
* C++中用来做控制态输入输出的iostream类库替代了标准C中的stdio函数库。
* 在C++中，除了值和指针之外，新增了引用。引用型变量是其他变量的一个别名，我们可以认为他们只是名字不相同，其他都是相同的。
* C++中的try/catch/throw异常处理机制取代了标准C中的setjmp()和longjmp()函数。
* 在C++中，允许有相同的函数名，不过它们的参数类型不能完全相同，这样这些函数就可以相互区别，这也就是C++的重载函数，C语言不允许重载。
* C++相对与C增加了一些关键字，如：bool、using、dynamic\_cast、namespace等等

## C++三大特性及其目的

封装：就是将抽象得到的数据和功能相结合，形成一个整体，也就是将数据与操作数据的源代码进行结合，形成”类”，其中数据和函数都是类的成员。**封装的目的**是增强安全性和简化编程，使用者不必了解具体的实现细节，而只是要通过外部接口，特定的访问权限来使用类的成员；因此封装可以隐藏实现细节，使得代码模块化。

继承：C++通过类派生机制来支持继承。被继承的类型称为基类或超类，新产生的类为派生类或子类。保持已有类的特性而构造新类的过程叫做继承，在已有类的基础上新增自己的特性而产生新类的过程称为派生。**继承和派生的目的**是保持已有类的特性并构造新类；能够实现代码重用及代码扩充。

多态：可以概括为“一个接口，多种方法”，程序在运行时才决定调用的函数。C++多态性是通过虚函数来实现的，虚函数允许子类重新定义成员函数，当子类重新定义父类的做法称为函数的重写。而重载则是允许有多个同名的函数，而这些函数的参数列表不同，允许参数个数不同，参数类型不同，或者两者都不同，而重写要求参数类型及个数都相同。**多态的目的**则是为了接口重用，就是说不论传递过来的究竟是哪个类的对象，函数都能够通过同一个接口调用到适应各自对象的实现方法。

## C++内存六大布局

C语言的：栈、堆、全局/静态区、常量区

C++的：栈、堆、全局/静态区、自由存储区、常量区、代码区

（1）栈：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限

（2）堆：就是那些由 new分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个new就要对应一个 delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。

（3）自由存储区：如果说堆是操作系统维护的一块内存，那么自由存储区就是C++中通过new和delete动态分配和释放对象的抽象概念。需要注意的是，自由存储区和堆比较像，但不等价。

（4）全局/静态存储区：全局变量和静态变量被分配到同一块内存中，在以前的C语言中，全局变量和静态变量又分为初始化的和未初始化的，在C++里面没有这个区分了，它们共同占用同一块内存区，在该区定义的变量若没有初始化，则会被自动初始化，例如int型变量自动初始为0。

（5）常量存储区：这是一块比较特殊的存储区，这里面存放的是常量，不允许修改

（6）代码区：存放函数体的二进制代码

堆和自由存储区的区别？？？

从技术上来说，堆（heap）是C语言和操作系统的术语。堆是操作系统所维护的一块特殊内存，它提供了动态分配的功能，当运行程序调用malloc()时就会从中分配，稍后调用free可把内存交还。而自由存储是C++中通过new和delete动态分配和释放对象的抽象概念，通过new来申请的内存区域可称为自由存储区。基本上，所有的C++编译器默认使用堆来实现自由存储，也即是缺省的全局运算符new和delete也许会按照malloc和free的方式来被实现，这时由new运算符分配的对象，说它在堆上也对，说它在自由存储区上也正确。但程序员也可以通过重载操作符，改用其他内存来实现自由存储，例如全局变量做的对象池，这时自由存储区就区别于堆了。

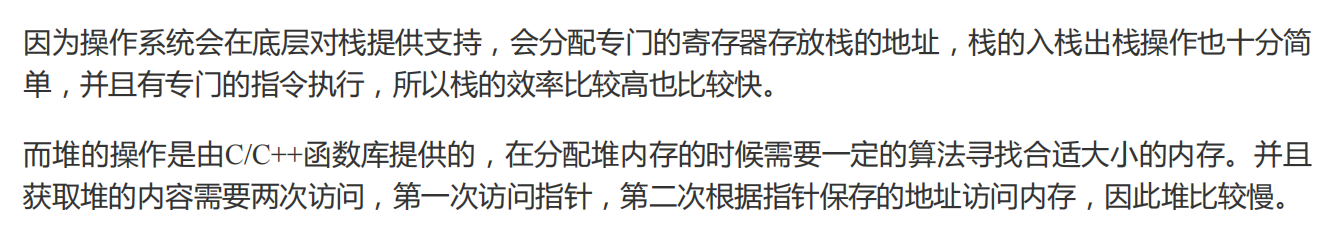
## C++堆和栈的区别

（1）申请方式不同：栈由系统自动分配，堆是自己申请和释放的。

（2）申请大小限制不同：栈顶和栈底是之前预设好的，栈是向栈底扩展，是大小固定的一块连续区域，windows下C++栈的大小是2M，linux可通过ulimit -a查看栈大小，64位机器默认大小为10M，可以通过ulimit -s修改。而堆向高地址扩展，是不连续的内存区域，大小可以灵活调整，操作系统会有一个记录空闲地址的链表，堆的申请过程其实遍历该链表的过程，找到第一个大于申请空间的地址，堆的大小受限于计算机虚拟地址的大小，32位系统理论上虚拟空间是4G（32位可以表示2^32个地址）。

（3）申请效率不同：栈由系统分配，速度快，不会有碎片。堆由程序员分配，速度慢，且会有碎片。

## C++堆和栈哪个快？



操作系统会将空闲的内存块用链表存储起来，所以堆申请的内存是非连续的，每次申请堆内存的时候会从前到后遍历找到第一个大于等于申请大小的内存块从链表中抽出（会在内存头部存放申请的大小方便释放内存）。

## C++ 函数重载和重写（为什么C不能重载）

重载：C++中可以通过上下文来确定同名函数的重载版本。重载函数的关键是函数参数列表，在C++里函数的参数列表也叫做函数特征标，这个特征标主要看的是函数的参数数目和类型，以及参数的排列顺序。所以，重载函数与返回值，参数名无关。

重写：重写是有关多态的一种类型，子类重写父类的虚函数，要求重写与被重写的函数参数列表一定相同。

区别：

1. 定义不同---重载是定义相同的方法名，参数不同;重写是子类重写父类的方法。
2. 范围不同---重载是在一个类中，重写是子类与父类之间的。
3. 多态不同---重载是编译时的多态性，重写是运行时的多态性。
4. 返回不同---重载对返回类型没有要求，而重写要求返回类型必须相同。
5. 参数不同---重载的参数个数、参数类型、参数顺序可以不同，而重写父子方法参数必须相同。
6. 修饰不同---重载对访问修饰没有特殊要求，重写访问修饰符的限制一定要大于被重写方法的访问修饰符。

## c++预处理、编译、汇编、链接的过程（静态链接、动态链接）

预处理 gcc -e a.c -o a.i

编译 gcc -s a.i -o a.s

汇编 gcc -c a.s -o a.o

链接 gcc a.o -o a

**预处理**过程主要处理那些源代码文件中的以“#”开始的预编译指令，比如“#include”，“#define”等。将所有的“#define”删除，并且展开所有的宏定义。

* 处理所有条件预编译指令，比如“#if”，“#ifdef”，“#elif”，“#else”，“#endif”。
* 处理“#include”预编译指令，将被包含的文件插入到该预编译指令的位置。注意，这个过程是递归进行的，也就是说被包含的文件可能还包含其他文件。
* 删除所有的注释“//”和“/--/”。
* 添加行号和文件名标识，比如#2“a.c”2，以便于编译时编译器产生调试用的行号信息及用于编译时产生编译错误或警告时能够显示行号。

**编译**：编译过程就是把预处理完的文件进行一系列词法分析，语法分析，语义分析，代码优化及优化后生成相应的汇编代码文件。

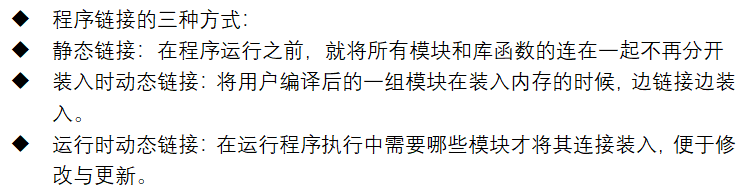
**汇编**：汇编过程就是由汇编器将汇编代码转变成机器可以执行的二进制指令。

**链接**的主要内存就是把各个模块之间相互引用长度部分都处理好，使得各个模块之间能够正确的衔接。简单的理解为将各个目标文件链接起来生成最终的可执行文件。

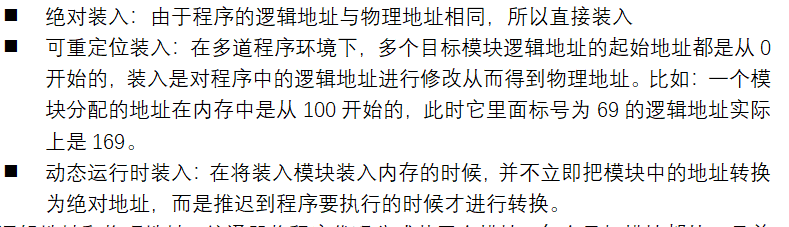
链接过程可以具体的分为以下四步：

* 合并段和符号表，合并多个文件的符号表及各段内容，放入一个新的文件中。
* 符号解析，在每个文件符号引用（引用外部符号）的地方找到符号的定义。这就是符号解析。
* 地址和空间分配，符号解析成功后，为程序分配虚拟地址空间。
* 符号重定位 // 指令段，符号重定向就是对.o文件中.text段指令中的无效地址给出具体的虚拟地址或者相对位移偏移量。

链接又分为：静态链接、动态链接



装入：由装入程序将得到的装入模块装入内存运行。



## 动态库与静态库区别，怎么制作静态库与动态库

动态库与静态库区别，怎么制作静态库与动态库：

静态库（.a 、.lib）动态库（.so 、.dll ）。所谓静态、动态是指链接过程。

（区别）

（1）lib是编译时用到的，dll是运行时用到的。如果要完成源代码的编译，只需要lib；如果要使动态链接的程序运行起来，只需要dll。

（2）如果有dll文件，那么lib一般是一些索引信息，记录了dll中函数的入口和位置，dll中是函数的具体内容；如果只有lib文件，那么这个lib文件是静态编译出来的，索引和实现都在其中。使用静态编译的lib文件，在运行程序时不需要再挂动态库，缺点是导致应用程序比较大，而且失去了动态库的灵活性，发布新版本时要发布新的应用程序才行。

（3）动态链接的情况下，有两个文件：一个是LIB文件，一个是DLL文件。LIB包含被DLL导出的函数名称和位置，DLL包含实际的函数和数据，应用程序使用LIB文件链接到DLL文件。在应用程序的可执行文件中，存放的不是被调用的函数代码，而是DLL中相应函数代码的地址，从而节省了内存资源。DLL和LIB文件必须随应用程序一起发行，否则应用程序会产生错误。如果不想用lib文件或者没有lib文件，可以用WIN32 API函数LoadLibrary、GetProcAddress装载。

（怎么制作静态库与动态库）liunx对编译后的文件使用ar命令可以制作静态库。类的动态链接库是利用多态性动态加载类，只能用于C++调用，有如下的几个步骤：（1）定义一个抽象类，提供纯虚函数接口。（2）具体实现类继承抽象类。（3）提供抽象类对象的创建和销毁的接口。一般可以通过g++ -shared -fPIC生成动态链接库。

## g++生成动态链接库

-shared：该选项指定生成动态连接库

-fPIC：表示编译为位置独立的代码，不用此选项的话编译后的代码是位置相关的所以动态载入时是通过代码拷贝的方式来满足不同进程的需要

## main函数之前和之后要执行哪些代码

main函数执行之前，主要就是初始化系统相关资源：

* 设置栈指针；
* 初始化静态static变量和global全局变量，即.data段的内容；
* 将未初始化部分的全局变量赋初值：数值型short，int，long等为0，bool为FALSE，指针为nullptr等等，即.bss段的内容 ；
* 全局对象初始化，在main之前调用构造函数，这是可能会执行前的一些代码；
* 将main函数的参数argc，argv等传递给main函数，然后才真正运行main函数；

main函数执行之后：

* 全局对象的析构函数会在main函数之后执行；
* 可以用 atexit 注册一个函数，它会在main 之后执行，main函数结束时会隐式地调用atexit()函数登记的函数，然后做一些自身的清理工作；atexit函数可以按照我们自己设定的顺序销毁全局变量；例如有个log类是写日志的，其他的全局类有可能调用它去写日志，所以log类必须在最后进行析构，如果没有设定析构顺序，那么函数退出有可能先析构log类，那么其他全局类就没办法写日志了。

## main函数传参：argc、argv

argc是int类型，表示运行程序的时候给main函数传递了几个参数；

argv是一个字符串数字，这个数组用来存储多个字符串，每个字符串就是我们给main函数传的一个参数。

argv[0]就是我们给main函数的第一个参数，argv[1]就是传给main的第二个参数。

## -------------------------C++线程问题-----------------------------

## C++如何解决线程安全问题（mutex、lock\_guard、unique\_lock）

线程之间的同步可以通过互斥锁、信号量进制、自旋锁、读写锁、条件变量来进行实现，在C++ 11中提供了<Mutex>、<atomic>、<condition\_variable>来实现上述功能。

1. 互斥锁：C++11中的互斥锁可以通过mutex头文件进行实现。

* **构造函数：**std::mutex不允许拷贝构造，也不允许move拷贝，最初产生的mutex对象是处于unlocked状态的。
* **lock()：**调用线程将锁住该互斥量，线程调用该函数会发生以下3种情况：

（1）如果该互斥量当前没有被锁住，则调用线程将该互斥量锁住，直到调用unlock之前，该线程一直拥有该锁。

（2）如果当前互斥量被其他线程锁住，则当前的调用线程被阻塞住。

（3）如果当前互斥量被当前调用线程锁住，则会产生死锁。

* **unlock()：**解锁，释放对互斥量的所有权。
* **try\_lock()：**尝试锁住互斥量，如果互斥量被其他线程占有，当前线程也不会被阻塞。返回true或者是false。

/\***Mutex提供了四种类型的互斥量**，第一个是std::mutex，这是最基本的mutex类；第二个是std::recursive\_mutex，这是递归mutex类，能多次锁定而不死锁，在实现原理上其实就对mutex加了一个引用计数，保证了递归函数的不死锁；第三个是std::time\_mutex，这是定时mutex类，可以锁定一定的时间，第四个是std::recursive\_timed\_mutex，这是定时递归mutex类。\*/

**为了避免在上锁和解锁之间的程序出现异常情况而导致锁无法正确释放，C++11中提供了unique\_lock和lock\_guard来保证上锁和解锁的实现，它们通过构造函数上锁，通过析构函数进行锁释放。**

**（unique\_lock和lock\_guard区别）**绝大多数情况下这两种锁是可以互相替代的，区别是unique\_lock比lock\_guard能提供更多的功能特性（但需要付出性能的一些代价），如下：

A、unique\_lock可以实现延时锁，即先生成unique\_lock对象，然后在有需要的地方调用lock函数，lock\_guard在对象创建时就自动进行lock操作了；

B、unique\_lock可以在需要的地方调用unlock操作，而lock\_guard只能在其对象生命周期结束后自动Unlock；

正是由于这两个差异特性，unique\_lock可以用于一次性锁多个锁以及用于条件变量的搭配使用，而lock\_guard做不到。

（2）信号量机制：是一个整型变量，可以对其执行 down 和 up 操作，也就是常见的P和V操作。P操作表示如果信号量大于0，执行减1操作；如果信号量等于0，进程睡眠，等待信号量大于0；V操作对信号量执行 +1 操作，唤醒睡眠的进程让其完成P操作。在C++ 11中可以通过atomic实现信号量计数来实现信号量机制。

（3）自旋锁：自旋锁是一种忙等形式的锁，会再用户态不同的询问锁是否可以获取，不会陷入到内核态中，所以更加高效。缺点是可能会对CPU资源造成浪费。但是在C++11中并没有直接提供自旋锁的实现。但是在C++11中提供了原子操作的实现，可以借助原子操作实现简单的自旋锁。

（4）条件变量：条件变量是一种同步机制，允许线程挂起，直到共享数据上的某些条件得到满足。条件变量上的基本操作有：触发条件(当条件变为 true 时)；等待条件，挂起线程直到其他线程触发条件。条件变量要和互斥量相联结，以避免出现条件竞争，C++ 11可以直接使用condition\_variable实现。

（5）读写锁：C++ 11没有提供读写锁的实现，读写锁的原理就是，可以多次读，但是写只能一次一次的写入。在C++ 11中可以借助<Mutex>和<condition\_variable>来实现读写锁，简单的思路就是写的时候上锁，长时间写可以通过条件变量进行挂起，写完再唤醒其他线程。

## C++原子操作、无锁队列的实现

原子操作：例如可以使用CAS函数（C++atomic类中的无锁函数compare\_exchange\_weak实现）实现各种无锁的数据结构解决并发问题，CAS是原⼦操作的⼀种，可⽤于在多线程编程中实现不被打断的数据交换操作，从而避免多线程同时改写某⼀数据时由于执行顺序不确定性以及中断的不可预知性产⽣的数据不一致问题。

无锁队列其实是采用了CAS函数进行push和pop操作，在做push操作时是通过do-while循环，首先执行CAS操作发现队尾结点后为空，于是就执行do-while中的CAS操作将尾节点\_tail的\_next指针赋值为newNode，然后退出do-while循环，调用第二个CAS操作将尾节点指针向后移动一位；由于CAS是一个原子操作，所以即使同时T2线程了也调用了do-while中的CAS操作，但是其判断p->\_next不为空，因为T1线程已经将尾节点向后移动了，所以其只能继续执行do，将p向后移动，重新移动到尾节点继续重新判断，直到成功为止。

Pop操作和push相似，假设T1先执行CAS操作将\_head向后移动了一位，并且删除了原先的头指针，那么当T2再执行时发现T1更新过后的\_head指针(移动了)与一开始获取的头指针p不相等了，那么就继续执行do-while循环重新获取头指针，然后重新进行CAS操作。

## C++多线程的join和detach

在一个线程中，开了另一个线程去干另一件事，使用join函数后，原始线程会等待新线程执行结束之后，再去销毁线程对象。

Detach 称为分离线程函数，使用detach()函数会让线程在后台运行，即说明主线程不会等待子线程运行结束才结束。

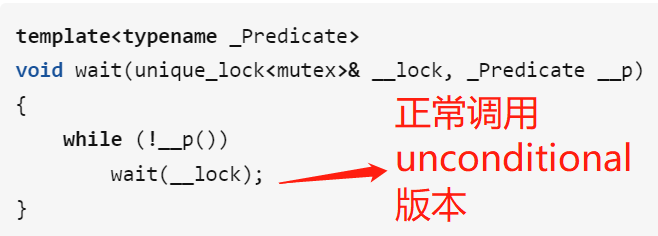
使用join要等到新线程执行完，再销毁线程对象，这样如果新线程使用了共享变量，等到新线程执行完再销毁这个线程对象，不会产生异常。如果不使用join，使用detach，那么新线程就会与原线程分离，如果原线程先执行完毕，销毁线程对象及局部变量，并且新线程有共享变量或引用之类，这样新线程可能使用的变量，就变成未定义，产生异常或不可预测的错误。

## C++条件变量condition\_variable的解释

condition\_variable是一个类，搭配互斥量mutex来用，该类主要有两个功能函数，即wait和notify，这里的notify分为notify\_once()和notify\_all()。当程序运行到wait函数的时候会先在此阻塞，然后自动unlock传入的互斥量，这样可以保证其他线程拿到这个锁能够接着执行下去，当运行到notify\_once()和notify\_all()函数的时候，就会唤醒wait函数，然后自动lock并继续下运行。

notify\_one()每次只能唤醒一个线程，那么notify\_all()函数的作用就是可以唤醒所有的线程，或者说有多个线程在wait，但是用notify\_one()去唤醒其中一个线程，那么这些线程就出现了去争夺互斥量的一个情况，那么最终没有获得锁的控制权的线程就会再次回到阻塞的状态，那么对于这些没有抢到控制权的这个过程就叫做虚假唤醒，通常可以听过一个while循环来配合虚假唤醒的问题。

C++条件变量的wait，有两个函数重载版本，一个叫unconditional无条件版本，第二个是predication版本接受一个函数指针，返回bool类型，只有当第二参数为false时才会阻塞当前线程，并且在收到其他线程的通知后只有这个参数为true时才会被解除阻塞。从函数的角度来看是这样的：



线程被唤醒后，先重新判断函数指针返回的值。如果为false，则会释放mutex并重新阻塞在wait，**该重载消除了意外唤醒的影响**。

## C++11中创建线程的5种方法（thread函数介绍）

C++中使用std::thread创建线程，调用的函数类型主要有以下五种：

* 函数指针——这是创建线程的基本形式。
* Lambda函数。
* 仿函数(函数对象)。
* 非静态成员函数。
* 静态成员函数。



## C++ future

std::async：

可以使用 std::async启动一个异步任务，会返回一个std::future对象，std::future对象中存放着最终计算的结果，它并不总会开启新的线程来执行任务，我们可以指定 std::launch::async来强制开启新线程，当需要最终结果时，调用 std::future::get() 方法即可，这个会阻塞线程直到期望值状态就绪为止。Async方法存在std::future 析构阻塞的问题。

packaged\_task：

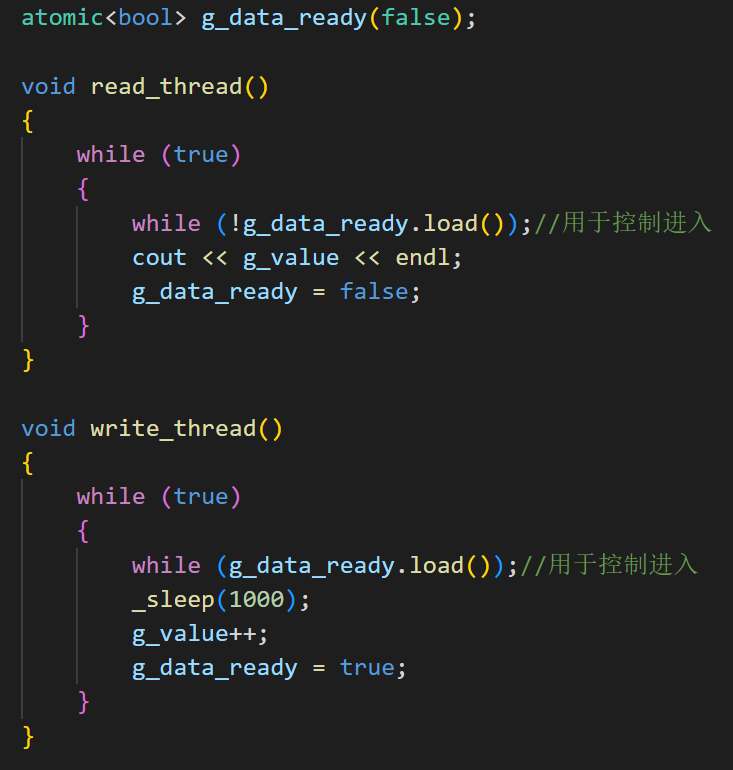
相当于是一个关联了std::future 的仿函数，packaged\_task打包一个函数以后，打包完成的packaged\_task也是一个**可调用对象**，对其调用get函数相当于全部都在主线程运行了，没有另外开线程，而且也可以将packaged\_task对象放在容器中使用。

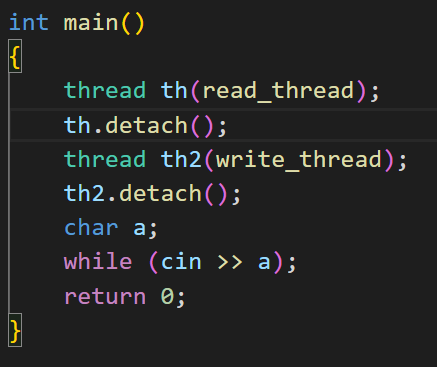
Promise：

Promise能够在某个线程中给它赋值,然后我们可以在其他线程中，把这个值取出来用，实现两个线程之间的通信；就是说我们通过promise保存一个值，在将来某个时刻我们通过把一个future绑定到这个promise上来得到这个绑定的值。

## C++原子操作实现互斥锁

开两个thread（之间是detach关系），然后不停调用对atomic对象的修改即可。





## -----------------------------指针和引用----------------------------

## 指针和引用的区别

* 指针是一个变量，存储的是一个地址，引用跟原来的变量实质上是同一个东西，是原变量的别名。
* 指针可以有多级，引用只有一级。
* 指针可以为空，引用不能为NULL且在定义时必须初始化。
* 指针在初始化后可以改变指向，而引用在初始化之后不可再改变。
* 当把指针作为参数进行传递时，也是将实参的一个拷贝传递给形参，两者指向的地址相同，但不是同一个变量，在函数中改变这个变量的指向不影响实参，而引用却可以。
* 不存在指向空值的引用，必须有具体实体；但是存在指向空值的指针。

## 什么时候用指针，什么时候用引用

* 需要返回函数内局部变量的内存的时候用指针。使用指针传参需要开辟内存，用完要记得释放指针，不然会内存泄漏。而返回局部变量的引用是没有意义的。
* 对栈空间大小比较敏感（比如递归）的时候使用引用。使用引用传递不需要创建临时变量，开销要更小。
* 类对象作为参数传递的时候使用引用，这是C++类对象传递的标准方式。

## 指针常量和常量指针的差别

指针常量是一个指针，读成常量的指针，**指向一个只读变量**，也就是后面所指明的int const 和 const int，都是一个常量，//可以写作int const \*p或const int \*p。

常量指针是一个**不能给改变指向的指针**。指针是个常量，必须初始化，一旦初始化完成，它的值（也就是存放在指针中的地址）就不能在改变了，即不能中途改变指向。

## C++ this指针

This指针指向对象的首地址，一个对象的this指针并不是对象本身的一部分，不会影响 sizeof(对象)的结果。this作用域是在类内部，当在类的非静态成员函数中访问类的非静态成员的时候（全局函数，静态函数中不能使用this指针），编译器会自动将对象本身的地址作为一个隐含参数传递给函数。也就是说，即使你没有写上this指针，

编译器在编译的时候也是加上this的，它作为非静态成员函数的隐含形参，对各成员的访问均通过this进行。

## 函数指针

c++函数其实和变量一样也有一个地址，函数指针指向的是特殊的数据类型，函数的类型是由其返回的数据类型和其参数列表共同决定的，而函数的名称则不是其类型的一部分，所以我们可以定义一个指针指向该函数，然后传入相应的参数去调用这个函数指针就可以了。

## 野指针、悬空指针

C++野指针和悬空指针

指向非法的内存地址指针叫作野指针，也叫悬挂指针，意为无法正常使用的指针。野指针通常出现有两种情况（1）使用未初始化的指针（野）。（2）指针所指向的对象已经消亡（悬空）。

（避免野指针）（1）C++引入了引用机制，如果使用引用可以达到编程目的，就可以不必使用指针。因为引用在定义的时候，必须初始化，所以可以避免野指针的出现。（2）如果一定要使用指针，那么需要在定义指针变量的同时对它进行初始化操作。定义时将其置位NULL或者指向一个有名变量，**或者考虑使用智能指针**。 （3）对指针进行delete操作后，将其设置为nullptr。

## C++为什么会引入nullptr

NULL在C++中就是0，这是因为在C++中void\* 类型是不允许隐式转换成其他类型的，所以之前C++中用0来代表空指针，但是在重载整形（函数重载）的情况下，会出现二义性问题，编译器会报错。所以，C++11加入了nullptr，可以保证在任何情况下都代表空指针。

## C++智能指针

(1) shared\_ptr

实现原理：采用引用计数器的方法，允许多个智能指针指向同一个对象，每当多一个指针指向该对象时，指向该对象的所有智能指针内部的引用计数加1，每当减少一个智能指针指向对象时，引用计数会减1，当计数为0的时候会自动的释放动态分配的资源。

(2) unique\_ptr

unique\_ptr采用的是独享所有权语义，一个非空的unique\_ptr总是拥有它所指向的资源。转移一个unique\_ptr将会把所有权全部从源指针转移给目标指针，源指针被置空；所以unique\_ptr不支持普通的拷贝和赋值操作。

(3) weak\_ptr

weak\_ptr：弱引用。 引用计数有一个问题就是互相引用形成环（环形引用），这样两个指针指向的内存都无法释放。需要使用weak\_ptr打破环形引用。weak\_ptr是一个弱引用，它是为了配合shared\_ptr而引入的一种智能指针，它指向一个由shared\_ptr管理的对象而不影响所指对象的生命周期，也就是说，它只引用，不计数。如果一块内存被shared\_ptr和weak\_ptr同时引用，当所有shared\_ptr析构了之后，不管还有没有weak\_ptr引用该内存，内存也会被释放。所以weak\_ptr不保证它指向的内存一定是有效的，在使用之前使用函数lock()检查weak\_ptr是否为空指针。

(4) auto\_ptr

主要是为了解决“有异常抛出时发生内存泄漏”的问题 。因为发生异常而无法正常释放内存。

## shared\_ptr的实现原理以及问题（不线程安全）

**实现原理：**采用引用计数器的方法，允许多个智能指针指向同一个对象，每当多一个指针指向该对象时，指向该对象的所有智能指针内部的引用计数加1，每当减少一个智能指针指向对象时，引用计数会减1，当计数为0的时候会自动的释放动态分配的资源。

（1）问题1：会出现互相循环引用的问题，例如在二叉树中出现指向父亲结点的指针如果用shared\_ptr的话由于互相指向，当退出定义域也不会释放，所以要用weak\_ptr，它是和shared-ptr配合使用的，weak\_ptr没有引用计数的机制，当退出作用域对象就会自动失效。

（2）问题2：shared\_ptr不是线程安全的

线程安全的定义：A、确保在多条线程访问的时候，我们的程序还能按照我们预期的行为去执行。B、调用代码无须额外的同步或者其他协调动作。

Shared\_ptr不是线程安全的。首先看shared\_ptr本身的规则，如果定义一个shared\_ptr对象x，在定义一个shared\_ptr对象y，y如果用x初始化，这个时候会发生两个步骤，首先两个对象内部的指针会同时指向目标内存，然后计数器变为2。在多线程的时候，这样的规则就会出现问题，**首先如果假设这里有一个线程共享的指针a，然后线程1有一个临时指针b，线程2临时指针c，假如线程1中的这个临时指针b借助a进行了初始化，相当于读操作，如果计数器更改操作还没有完成，就进入到了线程2，而在线程2中，对共享指针a要进行一个写操作，令a等于线程2中的c，这个时候，原本的a对象的计数器会变为0，进行一个析构，那么这个时候就相当于线程1中的指针变成的悬空指针（指向的内存被释放了）。综上，shared\_ptr不是线程安全的。**

Unique\_ptr也不是线程安全的。原因在于unique\_ptr可以接收裸指针（内置类型指针）作为成员函数参数。可能会导致两个线程中的临时unique\_ptr对象指向同一块内存，然后造成悬空指针问题。

## shared\_ptr如何赋值给另一个（或者函数返回临时变量shared\_ptr、shared\_ptr会出现的问题）

当我们在类中采用share\_ptr<this>传递时，这样会造成2个非共享的share\_ptr指向一个对象，最后造成2次析构该对象，从而出现了double free的问题。

enable\_shared\_from\_this的一种实现方法是，其内部有一个weak\_ptr类型的成员变量，当shared\_ptr构造的时候，如果其模板类型继承了enable\_shared\_from\_this，则对那个weak\_ptr类型的成员变量进行初始化操作，这样将来调用shared\_from\_this函数的时候，就能够通过weak\_ptr构造出对应的shared\_ptr。

## ----------------------------多态、、类-----------------------------

## C++三大特性及其目的

封装：就是将抽象得到的数据和功能相结合，形成一个整体，也就是将数据与操作数据的源代码进行结合，形成”类”，其中数据和函数都是类的成员。**封装的目的**是增强安全性和简化编程，使用者不必了解具体的实现细节，而只是要通过外部接口，特定的访问权限来使用类的成员；因此封装可以隐藏实现细节，使得代码模块化。

继承：C++通过类派生机制来支持继承。被继承的类型称为基类或超类，新产生的类为派生类或子类。保持已有类的特性而构造新类的过程叫做继承，在已有类的基础上新增自己的特性而产生新类的过程称为派生。**继承和派生的目的**是保持已有类的特性并构造新类；能够实现代码重用及代码扩充。

多态：可以概括为“一个接口，多种方法”，程序在运行时才决定调用的函数。C++多态性是通过虚函数来实现的，虚函数允许子类重新定义成员函数，当子类重新定义父类的做法称为函数的重写。而重载则是允许有多个同名的函数，而这些函数的参数列表不同，允许参数个数不同，参数类型不同，或者两者都不同，而重写要求参数类型及个数都相同。**多态的目的**则是为了接口重用，就是说不论传递过来的究竟是哪个类的对象，函数都能够通过同一个接口调用到适应各自对象的实现方法。

## 编译时多态和运行时多态

对模板参数而言，多态是通过模板实例化和函数重载解析实现的。**以不同的模板参数实例化导致调用不同的函数**，这就是所谓的**编译期多态**。相比较于运行期多态，实现编译期多态的类之间并不需要成为一个继承体系，它们之间可以没有什么关系，但约束是它们都有相同的隐式接口，在编译期间，编译器推断出模板参数，因此确定调用的是哪个具体类型的接口。**不同的推断结果调用不同的函数。**

**运行期多态**的实现依赖于虚函数机制。当某个类声明了虚函数时，编译器将为该类对象安插一个虚函数表指针，并为该类设置一张唯一的虚函数表，虚函数表中存放的是该类虚函数地址。运行期间通过虚函数表指针与虚函数表去确定该类虚函数的真正实现。

## 为什么要用虚函数？

为了把一个复杂的事情进行分层，把一个复杂的东西变成简单东西。

虚函数主要是实现代码复用，实现多态就是将接口与实现进行分离。

## 构造函数和析构函数可以是虚函数吗？

构造函数不能：因为虚函数的调用是通过类的实例化对象中的虚函数表指针vptr实现的，该指针存放在类对象的内部空间中，如果把构造函数设置成虚函数，那么构造的时候就要找vptr指针，而此时vptr还没有初始化

析构函数能且经常是虚函数：只有在基类析构函数定义为虚函数时，调用操作符 delete 销毁指向对象的基类指针时，才能准确调用派生类的析构函数（从该级向上按序调用虚函数），才能准确销毁数据。

## 构造函数、析构函数、虚函数可否声明为内联函数

前两个是无意义的：因为编辑器并不会把它们设为内敛，这两个操作编辑器会增加申请/释放内存，构造/析构对象等操作，这样下来其实就不是很精简了，而内联函数主要是对短小精简的代码进行内联展开提高函数的执行效率，这样就不是很高效了。

虚函数可以是内联函数，但是当虚函数表现多态性的时候不能内联。内联是在发生在编译期间，编译器会自主选择内联，而虚函数的多态性在运行期，编译器无法知道运行期调用哪个代码，因此虚函数表现为多态性时（运行期）不可以内联。

## C++虚函数表构建过程

1. 先将基类中的虚表内容拷贝一份到派生类虚表中。
2. 如果派生类重写了基类中某个虚函数，用派生类自己的虚函数替换虚表中基类的虚函数。
3. 派生类自己新增加的虚函数按其在派生类中的声明次序增加到派生类虚表的最后。

## C++类的8个默认函数

A(); // 默认构造函数;（默认有）

A(const A&); // 默认拷贝构造函数（默认有）

~A(); // 默认析构函数（默认有）

A& operator = (const A&); // 默认重载赋值运算符函数（默认有）

A\* operator & (); // 默认重载取址运算符函数

const A\* operator & () const; // 默认重载取址运算符const函数

A(A&&); // **默认移动构造函数**

A& operator = (const A &&); // **默认重载移动赋值操作符**

**移动构造成员函数**不会分配任何新的资源，它会将右值的资源占用，然后将右值的状态设置为默认构造后的状态。这样操作之后，左值的对象就获得了临时对象的资源的“控制权”，同时临时对象也可以安全析构而不会导致数据失效。同理的还有移动赋值，它是对操作符=的重载，接受一个自身类的实例的右值。

## C++什么时候调用移动构造函数

1. 确认某个资源不再被需要的时候，例如向容器里添加外部的内对象的时候，如果确认外部对象不再被使用，可以通过move转移类对象资源避免拷贝的大规模开销。
2. 通过临时变量来初始化对象的时候会调用移动构造函数，例如通过函数返回临时对象来初始化对象时，会调用移动构造，如果没有定义移动构造函数会使用拷贝构造。

## C++什么时候会调用拷贝构造函数

* 用类的一个实例化对象去初始化另一个对象的时候
* 函数的参数是类的对象时（非引用传递）
* 函数的返回值是函数体内局部对象的类的对象时，由于返回方式是值传递，所以会在返回值的地方调用拷贝构造函数。
* 用花括号列表初始化一个数组中的元素或以个聚合类中的成员时（无移动构造）。

## C++拷贝构造函数为什么需要引用，不引用可以吗

如果拷贝构造函数中的参数不是一个引用，那么就相当于采用了传值的方式，而传值的方式会调用该类的拷贝构造函数，从而造成无穷递归地调用拷贝构造函数。因此拷贝构造函数的参数必须是一个引用。

## C++什么情况下必须用初始化列表

（1）当初始化一个引用成员时；（2）当初始化一个常量成员时；（3）当调用一个基类的构造函数，而它拥有一组参数时；（4）当调用一个成员类的构造函数，而它拥有一组参数时；

## C++友元问题

1. 在定义一个类的时候，可以把一些函数（包括全局函数和其他类的成员函数）声明为“友元”，这样那些函数就成为该类的友元函数，在友元函数内部就可以访问该类对象的私有成员了。
2. 友元关系不能被继承。
3. 友元关系是单向的，不具有交换性。若类B是类A的友元，类A不一定是类B的友元，要看在类中是否有相应的声明。
4. 友元关系不具有传递性。若类B是类A的友元，类C是B的友元，类C不一定是类A的友元，同样要看中是否有相应的申明

## 结构体内存对齐的作用

指定对齐值：#pragma pack (value)

当计算机访问变量内存的时候，理论上我们可以在任何位置逐字访问，但是计算机一般是按照2，4，8这样的顺序去访问，这样假如我们把应该int变量放在一个奇数位置，访问这个变量就需要两个访问周期，这样会影响效率。

## 类的继承及初始化顺序：

第一步执行：虚拟基类的构造函数（多个虚拟基类则按照继承的顺序执行构造函数）。

第二步执行：基类的构造函数（多个普通基类也按照继承的顺序执行构造函数）。

第三步执行：类类型的成员对象的构造函数（按照初始化顺序）。

第四部执行：派生类自己的构造函数。

## C++多继承二义性问题（棱形继承）

为了解决从不同途径继承来的同名的数据成员在内存中有不同的拷贝造成数据不一致问题，将共同基类设置为虚基类。这时从不同的路径继承过来的同名数据成员在内存中就只有一个拷贝，同一个函数名也只有一个映射。这样不仅就解决了二义性问题，也节省了内存，避免了数据不一致的问题，采用::可以访问父类中不同的成员函数。

## 如何不让用户去实例化一个类对象（动态申请、静态申请）？

修改为只能动态申请：将构造函数和析构函数设置为protected或private

修改为只能静态申请：将new和delete操作符重载为private属性

## C++静态成员和普通成员的区别是什么（static）

（1）生命周期：静态成员变量从类被加载开始到类被卸载，一直存在；普通成员变量只有在类创建对象后才开始存在，对象结束，它的生命期结束；

（2）共享方式：静态成员变量是全类共享；普通成员变量是每个对象单独享用的；

（3）定义位置：普通成员变量存储在栈或堆中，而静态成员变量存储在静态全局区；

（4）初始化位置：普通成员变量在类中初始化；静态成员变量在类外初始化；

（5）默认实参：常量静态成员恶可以在类中初始化，并且可以使用常量静态成员变量作为默认实参。

## -----------------------------C++ 11部分--------------------------

## c++11的新标准

* nullptr替代 NULL
* 引入了 auto 和 decltype 这两个关键字实现了类型推导
* 基于范围的 for 循环for(auto& i : res){}
* 类和结构体的中初始化列表
* Lambda 表达式（匿名函数）
* std::forward\_list（单向链表）unordered\_map
* 右值引用和move语义

## 左值和右值是什么，move是干什么的

左值指既能够出现在等号左边，也能出现在等号右边的变量，表示的的是可寻址的变量，有持久性；

右值则是只能出现在等号右边的变量，一般是不可寻址的常量，或在表达式求值过程中创建的无名临时对象，短暂性的。

**左值和右值主要的区别之一是左值可以被修改，而右值不能。**

**move 就是做类型转换的**，把指代对象的左值表达式变成将亡值。

## 右值引用的作用（移动语义、完美转发）

右值引用和左值引用不同，通常他是对一个常量的引用，一般会有两个作用分别是：移动语义和完美转发。

移动语义：主要是在大量数据进行复制的时候，直接转让出去动态申请内容的所有权，不用进行大量的数据移动，节省空间且高效，主要通过move函数和移动赋值运算符来实现。

完美转发：完美转发是根据参数的类型进行传递，参数类型不会发生转换，如果参数类型是右值引用就转发右值引用，如果是其他的类型就转发其他类型。

## C++move函数的作用及原理

**move 就是做类型转换的**，把指代对象的左值表达式变成将亡值。std::move进行了一种人工的转化，在转化之后，其返回了一个右值。通过使用这个右值，我们可以调用任何重载了使用右值作为形参的版本的函数。这样的函数一般利用了“避免复制，而是转移资源控制权”这一理念，从而高效的转移被分配的内存，避免了多次重分配内存带来的性能降低。其代价是，我们必须有意识地遵守一个基本规定，“右值是不可使用的”。在使用了std::move转化出的右值之后，对于原始的对象我们不要对其进行任何的访问、修改，静待它被析构即可。

**Move函数调用static\_cast把输入的参数转换为右值引用**，move函数定义了一个函数模板，函数参数T&&是一个指向模板类型参数的右值引用，通过引用折叠，此参数可以与任何类型的实参匹配，引用折叠就是C++里面的一个类型推导规则，即左值与通用引用放在一起推导出来的 T 仍为左值，而右值与通用引用放在一起推导出来的 T 仍然为右值，通过引用折叠保证了模板可以传递任意实参，且保持类型不变。然后move通过static\_cast<>进行强制类型转换返回T&&右值引用，而static\_cast<T>之所以能使用类型转换，是通过remove\_refrence<T>::type模板偏特化移除T&&，T&的引用，获取具体类型T，然后定义右值转化类型。

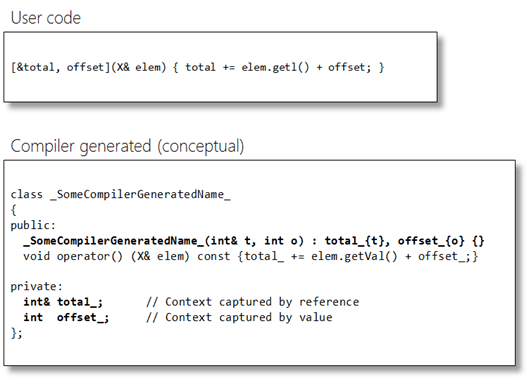
## Lambda表达式

C++ lambda表达式是C++11引入的一个新特性，可以方便快捷地创建一个“可调用对象”，C++11中有三种可调用对象：函数指针，重载()的类以及lambda表达式。



Lambda表达式的原理：

其实是编译器为我们了创建了一个类，这个类重载了()，让我们可以像调用函数一样使用。而对于捕获变量的lambda表达式来说，编译器在创建类的时候，通过构造函数的形式保存了需要捕获的变量。



## -----------------------------C++内存部分-------------------------

## C++有几种new

在C++中，new有三种典型的使用方法：plain new，nothrow new和placement new。

（1）plain new，就是普通的new，就是我们常用的new，也就是operator new，在空间分配失败的情况下，抛出异常std::bad\_alloc而不是返回NULL。

（2）nothrow new，nothrow new在空间分配失败的情况下是不抛出异常，而是返回NULL。

（3）placement new，这种new允许在一块已经分配成功的内存上重新构造对象或对象数组。placement new不用担心内存分配失败，因为它根本不分配内存，它做的唯一一件事情就是调用对象的构造函数。placement new构造起来的对象数组，要显式的调用他们的析构函数来销毁（析构函数并不释放对象的内存），千万不要使用delete，这是因为placement new构造起来的对象或数组大小并不一定等于原来分配的内存大小，使用delete会造成内存泄漏或者之后释放内存时出现运行时错误。**删除时要直接调用重新构造对象的析构函数，但是该过程并不会释放内存，后续继续调用原分配对象的delete即可。**

## New和delete的过程

1、 new简单类型直接调用operator new分配内存；对于复杂结构，先调用operator new分配内存，然后在分配的内存上调用构造函数；new[]先调用operator new[]分配内存，然后在p的前四个字节写入数组大小n，然后调用n次构造函数，此时new[]会额外一个数字表示存储数组大小；

① new表达式调用一个名为operator new(operator new[])函数，分配一块足够大的、原始的、未命名的内存空间；

② 编译器运行相应的构造函数以构造这些对象，并为其传入初始值；

③ 对象被分配了空间并构造完成，返回一个指向该对象的指针。

2、 delete简单数据类型默认只是调用free函数；复杂数据类型先调用析构函数再调用operator delete；如果假设指针p指向new[]分配的内存。因为保存了4个字节的数组长度，实际分配的内存地址为[p-4]，系统记录的也是这个地址。delete[]实际释放的就是p-4指向的内存。而delete会直接释放p指向的内存，这个内存根本没有被系统记录，所以会崩溃。

## C++释放的空间会立刻归还操作系统吗

不是的，被free回收的内存会首先被ptmalloc使用双链表保存起来，当用户下一次申请内存的时候，会尝试从这些内存中寻找合适的返回。这样就避免了频繁的系统调用，占用过多的系统资源。同时ptmalloc也会尝试对小块内存进行合并，避免过多的内存碎片。

## Delete[]如何知道释放空间大小

需要在new []一个对象数组时，需要保存数组的维度，C++的做法是在分配数组空间时多分配了4个字节的大小，专门保存数组的大小，在 delete []时就可以取出这个保存的数，就知道了需要调用析构函数多少次了。

## New和malloc的区别

New会报错，使用nothrow\_new后会返回null

Malloc会返回空指针

1、 new/delete是C++关键字，需要编译器支持。malloc/free是库函数，需要头文件支持；

2、 使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算。而malloc则需要显式地指出所需内存的尺寸。

3、 new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换。而malloc内存分配成功则是返回void \* ，需要通过强制类型转换，将void\*指针转换成我们需要的类型。

4、 new内存分配失败时，会抛出bad\_alloc异常。malloc分配内存失败时返回NULL。

5、 new会先调用operator new函数，申请足够的内存（通常底层使用malloc实现）。然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete先调用析构函数，然后调用operator delete函数释放内存（通常底层使用free实现）。malloc/free是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。

## Malloc和free的实现原理

（1）操作系统会有一个记录空闲地址的链表（ptmalloc双向链表），malloc的申请过程首先会遍历该链表，找到第一个大于申请空间的地址。

（2）如果系统没有空闲空间了，也就是链表上找不到合适大小的内存，则会使用系统调用brk或mmap的方式来申请内存，brk是将数据段的最高指针往高地址方向推，mmap是在进程的虚拟内存空间（文件映射区域）找到一块空闲的虚拟内存，这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。

malloc小于128k的内存，使用brk分配内存，将\_edata往高地址推；malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配。

## 内存溢出与内存泄露

内存溢出：表示所申请的内存空间超过了系统所提供的内存，导致无法申请成功，这就是内存溢出。

内存泄露：表示运行过程中所申请的临时变量在使用结束后没有进行回收，这样既无法使用也无法释放，就造成了内存泄露。

**常见的内存泄漏的原因：**（1）指针重新赋值（2）错误的内存释放（3）返回值的不正确处理。

检查泄露的方法：使用CRT库来检测，内存分配的时候需要通过CRT在运行时实现，要在分配内存和释放内存的时候做好记录，程序结束时对比分配内存和释放内存的记录就能确定是否发生了内存泄露。

## -------------------------强制转换、关键字等---------------------

## C++强制转换（reinterpret\_cast等）

Reinterpretcast：

指针、引用、函数指针、成员指针的强制转换，按照位数从小到大开始转换

Constcast：

* 常量指针被转化成非常量的指针，并且仍然指向原来的对象；
* 常量引用被转换成非常量的引用，并且仍然指向原来的对象；

**Staticcast**：基础+上行安全，下行不安全（无动态类型检查）+空指针类型转为目标指针类型

Dynamiccast：下行转换安全，加入了动态类型检查，进行动态类型检查时，如果转换失败，对指针将会返回一个NULL，对引用将会抛出一个异常，这就是安全性所在。

## #define和函数的区别

（1）宏在编译时完成替换，之后被替换的文本参与编译，相当于直接插入了代码，运行时不存在函数调用，执行起来更快；函数调用在运行时需要跳转到具体调用函数。

（2）宏定义属于在结构中插入代码，没有返回值；函数调用具有返回值。

（3）宏定义参数没有类型，不进行类型检查；函数参数具有类型，需要检查类型。

（4）宏定义不要在最后加分号。

## #define和typedef的区别

（1）宏主要用于定义常量及书写复杂的内容；typedef主要用于定义类型别名。

（2）宏替换发生在编译阶段之前，属于文本插入替换；typedef是编译的一部分。

（3）宏不检查类型；typedef会检查数据类型。

（4）宏不是语句，不在在最后加分号；typedef是语句，要加分号标识结束。

## #define和inline的区别

在使用时，宏只做简单字符串替换（编译前）。而内联函数可以进行参数类型检查（编译时），且具有返回值。

内联函数在编译时直接将函数代码嵌入到目标代码中，省去函数调用的开销来提高执行效率，并且进行参数类型检查，具有返回值，可以实现重载。

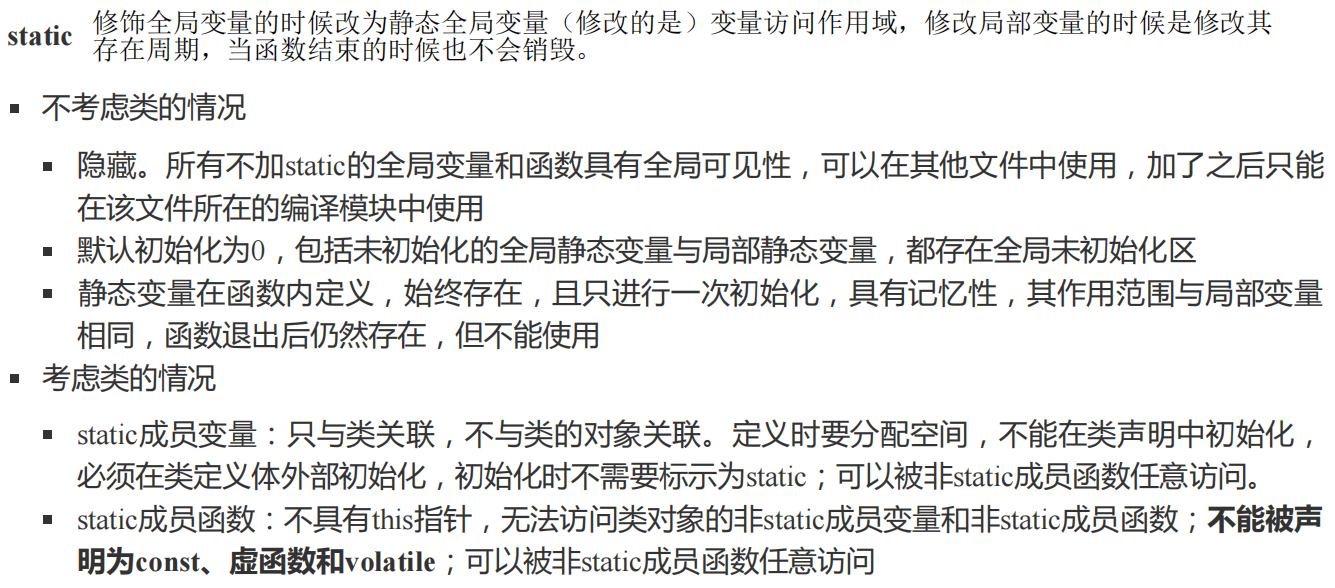
宏定义时要注意书写（参数要括起来）否则容易出现歧义，内联函数不会产生歧义。

内联函数有类型检测、语法判断等功能，而宏定义没有。

## C++struct和class的区别

1. 默认访问权限：struct作为数据结构的实现体，它默认的数据访问控制是public的，而class作为对象的实现体，它默认的成员变量访问控制是private的。
2. “class”这个关键字还用于定义模板参数，就像“typename”。但关建字“struct”不用于定义模板参数。
3. class和struct在使用大括号{ }上的区别：A、class和struct如果定义了构造函数的话，都不能用大括号进行初始化。B、如果没有定义构造函数，且所有成员变量全是public的话，class可以用大括号初始化。

## C++中const和static的作用





## C++静态成员和普通成员的区别是什么（static）

（1）生命周期：静态成员变量从类被加载开始到类被卸载，一直存在；普通成员变量只有在类创建对象后才开始存在，对象结束，它的生命期结束；

（2）共享方式：静态成员变量是全类共享；普通成员变量是每个对象单独享用的；

（3）定义位置：普通成员变量存储在栈或堆中，而静态成员变量存储在静态全局区；

（4）初始化位置：普通成员变量在类中初始化；静态成员变量在类外初始化；

（5）默认实参：常量静态成员恶可以在类中初始化，并且可以使用常量静态成员变量作为默认实参。

## C++中volatile、mutable、explicit用法

1. volatile表示该变量可能会被某些编译器未知的因素所修改，比如多线程任务中的其他线程，遇到这个变量时，操作系统就不会在对其进行优化，从而提供对特殊地址的稳定访问。

当读取该声明的变量时，操作系统每次都是从它所在的内存中进行读取，而不是读取寄存器中的值。

因此多线程任务中的共享变量需要定义为volatile。

（2）mutable主要是突破const的限制，其修饰的变量即使在const函数中也能进行修改。

（3）explicit关键字用来修饰类的构造函数，被修饰的构造函数的类，不能发生相应的隐式类型转换，只能以显示的方式进行类型转换。

## C++的sizeof以及数组传递

数组传递方法：

1. 指针传递过去（失去数组特性无法输出大小，但是支持随机访问）
2. 支持随机访问

Sizeof：输出的是变量或者数据类型的大小，分为指针/数组名

数组传回的方法：定义一个指针指向数组的首地址。

# **STL总结**

## STL的组件：

（1）容器（Containers）：从实现的角度来看，STL容器是一种class template

（2）分配器(Allocators)：负责空间配置与管理，分配器是一个实现了动态空间配置、空间管理、空间释放的class template。

（3）算法(Algorithms)：STL算法是一种function template

（4）迭代器(Iterators)：容器和算法之间的“泛型指针”，所有STL容器都附带有自己的专属迭代器。

（5）适配器（Adapters）：一种修饰容器、仿函数或迭代器接口的东西。例如STL提供的queue和stack，虽然看似容器，但其实只能算是一种容器适配器，因为底部完全借助deque。改变functor接口者，称为function adapter，改变container接口称为container adapter。

（6）仿函数（Functors）：行为类似函数，可作为算法的某种策略，从实现的角度来看，仿函数是一种重载了operator()的class或者class template，一般的函数指针可视为狭义的仿函数。

## 迭代器及iterator Traits（萃取机）作用和设计

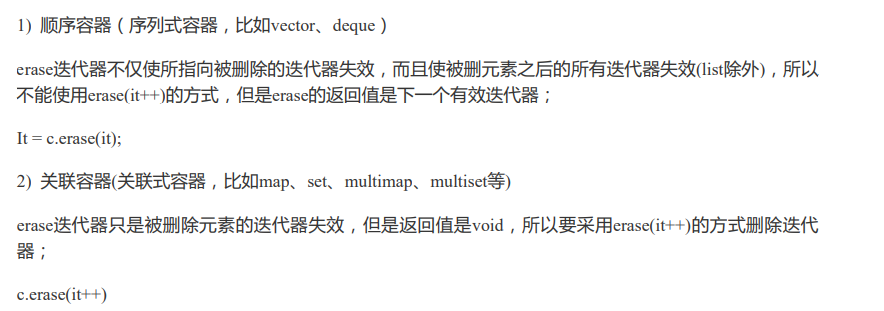
迭代器是算法和容器之间的桥梁，要实现算法，迭代器必须有能力回答算法的提问，例如C++中的sort（）函数可以对数组，也可以对容器对象排序，那排序的时候肯定就需要知道，数组或者容器的下一个对象怎么访问，交换规则等。迭代器类中包含了五种答案便于算法去提取信息，使用typedef定义：（1）iterator\_category迭代器类型（单向、双向、随机访问、输入、输出），有些迭代器只能做++操作，而有些迭代器甚至可以跳着走，例如直接跳到三个对象身位之后。（List的迭代器类型的标签是bidirectional\_iterator\_tag，标识了双向迭代）（2）Difference\_type回答迭代器之间距离的表示方式，int或者其他。（3）Value\_type回答了迭代器指向的数据类型。（4）reference获取引用。（5）pointer获取迭代器指针。

在STL中，指针也是一种迭代器，一种退化的迭代器。指针本身不能像结构体一样包含多种信息，因此为了区别C++中的指针和定义的迭代器类，STL中出现了萃取机（traits），萃取的过程其实是借助模板偏特化的决定的，当给定指针类型，给定一个特化规则，正常的迭代器类型给定一个规则。

## 迭代器前++和后++的区别

前置返回一个引用，后置返回一个对象，这样来说前置不会产生临时对象，后置必须产生临时对象，临时对象会导致效率降低，因此我们尽量使用前++操作避免产生临时对象。

## 容器内部删除一个元素



## STL的两级空间配置器

（为什么需要两级空间配置器）频繁的在堆开辟释放内存，则就会在堆上造成很多外部碎片，浪费了内存空间； 每次都要进行调用malloc、free函数等操作，使空间就会增加一些附加信息，降低了空间利用率；随着外部碎片增多，内存分配器在找不到合适内存情况下需要合并空闲块，浪费了时间，大大降低了效率。于是就设置了两级空间配置器，当开辟内存<=128bytes时，即视为开辟小块内存，调用二级空间配置器，当开辟空间大于128字节的时转去一级配置器。

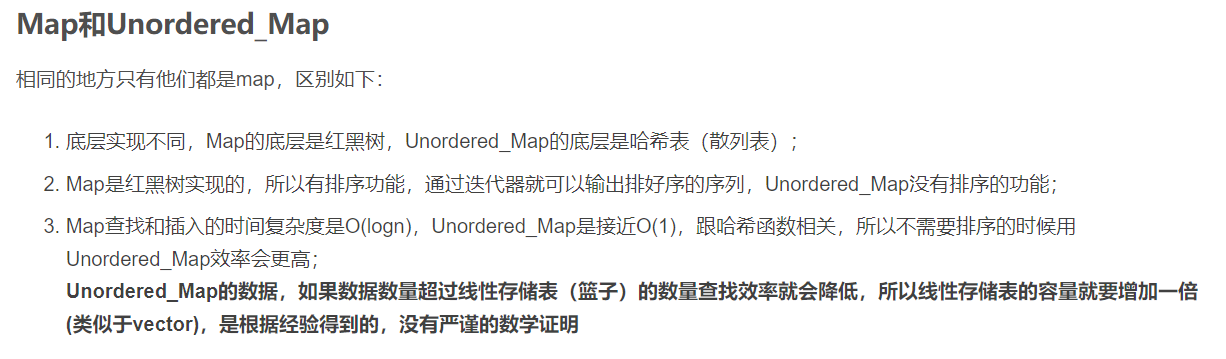
（一级空间配置器）一级空间配置器是以malloc()，free()，realloc()等C函数执行实际的内存配置 。大致过程是：（1）直接allocate分配内存，其实就是malloc来分配内存，成功则直接返回，失败就调用处理函数 （2）如果用户自定义了内存分配失败的处理函数就调用，没有的话就返回异常 （3）如果自定义了处理函数就进行处理，完事再继续分配试试

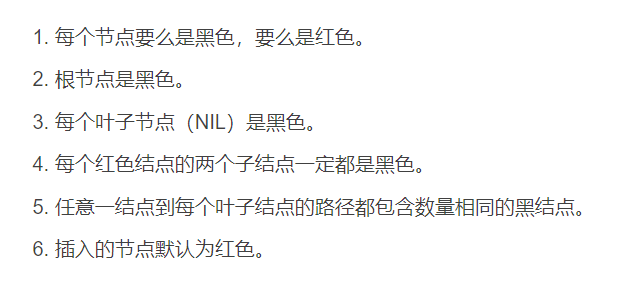
（二级空间配置器）维护16条链表，分别是0-15号链表，最小8字节，以8字节逐渐递增，最大128字节，你传入一个字节参数，表示你需要多大的内存，会自动帮你校对到第几号链表（如需要13bytes空间，我们会给它分配 16bytes大小），在找到第n个链表后查看链表是否为空，如果不为空直接从对应的free\_list中拔出，将已经拨出的指针向后移动一位。如果需要的内存块大小的链表没有了，这个时候二级空间配置器会使用malloc从堆上申请内存，会申请所需多一倍的空间块挂在到内存池中。如果malloc没有申请成功，此时程序的堆上没有足够的内存了，这个时候二级空间适配器回去大一些的内存块链表中查找所需内存，如果还是没有，最终会调用一级空间适配器。

（二级空间适配器带来的问题）（1）因为自由链表的管理问题，它会把我们需求的内存块自动提升为8的倍数，这时若你需要1个字节，它 会给你8个字节，即浪费了7个字节，所以它又引入了内部碎片的问题，若相似情况出现很多次，就会造成很多内部碎片；（2）.二级空间配置器是在堆上申请大块的狭义内存池，然后用自由链表管理，供现在使用，在程序执行过程中，它将申请的内存一块一块都挂在自由链表上，即不会还给操作系统，并且它的实现中所有成员全是静态的，所以它申请的所有内存只有在进程结束才会释放内存，还给操作系统，由此带来的问题有： 1.即我不断的开辟小块内存，最后整个堆上的空间都被挂在自由链表上，若我想开辟大块内存就会失败；2.若自由链表上挂很多内存块没有被使用，当前进程又占着内存不释放，这时别的进程在堆上申请不到空间，也不可以使用当前进程的空闲内存。

渐进式哈希

## map和unorder的原理（为什么用红黑树？）





Map原理为什么用红黑树不用其他的呢？

因为红黑树是一个非严格自平衡二叉树，他的综合性能是最佳的，红黑树和avl树相比，AVL树高度自平衡，每次插入和删除会导致失衡，会有额外的开销去调整平衡，综合下来算法开销要比红黑树大。

Sort原理：sort是集插入排序，快排，堆排序为一体的排序函数。当数据量小于16时，采用插入排序，当递归深度太深时，采用堆排序去操作。

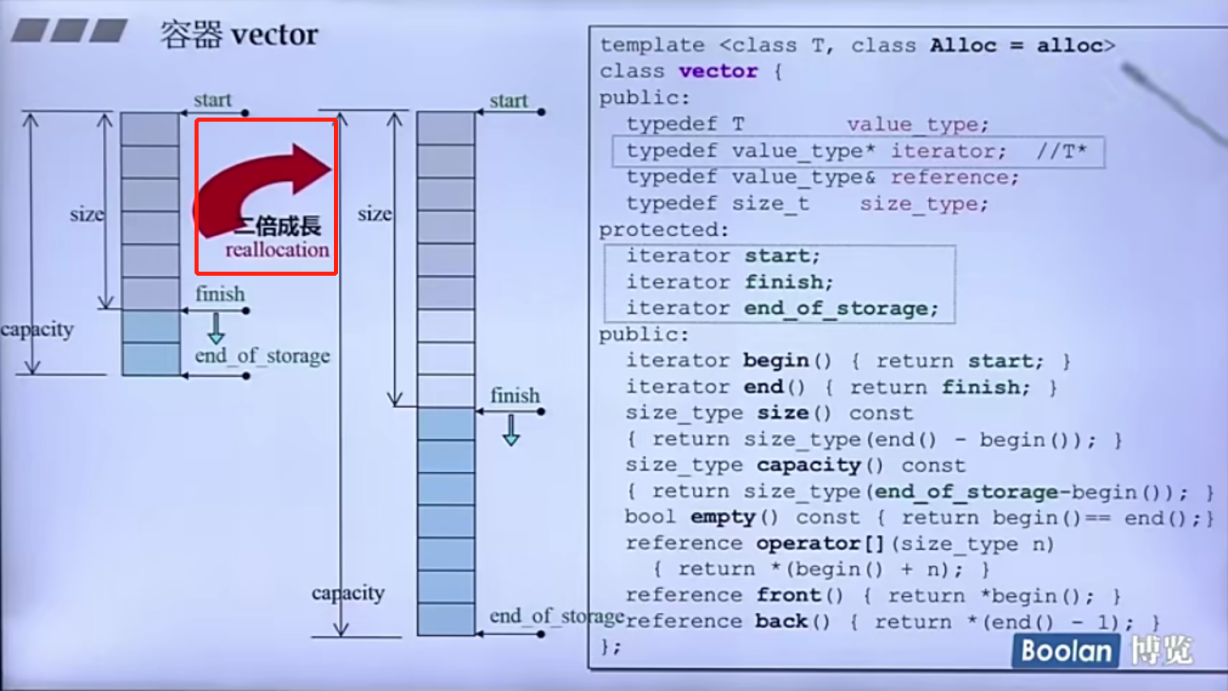
## Vector多线程动态扩充会出现的问题

可能会出现迭代器失效，程序奔溃的问题，当某个线程的vector进行push\_back的时候，如果发生动态扩容，则会导致原先的迭代器失效，访问失效的迭代器则会出现段错误。

## STL---------Vector

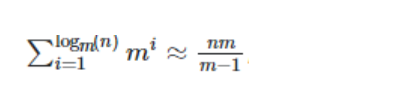
Vector实际上是一个支持动态扩容的数组，当push\_back放入元素的时候首先会判断是否有空闲位置，如果有则直接放入，如果没有空闲位置则以两倍的方式进行动态扩容，Vector模板类实际上主要是定义了三根指针，定义了起始位置start，当前数组的结束位置finish以及vector的真正的容器结尾end\_of\_storage，finish和start的差值就是vector的size。Vector是连续空间的容器，因此一定要重载[].

(扩容机制)vector如果空间大小不够，会采取两倍扩容的机制，前半段用来放置原数据（会对原数据做一个拷贝，类对象的话就是拷贝构造），后半段用来放置新数据。



C++ vector中的push\_back的时间复杂度是多少

假定有 n 个元素,倍增因子为 m。那么完成这 n 个元素往一个 vector 中的push\_back操作，需要重新分配内存的次数大约为 logmn，第 i 次重新分配将会导致复制 m^i (也就是当前的vector.size() 大小)个旧空间中元素，因此 n 次 push\_back操作所花费的总时间约为 n\*m/(m - 1):



很明显这是一个等比数列.那么 n 个元素，n 次操作，每一次操作需要花费时间为 m / (m - 1)，这是一个常量.

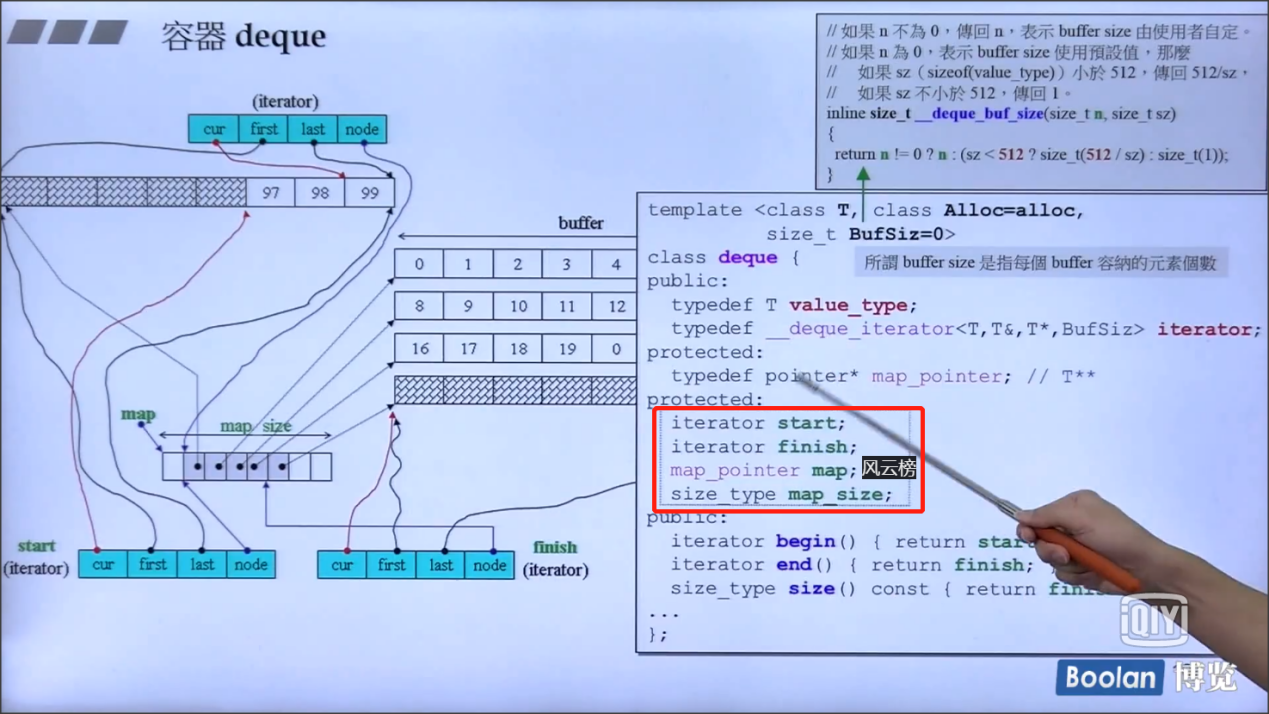
所以，我们采用均摊分析的方法可知，vector 中 push\_back 操作的时间复杂度为常量时间.​

## STL---------Deque

Deque是一个双向数组，它的内部是分段连续的，中间穿插了操作符重载，让使用者感觉整个deque是连续的。Deque的底层是一个指针数组，数组的每个位置指向一段大小是512字节的buffer。//Deque模板类结构主要是由起始位置迭代器start、结尾迭代器finish、数组指针map以及一个四个字节的map\_size参数构成。Deque分段连续特点，导致其迭代器较复杂，deque的迭代器包含了4个指针，这4个指针分别指向了buffer的起始位置、结束位置、当前访问元素以及哈希槽位置，所以总的来说，deque的sizeof大小应该是40（32位机器）。

（deque如何模拟连续空间）deque是分段连续空间，维持其"整体连续"假象的任务，落在了迭代器的operator++ 和 operator-- 两个运算符重载身上。在进行++或者--操作之后，判断迭代器是否越界，越界则重置节点，去访问数组下一位对应的buffer空间。

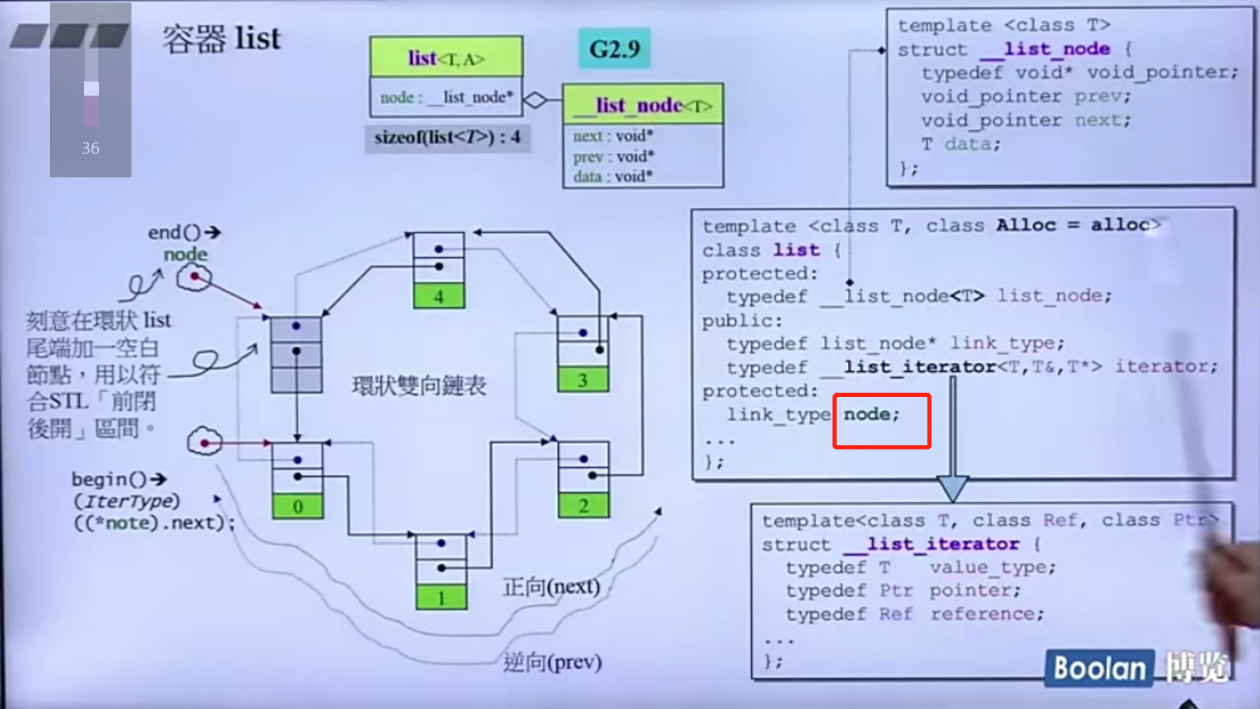
（deque扩容，扩容底层数组）这里数组其实是一个vector，当前后预留的空间不足的时候会扩充，扩充之后的旧数据会放在vector的中间（因为前后都可以放）。



## STL--------List

List其实是一个双向环状链表，在STL中，为了满足容器前闭后开的约定，List刻意在尾端增加了一个空白节点。List模板类中的的数据实际上只有一个节点指针，因此List对象的sizeof大小为指针大小，32位下为4字节，64位机器下为8字节，List模板类中还使用typedef定义了迭代器。在GNU 2.9版本中，该节点的结构体仅仅包含了两个void类型的指针，所以使用的时候还需要转型。

List的迭代器实际上是一个封装类，进行了大量的操作符重载，包括但不限于解地址符号\*，指针的->操作符，++操作，这使得迭代器使用起来和指针一样简单。特别说明关于++操作的重载，STL迭代器为了和C++整数的++操作对应起来，即可以连续的进行++ ++i 但是不可以进行连续的i++ ++，STL对++操作的重载进行规定，前++返回引用类型，后++本质上是调用前++，返回一个对象。



# 操作系统总结

## ---------------------------进程、线程、协程-----------------------

## 线程、进程和协程概念、区别以及运行状态+进程空间布局

（进程概念）进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。

（线程概念）线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享进程的**堆和方法区资源**，但**每个线程有自己的程序计数器、虚拟机栈和本地方法栈**，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

（协程概念）协程是一种比线程更加轻量级的存在。协程完全由程序所控制（在用户态执行），带来的好处是性能大幅度的提升 一个操作系统中可以有多个进程；一个进程可以有多个线程；同理，一个线程可以有多个协程。由于协程切换是在线程内完成的，涉及到的资源比较少，几乎只是切换了寄存器和协程栈的内容。

**操作系统开一个进程要分配哪些资源：**

每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段。

**进程和线程的区别：**

（1）进程是资源分配的最小单位，线程是程序执行的最小单位（资源调度的最小单位）

（2）进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。 线程是共享进程中的数据的，使用相同的地址空间，因此CPU切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。

（3）线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC)进行。

（4）但是多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。

**协程的作用**：

IO型的操作，利用协程在IO等待时间就去切换执行其他任务，当IO操作结束后再自动回调，那么就会大大节省资源并提供性能，从而实现异步编程（不等待任务结束就可以去执行其他代码）。

**协程和线程的区别：**

（1）由于协程的特性, 适合执行大量的I/O 密集型任务, 而线程在这方面弱于协程。协程涉及到函数的切换, 多线程涉及到线程的切换, 所以都有执行上下文, 但是协程不是被操作系统内核所管理, 而完全是由程序所控制（也就是在用户态执行）, 这样带来的好处就是性能得到了很大的提升, 不会像线程那样需要在内核态进行上下文切换来消耗资源，因此协程的开销远远小于线程的开销。

（2）同一时间, 在多核处理器的环境下, 多个线程是可以并行的，但是运行的协程的函数却只能有一个，其他的协程的函数都被暂停, 即协程是并发的。

（3）由于协程在同一个线程中, 所以不需要用来守卫临界区段的同步性原语（primitive）比如互斥锁、信号量等，并且不需要来自操作系统的支持。在协程之间的切换不需要涉及任何系统调用或任何阻塞调用

（4）通常的线程是抢先式(即由操作系统分配执行权), 而协程是由程序分配执行权

线程的六种状态：新创建、可运行、被阻塞、等待、计时等待、被终止。

进程的五种状态：创建状态、运行状态、就绪状态、阻塞状态、退出状态。

**进程的解释如下：**

（1）三种基本状态，分别是：运行态，这个时候进程占用cpu资源，并在cpu上运行，一个事件同时最多运行的cpu数量由机器cpu核心数决定。如果cpu已经被占满了，并且程序所需的所有资源都已经就绪了，例如网络接收缓冲区满了，这个时候进程就进入了就绪态，只需要cpu就可以立马运行。如果进程需要的资源被占用，那么进程就需要等待，这个状态就是进程的阻塞态。

（2）还有两个状态是，进程或者线程创建的时候，操作系统需要为进程或者线程分配所需的内存空间，初始化进程PCB或者线程的TCB等，这个时候进程处于创建态。当程序运行结束的时候，或者因为bug导致进程无法执行下去，操作系统需要回收分配的内存资源、撤销分配的PCB以及TCB，这个状态是进程的**终止态**。

（3）如果进程/线程不断的创建，系统资源已经不能满足进程运行的需求，这个时候就必须把某些进程挂起到磁盘镜像中，不参与进程的调度，起到平滑系统操作负荷的目的，这个时候就对应了进程或者线程的**挂起等待态、挂起就绪态**。常见的需要将进程或者线程挂载到磁盘的情况：

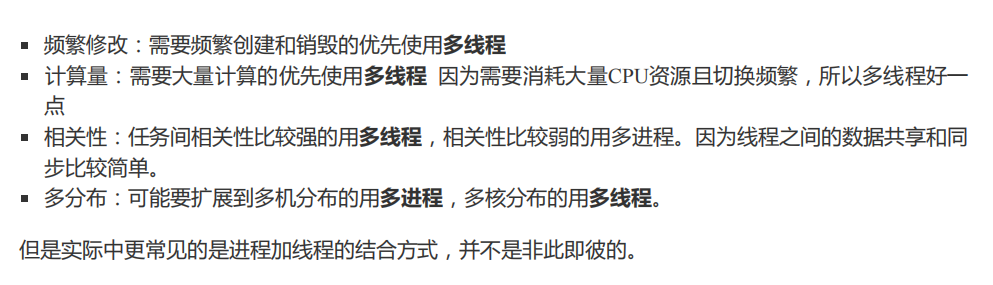
a、系统中的进程或者线程均处于等待或者阻塞状态，处理器空闲。

b、进程或者线程太多、系统资源不足的时候。

c、换出定期执行的任务，减轻系统负担。

进程的空间布局：程序段、初始化过的数据、未初始化过的数据、栈、堆。

## 多进程、多线程应该怎么去选



## 进程调度的算法

**先来先服务、短作业优先、优先级调度、高性价比优先、最短剩余时间优先（短作业优先的抢占版）、时间片轮转、多级反馈队列。**

（1）先来先服务：先进就绪队列，则先被调度，先来先服务是最简单的调度算法。当前面任务耗费很长时间执行，那么后面的任务即使只需要执行很短的时间，也必须一直等待，属于非抢占式。

（2）时间片轮转：每一个进程会被分配一个时间片，表示允许该进程在这个时间段运行，如果时间结束了，进程还没运行完毕，那么会通过抢占式调度，将CPU分配给其他进程，该进程回到就绪队列。这是一种最简单最公平的调度算法，但是由于进程的切换，需要耗费时间，如果时间片太短，频繁进行切换，会影响效率。如果进程时间片太长，有可能导致排后面的进程等待太长时间。

（3）最短作业优先：就是执行时间短的任务先进行调度，该调度算法对长作业不利，容易造成长作业任务饥饿。

（4）最短剩余时间优先。该算法可以理解最短作业优先和时间片轮转的结合。如果没有时间片的概念，那么最短剩余时间其实就是最短作业时间，因为每个进程都是从头执行到尾。

（5）优先级调度算法：根据任务的优先级进行任务的调度，如果多个进程优先级相同，则按照先来先服务的方式依次执行。优先级调度可以进一步细分为抢占式和非抢占式。

非抢占式：表示一旦该进程占有CPU就将一直执行到拮束或者阻塞。

抢占式：进程执行期间，一旦有更高优先级的进程进入就绪队列，那么该进程就会被暂停，重回就绪队列，让更高优先级的进程执行。但是为了防止最高优先级进程一直执行，每个进程依然有自己的时间片，每次时间片结束后，会根据一定规则降低该进程优先级，避免某些最高优先级长作业进程一直占用CPU。 但是优先级调度可能会导致低优先级的进程永远不会执行。

（6）多级反馈队列：多级反馈队列调度基于时间片轮转和优先级调度，设置多个就绪队列，赋予每个就绪队列优先级，优先级越高的队列进程的时间片越短。当有新的进程创建时，先进入第1级就绪队列，时间片结束之前就运行完毕，则终止，否则进入第2级队列等待下一次调度，只有高优先级执行完毕之后才会执行低优先级任务队列。同一优先级的任务队列按时间片轮转执行。为了避免进程饥饿问题，每经过一段时间就会将系统中所有工作重新加入最高优先级队列。

## 同一个进程内的多个线程共享哪些信息，不共享哪些信息？

线程的栈区、程序计数器、栈指针以及函数运行使用的寄存器是线程**私有的**，统一称为**线程上下文**。有时候我们还可以使用**TLS线程局部存储的技术**，TLS表示存放在该区域中的变量是全局变量，所有线程都可以访问；虽然看上去所有线程访问的都是同一个变量，但这个全局变量独属于一个线程，一个线程对此变量的修改对其他线程不可见。

进程中的线程除了线程上下文信息以外其他的内容都是可以共享的，包括代码区（编译后的可执行机器指令，从可执行文件加载到内存的）、堆区、全局变量、静态变量。

## 不同的线程可以互相访问对方的堆和栈吗？

* 不同进程内的线程肯定不能，因为有进程地址保护机制。
* 相同进程内的堆区是共享的能够相互访问，栈有两种方式也能进行访问：因为不像进程地址空间之间的严格隔离，线程的栈区没有严格的隔离机制来保护，因此如果一个线程能拿到来自另一个线程栈上的指针，那么该线程就可以改变另一个线程的栈区，也就是说这些线程可以任意修改属于另一个线程栈区中的变量；**TLS线程局部存储的技术**，TLS表示存放在该区域中的变量是全局变量，所有线程都可以访问；虽然看上去所有线程访问的都是同一个变量，但这个全局变量独属于一个线程，一个线程对此变量的修改对其他线程不可见。

## 进程上下文切换、线程上下文切换、中断上下文切换

**进程上下文切换：**

进程是由内核来管理和调度的，进程的切换只能发生在**内核态**，因此进程的上下文不仅包括了**虚拟内存、栈、全局变量等用户空间的资源，还包括了内核堆栈、寄存器等内核空间的状态**。

因此进程的上下文切换就比系统调用（用户态到内核态的转变）时多了一步：在保存当前进程的内核状态和 CPU 寄存器之前，需先把该进程的虚拟内存、栈等保存下来；而加载了下一进程的内核态后，还需要刷新进程的**虚拟内存和用户栈（虚拟内存的块表失效了）**。

**进程上下文切换过程：**

1. 保存处理机上下文，包括程序计数器和其他寄存器。
2. 更新PCB信息。
3. 把进程的PCB移入相应的队列，如就绪、在某事件阻塞等队列。
4. 选择另一个进程执行，并更新其PCB。
5. 更新内存管理的数据结构。
6. 恢复处理机上下文。

**线程上下文切换：**

线程与进程最大的区别在于，线程是操作系统调度的最小单位，而进程是操作系统分配资源的最小单位。

如果前后两个线程属于不同进程。此时因为资源不共享，所以切换过程就跟进程上下文切换是一样的。前后两个线程属于同一个进程。此时虚拟内存是共享的，上下文切换时，虚拟内存这些资源保持不动，只需要切换线程的私有数、寄存器等不共享的数据。

**进程切换和线程切换的区别：**

因为每一个进程都有自己的虚拟地址空间，进程内的线程共享进程的虚拟地址空间。

当**进行进程切换的时候需要涉及虚拟地址空间的切换，而线程不会**，虚拟地址空间切换后就会导致快表失效，重新进行查询的话就会导致效率下降，所以当计算量大的时候一般会采用多线程的方式防止频繁地切换浪费资源。

**中断上下文切换：**

为了快速响应硬件的事件，中断处理会打断进程的正常调度和执行，转而调用中断处理程序，响应设备事件。而在打断其它进程时，就需要将进程当前的状态保存下来，这样在中断结束后，进程仍然可以从原来的状态恢复运行；即便中断过程打断了一个正在用户态的进程，也不需要保存和恢复这个进程的虚拟内存、全局变量等用户态资源。中断上下文其实只包括内核态中断服务程序执行所必需的状态，包括 CPU 寄存器、内核堆栈、硬件中断参数等。

## 什么时候需要进行上下文切换

* 在一些**调度算法**中，有的进程有时候要暂时离开CPU，例如时间片完或者资源不足，然后让另一个进程进来CPU运作，此时需要进行上下文切换。
* 中断处理：在接受到中断信息的时候，CPU必须要进行上下文切换。
* 内核态和用户态的转换。

## 进程PCB，进程控制块的内容

1. 程序ID（PID、进程句柄）：它是唯一的，一个进程都必须对应一个PID。PID一般是整形数字。
2. 特征信息：一般分系统进程、用户进程、或者内核进程等
3. 进程状态：运行、就绪、阻塞，表示进程现的运行情况
4. 优先级：表示获得CPU控制权的优先级大小
5. 通信信息：进程之间的通信关系的反映，由于操作系统会提供通信信道
6. 现场保护区：保护阻塞的进程用
7. 资源需求、分配控制信息
8. 进程实体信息，指明程序路径和名称，进程数据在物理内存还是在交换分区（分页）中
9. 其他信息：工作单位，工作区，文件信息等。

## 线程TCB，线程控制块的内容

1、threadID：线程的唯一标识。

2、status：线程的运行状态

3、register：线程关于CPU中寄存器的情况

4、PC程序计数器：线程执行的下一条指令的地址

5、优先级：线程在操作系统调度的时候的优先级

6、线程的专属存储区：线程单独的存储区域

7、用户栈：线程执行的用户方法栈，用来保存线程当前执行的用户方法的信息

8、内核栈：线程执行的内核方法栈，用来保存线程当前执行的内核方法信息。

## 用户态和内核态的概念、优势、转换方式

内核态: CPU可以访问内存所有数据, 包括外围设备, 例如硬盘, 网卡。CPU也可以将自己从一个程序切换到另一个程序。

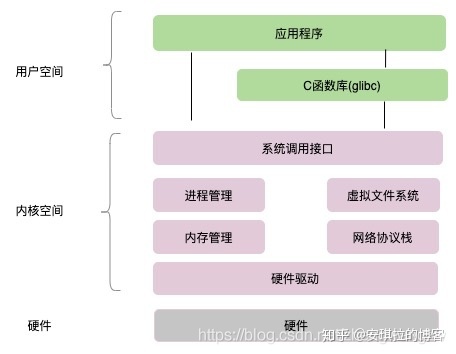
用户态:只能受限的访问内存, 且不允许访问外围设备。占用CPU的能力被剥夺, CPU资源可以被其他程序获取。

**内核态和用户态的优势：**运行在内核态的程序可以访问的资源多，但可靠性、安全性要求高，维护管理都较复杂；用户态程序程序访问的资源有限，但可靠性、安全性要求低，自然编写维护起来比较简单。一个程序到底应该运行在内核态还是用户态则取决于其对资源和效率的需求。

**用户态转为内核态的三个方式：**

* 系统调用：用户态进程主动切换到内核态的方式，用户态进程通过系统调用申请使用操作系统的提供的程序完成操作，系统调用本身就是中断。
* 异常：当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。
* 外设中断：当外设完成用户的请求时，会向CPU发送中断信号。

## 用户空间和内核空间的内容(内核态和用户态的操作有哪些)



## 系统调用的概念及其过程

系统内核通过包装一些能够实现特定功能的特殊硬件指令和硬件状态，即我们所说的内核函数，通过一组系统调用的接口呈现给用户，平常我们的进程几乎都是用户态，读取用户数据，当涉及到系统级别资源的操作（例如文件管理、进程控制、内存管理等）的时候，就要用到系统调用了。

**过程：**

当系统发起调用时通常通过一个陷阱(trap)特殊硬件指令，硬件将控制转移到预先指定的陷阱处理程序（trap handler）（即预先设置的操作系统），并同时将特权级别提升到内核模式。

在内核模式下，操作系统可以完全访问系统的硬件，因此可以执行诸如发起I/O请求或为程序提供更多内存等功能。

当操作系统完成请求的服务时，它通过特殊的陷阱返回（return-from-trap）指令将控制权交还给用户，该指令返回到用户态，同时将控制权交还给应用程序，回到应用离开的地方。

## 守护进程、僵尸进程和孤儿进程

**守护进程：**指在后台运行的，没有控制终端与之相连的进程。它独立于控制终端，周期性地执行某种任务，Linux的大多数服务器就是用守护进程的方式实现的。

**僵尸进程：**如果子进程先退出，父进程还没退出，那么子进程必须等到父进程捕获到了子进程的退出状态才真正结束，否则这个时候子进程就成为僵尸进程。

设置僵尸进程的目的是维护子进程的信息，以便父进程在以后某个时候获取。这些信息至少包括进程ID，进程的终止状态，以及该进程使用的CPU时间，所以当终止子进程的父进程调用wait或waitpid时就可以得到这些信息。如果一个进程终止时该进程有子进程处于僵尸状态，那么它的所有僵尸子进程的父进程将被重置为init进程并且清理它们。

**孤儿进程：**如果父进程先退出，子进程还没退出，那么子进程的父进程将变为init进程。（注：任何一个进程都必须有父进程）。一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将变为init进程，并由init进程对它们完成状态收集工作。

## 如何避免僵尸进程

1. 父进程可将SIGCHLD的处理函数设为SIG\_IGN（亦为默认设定）通知内核对子进程的结束不关心，由内核回收。这种忽略SIGCHLD信号的方法，常用于并发服务器的性能的一个技巧因为并发服务器常常fork很多子进程，子进程终结之后需要服务器进程去wait清理资源。如果将此信号的处理方式设为忽略，可让内核把僵尸子进程转交给init进程去处理，省去了大量僵尸进程占用系统资源。
2. 如果父进程很忙可以用signal注册信号处理函数，在信号处理函数调用wait/waitpid等待子进程退出。
3. 通过两次调用fork。父进程首先调用fork创建一个子进程然后waitpid等待子进程退出，子进程再fork一个子孙进程后退出。这样子进程退出后会被父进程等待回收，而对于子孙进程其父进程已经退出所以子孙进程成为一个孤儿进程，孤儿进程由init进程接管，子孙进程结束后，init会回收。

## 线程结束的四种方式

* 线程运行结束：程序运行结束，线程自动结束。
* 使用自定义标志退出线程：在一般情况下，在 run 方法执行完毕的时候，线程会正常结束。然而，有些线程是后台线程，需要长时间运行，只有在系统满足某些特殊条件后，才能退出这些线程。这时可以使用一个变量来控制循环，并通过设置这个变量为 true 或 false 来控制 while 循环是否退出。
* 使用Interrupt方法结束线程：线程处于阻塞状态时调用interrupt方法时，会抛出InterruptException异常。通过代码捕获异常，然后通过break跳出循环状态。从而能够结束这个线程的执行；当线程未处于阻塞状态时，需要使用interrupted()判断线程的中断标识来退出循环。当使用interrupt()方法时，中断标志会置为true，此时使用自定义标志来控制循环是一样的道理。
* Stop方法终止线程：使用thread.stop()来强行终止线程，但是stop方法很危险，因为在调用之后创建子线程的线程会抛出ThreadDeatherror的错误，并且会释放子线程所持有的所有锁，而一般加锁的代码块，都是为了保护数据的一致性，如果在调用thread.stop()后导致该线程所持有的所有锁突然释放（不可控制），可能会导致被保护的数据呈现不一致的情况。

## 线程假死情况

“假死”的现象就是全部线程都进入了WAITING状态（死锁），则程序就不再执行任何业务功能了，整个项目呈停止状态。当1个生产者多个消费者或多个生产者多个消费者的场景下，用notify通知wait线程是容易出现假死线程，主要有两种情况：

1、notify的唤醒消息发出后只能唤醒一个进入wait的线程，但只是唤醒它，让CPU有调度它的可能性，并不是说一定会执行这个被唤醒的线程，是否被执行是随机的；

2、唤醒了同类的线程，如多个消费者时，当1个消费者消费后应该唤醒异类即生产者的线程，但也可能唤醒的是同类的线程，如果唤醒的是同类线程就有进入假死状态的可能，所以在这样的场景下应该尽可能的使用notify\_all。

## 线程崩溃会影响整个进程吗

对C++来说，线程崩溃会导致整个进程的崩溃（java不会）。在进程中，各个线程的地址空间是共享的，既然是共享，那么某个线程对地址的非法访问就会导致内存的不确定性，进而可能会影响到其他线程，这种操作是危险的，操作系统会认为这很可能导致一系列严重的后果，于是干脆让整个进程崩溃。

**（进程崩溃原理-信号机制）**进程崩溃是通过信号实现的，当操作系统检测到异常的时候会调用kill系统调用发送信号，崩溃通常发送SIGSEGV (Segmentation Violation/Fault)信号，进程收到操作系统发的信号之后CPU 暂停当前程序运行，并将控制权转交给操作系统，之后操作系统根据情况执行相应的信号处理程序，一般执行完信号处理程序逻辑后会让进程退出。

## 线程在操作系统层面如何实现的(操作系统如何对线程提供支持)

从操作系统的层面来说，线程的实现有三种方式：**内核支持线程，用户级线程，组合方式（前两种的结合）**。

内核支持线程：内核支持线程在内核的支持下运行，在内核空间内完成创建，阻塞，撤销，切换。每一个内核线程设置了一个线程控制块TCB，可通过TCB来感知内核线程的存在，进而加以控制。

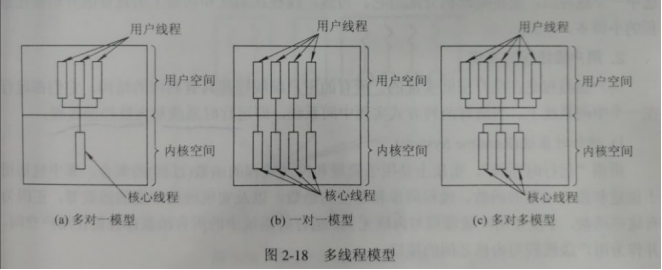
* **优点：**1.在多处理器系统中，内核能够同时调用同一进程的多个线程并行（真正的同时执行）执行。2.一个进程中的线程被阻塞了，内核可以调度该进程中的其他线程或者其他进程中的线程占有处理器运行。3.线程切换较快，切换开销小。4.内核本身也可采用多线程技术，可以提高系统的执行速度和效率。
* **缺点：**1.对于用户的线程的切换，需要从用户态转到内核态中进行，切换的开销较大。

用户级线程：用户级线程在用户空间中实现，对线程的创建，撤销，同步，通信都无需内核的支持，一个系统中用户级线程可以达到数百个至数千个。

* **优点：**1.线程的切换不需要转换到内存空间，因此节省了模式切换的开销2.调度算法是专用的，不同的进程可以有不同的调度算法。3.用户级线程的实现与OS平台无关，因此用户级线程可以在不持之线程机制的操作系统平台上运行
* **缺点：**1.一个线程执行系统调用会阻塞该进程的所用线。2.单纯的用户级线程实现方式中，内核每次只分配一个CPU给进程，因此每次只能有一个线程获得CPU执行，其他线程在该线程放弃CPU之前只能等待。

组合方式：组合方式是内核支持线程和用户级线程的组合。根据用户级线程和内核支持线程连接方式的不同可划分出三种模型，**多对一模型，一对多模型，多对多模型**。

* **多对一模型：**用户级线程对内核支持线程进行多路复用，一次只允许一个用户级线程映射到内核支持线程对内核进行访问，当一个线程阻塞时，其他线程也必须阻塞。
* **一对一模型：**一个用户级线程映射一个内核支持线程，一个线程阻塞时，允许另一个线程调度运行。每创建一个用户级线程都需要创建一个内核支持线程与之对应，开销较大。
* **多对多模型：**相同数量的用户级线程可以映射到相同数量或者更少的内核支持线程，综合了上述两种模型的优点。



## 操作系统缓冲原理，cache(高速缓存)

> 使用的原因：由于cpu访问速度与内存访问速度差异较大的原因，为了加快cpu访问主存的速度，在cpu与内存之间加入了cache高速缓存器，以此提高cpu访问内存的速度。

> 程序的执行过程中遵循局部性原理，即时间局部性原理和空间局部性原理。时间局部性是指程序中的一条指令一旦执行，不久后改指令还可能再次被执行。空间局部性指的是一旦程序访问了某个存储单元，在不久后，其附近的存储单元也会被访问

> Cache就是借助局部性原理把主存中被频繁访问的活跃程序块和数据块复制到 cache 中，分为指令cache和数据cache

> 内存放入cache规则：

直接映射（direct）：每个主存块映射到 cache 的固定行中。

全相连映射（full associate）：每个主存块映射到 cache 的任意行中。

组相连映射（set associate）：每个主存块映射到 cache 的固定组的任意行中

## CPU多核并行情况分析（CPU缓存一致性：cache）

CPU速度远高于内存，为了协调CPU和内存在速度上的差异，在CPU中增加了高速缓存，从下到上依次是L3 cache, L2 cache, L1 cache，越靠近cpu的缓存速度越快，对多核CPU来说，每个核心都含有一套L1（甚至和L2）缓存，而共享L3（或者和L2）缓存。多CPU的每个CPU都相互独立，拥有自己的缓存，CPU之间无法共享缓存。在并行的情况下可能会出现多个线程同时写各自的cache的情况，因此CPU有“缓存一致性”原则，即每个处理器（核）都会通过嗅探在总线上传播的数据来检查自己的缓存值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器（核）的缓存更新。因此，在多核多线程的场景下，在声明变量时应该使用volatile，volatile变量要求在更新了缓存之后立即写入到系统内存，而非volatile变量，则是CPU修改缓存，而cpu会在适当的时候（MESI协议）将缓存数据写入内存。

## 多核CPU的缓存一致性协议（MESI）

每个处理器都有自己的高速缓存，而又共享同一主内存。当多个处理器都涉及同一块主内存区域的更改时，将导致各自的的缓存数据不一致，因此就有了缓存一致性协议，最常用的就是MESI。

MESI是修改、独占、共享、失效四个单词的缩写，是用来修饰缓存行的状态。在每个缓存行前额外使用2bit，来表示此四种状态：

1. 修改：该缓存行仅出现在此cpu缓存中，缓存已被修改，和内存中不一致，等待同步至内存。
2. 独占：该缓存行仅出现在此cpu缓存中，缓存和内存中保持一致。
3. 共享：该缓存行可能出现在多个cpu缓存中，且多个cpu缓存的缓存行和内存中的数据一致。
4. 失效：由于其他cpu修改了缓存行，导致本cpu中的缓存行失效。

**（（（（（缓存行的概念）））））**

**缓存和内存交换数据的最小单位，缓存行是CPU缓存中可分配、操作的最小存储单元。与CPU架构有关，通常有32字节、64字节、128字节不等。目前64位架构下，64字节最为常用。**

在MESI协议中，每个缓存行不仅知道自己的读写操作，而且也监听其它缓存行的读写操作。每个缓存行的状态根据本cpu和其它cpu的读写操作在4个状态间进行迁移，MESI的监听机制是这样的：

1. 当缓存行处于修改状态时，会时刻监听其他cpu对该缓存行对应主内存地址的读取操作，一旦监听到，将本cpu的缓存行写回内存，并标记为共享状态
2. 当缓存行处于独占状态时，会时刻监听其他cpu对该缓存行对应主内存地址的读取操作，一旦监听到，将本cpu的缓存行标记为共享状态
3. 当缓存行处于共享状态时，会时刻监听其他cpu对使缓存行失效的指令（即其他cpu的写入操作），一旦监听到，将本cpu的缓存行标记为失效状态（其他cpu进入修改状态）
4. 当缓存行处于失效状态时，从内存中读取，否则直接从缓存读取。

相当于当某个cpu修改缓存行数据时，其他的cpu通过监听机制获悉共享缓存行的数据被修改，会使其共享缓存行失效。本cpu会将修改后的缓存行写回到主内存中。此时其他的cpu如果需要此缓存行共享数据，则从主内存中重新加载，并放入缓存，以此完成了缓存一致性。

## MESI缓存伪共享问题

在多核多线程并发场景下，多核要操作的不同变量处于同一缓存行，某cpu更新缓存行中数据，并将其写回缓存，同时其他处理器会使该缓存行失效，如需使用，还需从内存中重新加载。这对效率产生了较大的影响。

/////\*（具体场景）假设现在的共享的L3缓存的缓存行中存在两个变量，x和y，假设这个时候core1需要操作变量x，core2需要操作变量y。

（1）core1修改缓存行内的变量x后，按照缓存一致性协议MESI，core2需将缓存行置为失效，core1将最新缓存行数据写回内存。

（2）core2需重新从内存中加载包含变量y的缓存行数据，并放置缓存。如果core2修改变量y，需要core1将缓存行置为失效，core2将最新缓存写回内存。

（3）core1或其他处理器如需操作同一缓存行内的其他数据，同上述步骤。\*//////

（缓存伪共享问题解决思路）空间换时间。以64字节的缓存行为例，伪共享问题产生的前提是，并发情况下，不同cpu对缓存行中不同变量的操作引起的。那么，如果把缓存行中仅存储目标变量，其余空间采用“无用”数据填充补齐64字节，就不会才产生伪共享问题。这种方式就是：缓存行填充（也称缓存行对齐）。

## ----------------------进程、线程的通信和同步-------------------

## 线程（ITC）、进程（IPC）的通信方式

进程：管道、信号、信号量、socket、共享内存、消息队列。

（1）管道/匿名管道(Pipes) ：用于具有亲缘关系的父子进程间或者兄弟进程之间的通信。只存在于内存中的文件。

（2）有名管道(Names Pipes) : 匿名管道由于没有名字，只能用于亲缘关系的进程间通信。为了克服这个缺点，提出了有名管道。有名管道严格遵循先进先出(first in first out)。有名管道以磁盘文件的方式存在，可以实现本机任意两个进程通信。

（3）信号(Signal) ：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。信号包括：sigkill：用来立即结束程序的运行，sigfpe：在发生算数错误时发出，sigstop：用于暂停一个进程，sigtstp：挂起键（ctrl+z）会造成进程挂起；

**附：**kill pid 或者 kill -15 pid向系统发送一个SIGTERM信号给对应的程序，程序接收到该signal后，有三种情况：（1）程序立刻停止（2）当程序释放相应资源后再停止（3）程序可能仍然继续运行。 因而SIGTERM可能发生阻塞，忽略情况。而kill -9 pid系统发送SIGKILL，用来强制尽快终止一个进程，接收到该信号的程序应该立即结束运行，不能被阻塞或者忽略。

（4）消息队列(Message Queuing) ：消息队列是消息的链表,具有特定的格式,存放在内存中并由消息队列标识符标识。**管道和消息队列的通信数据都是先进先出的原则。与管道（无名管道：只存在于内存中的文件；命名管道：存在于实际的磁盘介质或者文件系统）不同的是消息队列存放在内核中，只有在内核重启(即，操作系统重启)或者显示地删除一个消息队列时，该消息队列才会被真正的删除。消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。**

（5）信号量(Semaphores) ：信号量是一个计数器，用于多进程对共享数据的访问，信号量的意图在于进程间同步。这种通信方式主要用于解决与同步相关的问题并避免竞争条件。

**信号量是一个整型变量，用来记录共享资源的数量，可以对其执行 down 和 up 操作，也就是常见的P和V操作，执行P操作首先判断当前信号量是否为空，不为空则将其-1，然后令其访问共享资源，然后放入就绪队列；如果当前信号量为0，则将其睡眠并放入等待队列中。当执行V操作时，首先会判断等待队列是否为空，若为空则取出队头将其唤醒放入就绪队列中，若等待队列为空则将信号量个数+1。**

（6）共享内存(Shared memory) ：使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等。可以说这是最有用的进程间通信方式。

**查看共享内存：ipcs -m命令查看共享内存**。

（7）套接字(Sockets) : 此方法主要用于在客户端和服务器之间通过网络进行通信。套接字是支持TCP/IP的网络通信的基本操作单元，可以看做是不同主机之间的进程进行双向通信的端点，简单的说就是通信的两方的一种约定，用套接字中的相关函数来完成通信过程。

线程：共享内存、条件变量、furture

1. 共享内存（全局变量）：比较方便的一种方式，因为线程间共享进程的资源。
2. 通过条件变量进行线程间的通信：当发生某个条件的时候去唤醒或者通知另一个线程。
3. 通过std::furture来进行线程间的通信：通过一个回调的方式获得一个线程异步的结果。

## 共享内存的实现方式（C++接口使用）

**查看共享内存：ipcs -m命令查看共享内存**。

**系统V共享内存：**

进程间需要共享的数据被放在一个叫做IPC共享内存区域的地方，这块区域由操作系统内核管理，每一个共享内存区都有一个shmid\_kernel的结构体对象，LINUX规定最多只有128个共享内存区域。具体的共享内存流程是，进程通过调用shmget（Shared Memory GET）来分配一个共享内存块并返回一个标识符，操作系统内核在创建共享内存区相应的shmid\_kernel结构体的同时，还将在特殊文件系统shm中，创建并打开一个同名文件，这个文件不属于任何一个进程（任何进程都可以访问）。结构体中的shm\_file参数存储了被映射文件的地址，经过这样的操作之后共享内存区域就与shm文件系统中的文件对应起来。在创建了一个共享内存区域后，还要将它映射到进程地址空间，系统调用shmat()完成此项功能，共享内存的实现也就是一个特殊“文件”的读写过程。

**Linux中mmap实现共享内存：**mmap()系统调用使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以向访问普通内存一样对文件进行访问，不必再调用read()，write（）等操作。

C++中该共享内存方式的接口：

1. 创建共享内存(shmget函数：int shmget(key\_t key, size\_t size, int shmflg)，共享内存标识符+共享内存大小+创建方式+权限(表示自己的读写权限)-->>返回-1表示失败，否则返回共享内存的操作句柄)。
2. 附加共享内存到进程(shmat函数：共享内存操作句柄+进程对共享内存的操作属性)-->>返回值表示为附加到共享内存当中的虚拟地址，如果返回NULL则表示附加失败，此函数其实是将通过页表映射将物理内存的首地址映射到进程虚拟地址空间中的共享区中的一个位置。
3. 分离共享内存(shmdt函数)，根据返回的共享区地址将附加的映射关系去掉。

共享内存操作函数(shmctl函数：共享内存操作句柄+共享内存属性对应的结构体信息)。

## 系统V和mmap()实现共享内存的区别

1、 系统V共享内存中的数据，从来不写入到实际磁盘文件中去；而通过mmap()映射普通文件实现的共享内存通信可以指定何时将数据写入磁盘文件中。 系统V共享内存机制实际是通过映射特殊文件系统shm中的文件实现的，文件系统shm的安装点在交换分区上，系统重新引导后，所有的内容都丢失。

2、 系统V共享内存是随内核持续的，即使所有访问共享内存的进程都已经正常终止，共享内存区仍然存在（除非显式删除共享内存），在内核重新引导之前，对该共享内存区域的任何改写操作都将一直保留。

3、 通过调用mmap()映射普通文件进行进程间通信时，一定要注意考虑进程何时终止对通信的影响。而通过系统V共享内存实现通信的进程则不然。

## 进程同步的四个方法

1. **临界区**：对临界资源进行访问的那段代码称为临界区。为了互斥访问临界资源，每个进程在进入临界区之前，需要先进行检查。
2. **同步与互斥**：同步：多个进程因为合作产生的直接制约关系，使得进程有一定的先后执行关系；互斥：多个进程在同一时刻只有一个进程能进入临界区。
3. **信号量**：信号量（Semaphore）是一个整型变量，可以对其执行 down 和 up 操作，也就是常见的P和V操作。P操作表示如果信号量大于0，执行减1操作；如果信号量等于0，进程睡眠，等待信号量大于0；V操作对信号量执行 +1 操作，唤醒睡眠的进程让其完成P操作。

PV操作需要被设计成原语，不可分割，通常的做法是在执行这些操作的时候屏蔽中断。

如果信号量的取值只能为0或者1，那么就成为了互斥量（Mutex），0表示临界区已经加锁，1表示临界区解锁。

1. **管程**：使用信号量机制实现的生产者消费者问题需要客户端代码做很多控制，而管程把控制的代码独立出来，不仅不容易出错，也使得客户端代码调用更容易。

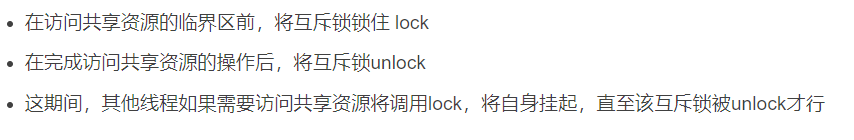
## 线程安全问题or不安全

（1）线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。

（2）线程不安全就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据。

## 线程同步（安全）的方法

Mutex锁：mutex使用上锁和解锁来控制对资源的访问，上锁和解锁都是原子操作。



**c++针对上锁和解锁过程中可能会出现异常的问题，c++11推出了lock\_guard自动释放锁，它使用构造函数加锁和析构函数解锁。**

信号量：是一个整型变量，可以对其执行 down 和 up 操作，也就是常见的P和V操作。P操作表示如果信号量大于0，执行减1操作；如果信号量等于0，进程睡眠，等待信号量大于0；V操作对信号量执行 +1 操作，唤醒睡眠的进程让其完成P操作。

PV操作需要被设计成原语，不可分割，通常的做法是在执行这些操作的时候屏蔽中断。

如果信号量的取值只能为0或者1，那么就成为了刚才说的互斥量（Mutex），0表示临界区已经加锁，1表示临界区解锁。

Windows：

（1）临界区：一段独占对某些共享资源访问的代码。

（2）事件：该方法是windows提供的最灵活的线程同步的方式，事件处于激发和未激发状态。

Linux：

（1）读写锁：读写锁有3种状态：读模式下加锁状态，写模式下加锁状态，不加锁状态。一次只有一个线程可以占有写模式的读写锁，但是多个线程可以同时占有读模式的读写锁。

（2）条件变量：条件变量始终与互斥锁（mutex）一起使用，它是以原子方式阻塞线程，直到某个特定条件满足为止。对条件的测试是在互斥锁（互斥）的保护下进行的。

（3）自旋锁：自旋锁与互斥量类似，但它不是通过休眠使进程阻塞，而是在获取锁之前一直处于忙等状态。

（4）屏障：屏障是协调多个线程并行工作的机制，屏障允许每个线程等待，知道这些线程都达到某个点，然后继续执行。

## C++线程同步的实现方式

线程之间的同步可以通过互斥锁、信号量进制、自旋锁、读写锁、条件变量来进行实现，在C++ 11中提供了<Mutex>、<atomic>、<condition\_variable>来实现上述功能。

1. 互斥锁：C++11中的互斥锁可以通过mutex头文件进行实现。

* **构造函数：**std::mutex不允许拷贝构造，也不允许move拷贝，最初产生的mutex对象是处于unlocked状态的。
* **lock()：**调用线程将锁住该互斥量，线程调用该函数会发生以下3种情况：

（1）如果该互斥量当前没有被锁住，则调用线程将该互斥量锁住，直到调用unlock之前，该线程一直拥有该锁。

（2）如果当前互斥量被其他线程锁住，则当前的调用线程被阻塞住。

（3）如果当前互斥量被当前调用线程锁住，则会产生死锁。

* **unlock()：**解锁，释放对互斥量的所有权。
* **try\_lock()：**尝试锁住互斥量，如果互斥量被其他线程占有，当前线程也不会被阻塞。返回true或者是false。

/\***Mutex提供了四种类型的互斥量**，第一个是std::mutex，这是最基本的mutex类；第二个是std::recursive\_mutex，这是递归mutex类，能多次锁定而不死锁，在实现原理上其实就对mutex加了一个引用计数，保证了递归函数的不死锁；第三个是std::time\_mutex，这是定时mutex类，可以锁定一定的时间，第四个是std::recursive\_timed\_mutex，这是定时递归mutex类。\*/

**为了避免在上锁和解锁之间的程序出现异常情况而导致锁无法正确释放，C++11中提供了unique\_lock和lock\_guard来保证上锁和解锁的实现，它们通过构造函数上锁，通过析构函数进行锁释放。**

**（unique\_lock和lock\_guard区别）**绝大多数情况下这两种锁是可以互相替代的，区别是unique\_lock比lock\_guard能提供更多的功能特性（但需要付出性能的一些代价），如下：

A、unique\_lock可以实现延时锁，即先生成unique\_lock对象，然后在有需要的地方调用lock函数，lock\_guard在对象创建时就自动进行lock操作了；

B、unique\_lock可以在需要的地方调用unlock操作，而lock\_guard只能在其对象生命周期结束后自动Unlock；

正是由于这两个差异特性，unique\_lock可以用于一次性锁多个锁以及用于条件变量的搭配使用，而lock\_guard做不到。

1. 信号量机制：是一个整型变量，可以对其执行 down 和 up 操作，也就是常见的P和V操作。P操作表示如果信号量大于0，执行减1操作；如果信号量等于0，进程睡眠，等待信号量大于0；V操作对信号量执行 +1 操作，唤醒睡眠的进程让其完成P操作。在C++ 11中可以通过atomic实现信号量计数来实现信号量机制。
2. 自旋锁：自旋锁是一种忙等形式的锁，会再用户态不同的询问锁是否可以获取，不会陷入到内核态中，所以更加高效。缺点是可能会对CPU资源造成浪费。但是在C++11中并没有直接提供自旋锁的实现。但是在C++11中提供了原子操作的实现，可以借助原子操作实现简单的自旋锁。
3. 条件变量：条件变量是一种同步机制，允许线程挂起，直到共享数据上的某些条件得到满足。条件变量上的基本操作有：触发条件(当条件变为 true 时)；等待条件，挂起线程直到其他线程触发条件。条件变量要和互斥量相联结，以避免出现条件竞争，C++ 11可以直接使用condition\_variable实现。
4. 读写锁：C++ 11没有提供读写锁的实现，读写锁的原理就是，可以多次读，但是写只能一次一次的写入。在C++ 11中可以借助<Mutex>和<condition\_variable>来实现读写锁，简单的思路就是写的时候上锁，长时间写可以通过条件变量进行挂起，写完再唤醒其他线程。

## C++原子操作、无锁队列的实现

原子操作：例如可以使用CAS函数（C++atomic类中的无锁函数compare\_exchange\_weak实现）实现各种无锁的数据结构解决并发问题，CAS是原⼦操作的⼀种，可⽤于在多线程编程中实现不被打断的数据交换操作，从而避免多线程同时改写某⼀数据时由于执行顺序不确定性以及中断的不可预知性产⽣的数据不一致问题。

无锁队列其实是采用了CAS函数进行push和pop操作，在做push操作时是通过do-while循环，首先执行CAS操作发现队尾结点后为空，于是就执行do-while中的CAS操作将尾节点\_tail的\_next指针赋值为newNode，然后退出do-while循环，调用第二个CAS操作将尾节点指针向后移动一位；由于CAS是一个原子操作，所以即使同时T2线程了也调用了do-while中的CAS操作，但是其判断p->\_next不为空，因为T1线程已经将尾节点向后移动了，所以其只能继续执行do，将p向后移动，重新移动到尾节点继续重新判断，直到成功为止。

Pop操作和push相似，假设T1先执行CAS操作将\_head向后移动了一位，并且删除了原先的头指针，那么当T2再执行时发现T1更新过后的\_head指针(移动了)与一开始获取的头指针p不相等了，那么就继续执行do-while循环重新获取头指针，然后重新进行CAS操作。

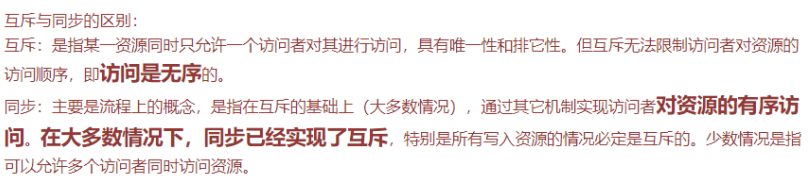
## 多线程的join和detach

在一个线程中，开了另一个线程去干另一件事，使用join函数后，原始线程会等待新线程执行结束之后，再去销毁线程对象。

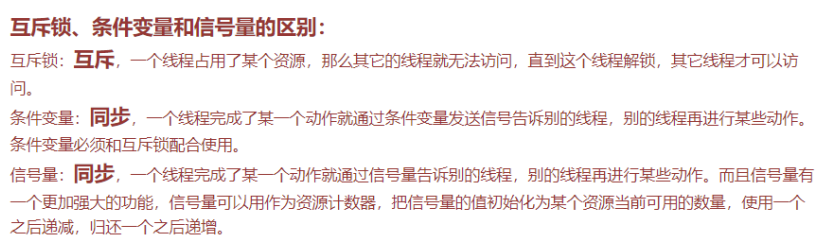
Detach 称为分离线程函数，使用detach()函数会让线程在后台运行，即说明主线程不会等待子线程运行结束才结束。

使用join要等到新线程执行完，再销毁线程对象，这样如果新线程使用了共享变量，等到新线程执行完再销毁这个线程对象，不会产生异常。如果不使用join，使用detach，那么新线程就会与原线程分离，如果原线程先执行完毕，销毁线程对象及局部变量，并且新线程有共享变量或引用之类，这样新线程可能使用的变量，就变成未定义，产生异常或不可预测的错误。

## 互斥与同步的区别



## 互斥、条件变量和信号量的区别



## 互斥锁和信号量的区别和使用场景

**互斥锁量用于线程的互斥，信号量用于线程的同步。**

**互斥锁**是为了保证临界资源只被一个线程访问，是为了保证临界资源的线程安全；因此互斥锁的获取者与释放者必须是同一个线程。

**信号量**虽然也能提供临界资源保护的功能，但是信号量的应用场景并不是为了保护临界资源，只是为了线程间的同步执行；A线程执行完自己的操作后，发送了信号量，B线程获取到信号量后接着执行自己的操作，但是A的操作与B的操作可能没有任何关系，也不会访问相同的临界资源。

## 互斥锁、自旋锁、读写锁（乐观锁和悲观锁）

**互斥锁**：互斥锁是一种「独占锁」，比如当线程 A 加锁成功后，此时互斥锁已经被线程 A 独占了，只要线程 A 没有释放手中的锁，线程 B 加锁就会失败，于是就会释放 CPU 让给其他线程，既然线程 B 释放掉了 CPU，自然线程 B 加锁的代码就会被阻塞。

刚才这种互斥锁加锁失败而阻塞的现象，是由操作系统内核实现的。当加锁失败时，内核会将线程置为「睡眠」状态，等到锁被释放后，内核会在合适的时机唤醒线程，当这个线程成功获取到锁后，于是就可以继续执行。所以互斥锁加锁失败时，会从用户态陷入到内核态，让内核帮我们切换线程，虽然简化了使用锁的难度，但是存在一定的性能开销成本。

**因此如果执行代码的时间比较短，有可能上下文切换的时间比代码执行时间更长，此时就不能选择互斥锁，可以选用自旋锁。**

那这个开销成本是什么呢？会有两次**线程上下文切换**的成本：

* 当线程加锁失败时，内核会把线程的状态从「运行」状态设置为「睡眠」状态，然后把 CPU 切换给其他线程运行；
* 接着，当锁被释放时，之前「睡眠」状态的线程会变为「就绪」状态，然后内核会在合适的时间，把 CPU 切换给该线程运行。

**线程上下文切换：当两个线程是属于同一个进程，因为虚拟内存是共享的，所以在切换时，虚拟内存这些资源就保持不动，只需要切换线程的私有数据、寄存器等不共享的数据。**

**自旋锁：**自旋锁是通过 CPU 提供的 CAS 函数（C++atomic类中的无锁函数compare\_exchange\_weak实现），在「用户态」完成加锁和解锁操作，不会主动产生线程上下文切换，所以相比互斥锁来说，会快一些，开销也小一些。

一般加锁的过程，包含两个步骤：

* 第一步，查看锁的状态，如果锁是空闲的，则执行第二步；
* 第二步，将锁设置为当前线程持有；

CAS 函数就把这两个步骤合并成一条硬件级指令，形成**原子指令**，这样就保证了这两个步骤是不可分割的，要么一次性执行完两个步骤，要么两个步骤都不执行。当发生多线程竞争锁的情况，加锁失败的线程会「忙等待」，直到它拿到锁，这里的「忙等待」可以用 while 循环等待实现。

**读写锁：**读写锁是由「读锁」和「写锁」两部分构成，如果只读取共享资源用「读锁」加锁，如果要修改共享资源则用「写锁」加锁。所以，读写锁适用于能明确区分读操作和写操作的场景。

* 当「写锁」没有被线程持有时，多个线程能够并发地持有读锁，这大大提高了共享资源的访问效率，因为「读锁」是用于读取共享资源的场景，所以多个线程同时持有读锁也不会破坏共享资源的数据。
* 但是，一旦「写锁」被线程持有后，读线程的获取读锁的操作会被阻塞，而且其他写线程的获取写锁的操作也会被阻塞。

写锁是独占锁，因为任何时刻只能有一个线程持有写锁，类似互斥锁和自旋锁，而读锁是共享锁，因为读锁可以被多个线程同时持有，因此读写锁在读多写少的场景，能发挥出优势。

读写锁的实现方式：读优先锁、写优先锁、公平读写锁。

**乐观锁和悲观锁：**自旋锁、互斥锁、读写锁都属于悲观锁，因为我们认为**多线程同时修改共享资源的概率比较高，于是很容易出现冲突，所以访问共享资源前，先要上锁。相反的，如果多线程同时修改共享资源的概率比较低，就可以采用乐观锁，**乐观锁虽然去除了加锁解锁的操作，但是一旦发生冲突，重试的成本非常高，所以只有在冲突概率非常低，且加锁成本非常高的场景时，才考虑使用乐观锁。

## 信号量可以控制进程的资源使用顺序吗？阻塞的信号量是如何唤醒的？

可以，定义n个进程的n个信号量，按要求的执行顺序去重写它们的run方法，执行第一个进程后释放第二个进程的信号，只能在一开始进行顺序控制。

阻塞的信号量是在v操作后，会判断等待队列中是否有需要这个资源的进程，有的话唤醒，没有就资源+1.

## ---------------------------中断和死锁---------------------------

## 操作系统如何响应中断（硬中断和软中断）

中断是指程序执行过程中，遇到急需处理的事件时，操作系统会打断正在执行的进程，调用内核中的中断处理程序来响应请求，中断是一种异步的事件处理机制，可以提高系统的并发处理能力。

由于中断请求会打断其他进程，所以中断处理程序需要尽可能快地执行。有时，中断处理程序在响应中断时，可能会临时关闭中断，这意味着，如果当前的中断处理程序没有执行完之前，系统中的其他中断请求是无法响应的，这可能造成中断的丢失。

**Linux 系统为了解决中断处理程序执行过长和中断丢失的问题，将中断过程分成了两个阶段，分别是「上半部和下半部分」，对应着硬中断和软中断。**

上半部用来快速处理中断，一般会暂时关闭中断请求，主要负责处理跟硬件紧密相关或者时间敏感的事情。

下半部用来延迟处理上半部未完成的工作，一般以「内核线程」的方式运行，主要是负责上半部未完成的工作，通常都是耗时比较长的事情，特点是延迟执行。

以网卡接受网络数据来举例子，网卡收到网络包后，会通过硬件中断通知内核有新的数据到了，于是内核就会调用对应的中断处理程序来响应该事件，**硬中断（中断的上半部分）**做到快速处理，所以只要把网卡的数据读到内存中，然后更新一下硬件寄存器的状态，比如把状态更新为表示数据已经读到内存中的状态值。**软中断**把一些处理比较耗时且复杂的事情，交给「软中断处理程序」去做，也就是中断的下半部，其主要是需要从内存中找到网络数据，再按照网络协议栈，对网络数据进行逐层解析和处理，最后把数据送给应用程序。

## 死锁的四个必要条件和处理策略

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） 不可剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

1. 鸵鸟算法，针对死锁不做任何处理。
2. 死锁的检测和解除，通过检测资源分配图中是否有环路判断其是否有死锁，如果发生死锁则可采用：资源剥夺、撤销进程、进程回退的方式。
3. 死锁预防：破坏其四个必要条件之一。

* **破坏互斥条件**：改造独占性资源为虚拟资源，大部分资源已无法改造。
* **破坏不可剥夺条件**:当一进程占有一独占性资源后又申请一独占性资源而无法满足，则退出原占有的资源。
* **破坏请求与保持条件**:采用资源预先分配策略，即进程运行前申请全部资源，满足则运行，不然就等待，这样就不会占有且申请。
* **破坏循环等待条件**：实现资源有序分配策略，对所有设备实现分类编号，所有进程只能采用按序号递增的形式申请资源。

1. 死锁避免：在动态分配资源的过程中，采用**银行家算法**防止系统进入不安全状态，从而避免死锁

## Linux如何检查死锁信息

1. 使用gdb调试attach对应的进程，然后查看该进程中的线程信息，在根据每个线程中的执行信息，查看死锁情况。
2. 分析core.dump文件也可以看到对应的死锁信息。

## ----------------------------内存部分-----------------------------

## 内存分配的内部碎片和外部碎片

内部碎片就是已经被分配出去（能明确指出属于哪个进程）却不能被利用的内存空间；

内部碎片是处于 （操作系统分配的用于装载某一进程的内存）区域内部或页面内部的存储块。占有这些区域或页面的进程并不使用这个存储块。直到进程释放它，或进程结束时，系统才有可能利用这个存储块。

外部碎片指的是还没有被分配出去（不属于任何进程），但由于太小了无法分配给申请内存空间的新进程的内存空闲区域。

## 内存管理功能（内存分配、地址转换、内存扩充、内存保护）

> 内存的分配与回收：当作业或进程创建后系统会为他们分配内存空间，当结束后内存空间也会被回收。

> 地址转换：将程序中的逻辑地址转换成内存中的物理地址

> 内存空间的扩充：利用虚拟存储技术或自动覆盖技术，从逻辑上扩充内存

> 存储保护：保证个个作业在自己的内存空间内运行，互不干扰

内存的分配与回收：分为连续分配和非连续分配，连续分配根据分配方式不同分为单一连续分配、固定分区分配和动态分区分配（动态分配算法有：首次适应、最佳适应、最坏适应和临近适应算法）；非连续分配分为基本分页存储管理、基本分段存储管理和段页式存储管理。

地址转换：使用MMU将逻辑地址转换为物理地址。

内存保护：为了保护操作系统各个进程直接互不干涉，从而提出了内存保护。内存保护是在cpu中设定了重定位寄存器和界地址寄存器，当进程访问内存的时候就会判断其是否越界。

内存空间的扩充：基于局部性原理使用了虚拟内存来对物理内存进行扩充。

## 逻辑地址转为物理地址：MMU(内存管理单元)

主要将逻辑地址转换为物理地址，程序执行的时候内部一般保存的是逻辑地址，当程序访问逻辑地址的时候，MMU会将其进行转换为相应的物理地址再去访问内存。（快表在MMU中）

1、（如果有块表）查快表，如果命中直接返回物理内存地址。

2、在系统中会设置一个页表寄存器(PTR)，用来储存页表在内存中的起始地址F和页表长度M

3、根据逻辑地址计算出页号和业内偏移量

4、判断页号是否越界

5、查询页表找到页号对应的页表项，确定页面的内存块号(第一次访存，因为页表在内存中)

6、用内存块号和业内偏移量的到物理地址，如果有快表，将物理地址放入快表缓存。

7、访问内存目标单元(第二次访存)

## 页面置换算法

* **最近最久未使用（LRU）：如果一个数据在最近一段时间没有被访问到，那么在将来它被访问的可能性也很小。所以当限定的空间已存满数据时，要当把最久没有被访问到的数据页淘汰。**
* **先进先出（FIFO）：先进先出算法总是淘汰最先进入主存的页面，因此会选择在主存中存在时间最久的页面进行淘汰。**
* **最近最少使用（LFU）：该算法是基于“如果一个数据在最近一段时间内使用次数很少，那么在将来一段时间内被使用的可能性也很小”的思路，在淘汰页面时会选择访问次数最少的数据页进行换出。**

## 内存分配方式（连续、非连续分配、分页、分段、页表相关概念

1. 连续分配：

为用户分配一个连续的内存空间，比如某个作业需要100mb的内存空间，就为这个作业在内存中划分一个100mb的内存空间。连续分配主要有单一连续分配、固定分区分配以及动态分区分配。

**单一连续分配**：将内存去划分为系统区域用户区，系统区为操作系统使用，剩下的用户区给一个进程或作业使用。

特点：操作简单、没有外部碎片，适合单道处理系统。但是会有大量的内部碎片浪费资源，存储效率低。

**固定分区分配**：(1)分区大小相等：将内存的用户区分成大小相等的区域，每个进程只能申请一块区域；(2)分区大小不等：将内存的用户区分成大小不等的区域，分配原则是多个较小的区域、适量中等大小区域、少量的最大分区。每个进程根据大小只能申请一块区域。特点：固定分区分配虽然没有外部碎片，但是会造成大量的内部碎片。分区大小相等缺乏灵活性，大的进程可能放不进去；分区大小不等可能会造成大量的内部碎片，利用率极低。

**动态分区分配：**不会先划分内存区域，当进程进入内存的时候才会根据进程大小动态的为其建立分区，使分区大小刚好适合进程的需要。特点：在开始是很好的，进程一次按照顺序存入内存，但是运行久了以后随着进程的消亡，会出现很多成段的内存空间，时间越来越长就会导致很多不可利用的外部碎片，降低内存的利用率。这时需要分配算法来解决。

**动态分区分配的四个算法：**

（1）首次适应算法：进程进入内存之后从头开始查找第一个适合自己大小的分区。空间分区就是按照地址递增的顺序排列。算法开销小，回收后放到原位置就好。综合看这个算法性能最好。

（2）最佳适应算法：将分区从从小到大排列(容量递增)，找到最适合自己的分区，这样会有更大的分区被保留下来，满足别的进程需要。但是算法开销大，每次进程消亡产生新的区域后要重新排序。并且当使用多次后会产生很多外部碎片。

（3）最坏适应算法：将分区从大到小排列(容量递减)，进程每次都找最大的区域来使用。可以减少难以利用的外部碎片。但是大分区很快就被用完了，大的进程可能会有饥饿的现象。算法开销也比较大。

（4）邻近适应算法：空间分区按照地址递增的顺序进行排列，是由首次适应演变而来，进程每次寻找空间，从上一次查找的地址以后开始查找(不同于首次适应，首次适应每次从开头查找)。算法开销小，大的分区也会很快被用完。

2. 非连续分配：

可以将一个进程分散的装入内存分区。根据分区的大小是否固定可以分成分页存储管理(固定)与分段存储管理(不固定)，为了避免两者的缺点，还可以二者混用成段页式存储管理。再根据进程运行作业时是否将作业的的全部代码装入内存，又分为基本分页存储管理(全部装入内存)和请求分页存储管理(非一次全装入内存)。

**基本分页存储：**

把主存空间划分为大小相等的块，块相对较小，作为主存的基本单元。每个进程也以块为单位划分，进程执行时，以块为单位申请内存空间。Linux的分页大小一般是4KB。

**地址结构：**地址结构包含两部分，第一部分是页号(P)，根据页号的位数可以算出地址结构可容纳最大页数；第二部分是页内偏移量(W)，可以计算出页面的大小。地址结构决定了虚拟内存的寻址空间有多大。

**页表：**为了便于在内存中找到进程的每个页面所对应的物理块，系统为每个进程建立一张页表，记录每个页面(进程中的块)在内存中的物理块号，一般放在内存中。页表项由两部分构成，第一部分存储页号，第二部分储存物理内存中的块号。

**基本分段存储：**

分段：按照进程的程序逻辑进行划分，例如进程由主程序、两个子程序、栈和数据组成。于是可以把这个进程分成5段，每一段的逻辑地址从0开始编址，并分配一段连续的内存空间。(注意这里段内必须连续，段与段之间可以分散)

**逻辑地址结构：**是由两部分组成第一部分为段号S，看进程是哪一个段，段号的位数决定了进程分了多少段；第二部分为段内偏移量W，段内偏移量决定了这段进程的最大长度。

**段表：**每个进程都以一张逻辑空间与内存空间映射的段表，每个段表对应进程的一段，段表项即该段在内存中的起始地址和长度。段表由三个部分组成：段号、段长、本段在主存中的起始地址(基址)。

## 连续分配、分页存储、分段存储的优缺点

****连续分配**，设计简单，直接寻址，效率高。缺点：内存利用效率较低，有内部碎片。**

1. **分页和分段两者都采用离散分配方式，且都要通过地址映射机构来实现地址变换。**
2. **页是信息的物理单位，分页是为了实现非连续分配,以便解决内存碎片问题, 提高内存的利用率。段则是信息的逻辑单位，分段的目的是为了能更好地满足用户的需要。**
3. **页的大小固定且由系统决定，由系统把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的；而段的长度却不固定，决定于用户所编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。**

## 分段存储和分页存储的区别：

1、页是信息的物理单位，分页的主要目的是为了实现离散分配，提高内存的利用率。分页仅仅是系统管理上的需求；段是信息的逻辑单位，分段的主要目的是更好地满足用户需求。分段是用户可见的，在编程时需要显示的给出段名。

2、页的大小是固定的，由系统决定；段的大小是不固定的，取决于系统程序。

3、分页存储需要提供页号和页内偏移量表示对应地址，段式存储需要提供段名、段长度和段内偏移量。

4、分段比分页更容易实现信息的共享和保护。注意：不能修改的代码称为纯代码(可重入代码)，这样的代码段不是临界资源，可以共享。可修改的代码是不可以共享的（比如有很多变量的代码段）。

比如：生产者进程的一个进程段，是用来判断该缓冲区此时是否可以访问，这个时候消费者进程的段表项也可以指向这里。

**为什么分页管理不方便实现代码共享？因为将生产者进程分页，由于页面的空间有限，一段逻辑代码可能被装入多个空间，一个空间也可能有多个代码段被装进来。**

5、访问一个逻辑地址需要几次访问内存？ 单级页表：1.查内存中的页表——2.访问目标内存单元 分段：1.查询内存中的段——2.访问目标内存单元 分段与分页系统相似，分段系统也可以引入快表机构，将近期访问过的段表放到快表中，这样可以少一次访问，加快地址变换速度。

6、**分页：内存空间利用率高，不会产生外部碎片，只有少量的内部碎片；不方便按照逻辑模块实现信息的共享与保护**

## 请求分页中页表的内容、、、地址转换过程

页表的内容：

1、页号：页的编号

2、内存块号(物理块号)：储存物理地址内存中的块号

3、状态位P：用来表示该页是否已经调入内存

4、访问字段A：记录页面指标。记录本页在一段时间内被访问的次数，或者记录本页进入内存多长时间未被访问。共页面置换算法换出页面时访问。

5、修改位M：标记本页在调入内存后有没有被修改过，因为修改过的页面是要重新写入内存的，在选择换出时可以选择这种页面。

6、外存地址：用来表示该页在外存中的地址

逻辑地址转换物理地址过程：

1、程序请求访问某页，首先会去判断页号是否大于页表长度，如果大于，产生越界中断，程序结束。

2、检查快表，如果命中则修改页表的访问位和修改位，并形成物理地址，转换结束。

3、如果没有命中快表，就访问页表查询物理地址，如果此时的页在内存中，则修改快表，添加对应项，然后修改访问位和修改位，并形成物理地址，转换结束。

4、如果此时页表不在内存中，则产生缺页中断，请求调页。

5、缺页中断将保留CPU现场，阻塞进程，并从外存中找到所缺的页。

6、判断内存是否已满，如果此时内存满了，则采用页面置换算法换出一页。

7、操作系统读缺页，启动I/O将缺页读入内存，并添加快表。

## 段页式存储特点

为每个作业或者进程建立一张段表以管理内存分配与释放、缺段处理等。另外由于一个段又被划分为若干个页，每个段必须建立一张页表以把段中的虚页变换为内存中的实际页面。段页式管理是段式管理和页式管理相结合而成，具有两者的优点；另外由于管理软件的增加，复杂性和开销也增加。另外需要的硬件以及占用的内存也有所增加，使得执行速度下降。

## 内存分配的伙伴算法

主要是在运行一个程序的时候，系统会分配给其一些内存，但是分配内存的大小是不确定的，无法预估驻留集和工作集的大小，因此可能会发生抖动现象，需要继续进行分配。

（介绍）伙伴算法是一种动态存储器管理算法。该算法通过不断地平分较大的空闲内存块来获得较小的空闲内存块，直到获得所需要的内存块，当内存释放时，该算法尽可能地合并空闲块。

（原理）伙伴算法在分配和回收过程中按照2的次幂（1、2、4、8...）分配内存块给程序。伙伴算法的实现依赖数组和链表，数组的每一位其实是一个链表，链表中挂载的是空闲的内存块大小，数组的索引标识了链表上内存块的大小，例如索引为2的位置上，内存块大小是2的2次幂，即4。当进行内存分配的时候，会找到第一个2的次幂大于请求空间的链表，假设为2的k次幂，如果链表不为空直接分配内存块，如果链表为空，则按照数组往后查找，找到第一个不为空的链表，假设找到了2的k+1次幂，“伙伴“就指的是，将这个内存块一分为二，将一块内存交给用户，另一块内存就挂到2的k次幂的链表上（如果差距过大，这个分给用户的内存块会一直拆分到K次幂大）。”伙伴“需要满足：（1）两个块大小相同（2）两个块地址连续（3）两个块必须是同一个大块中分离出来的

内存回收是内存分配的逆过程，假设以上存储要释放4KB内存，首先到链表中4KB位置查看是否有它的“伙伴”，发现该位置不为空，于是合并成一个8KB的块，继续寻找它的“伙伴”，然后合并成一个16KB的块，插入链表中。

优缺点：

（1）伙伴算法采用2的幂次方进行分配内存块，可以避免把大的内存块拆分的过小，更重要的是可以加快分配和释放速度，但如果所需要的空间不是2的整数次幂，则会产生许多内部碎片。（2）分配和合并采用链表和数组操作，操作方便，但是开销比较大。（3）一个很小的块往往会阻碍一个大块的合并，一个系统中，对内存块的分配，大小是随机的，一片内存中仅一个小的内存块没有释放，旁边两个大的内存块就不能合并。

## 内存的覆盖和交换

覆盖：由于程序在运行的时候并不是任何时候都要访问程序的所有数据和代码，所以可以将用户空间分成一个固定区和若干个覆盖区。将经常使用的程序段放在固定区(不会被调出)。而那些互斥使用的程序可以交替使用覆盖区，如果不使用的话，会被调出内存。

覆盖的特点：程序的层次结构必须由程序员来申明，操作系统完成自动覆盖，缺点对用户不透明，增加用户负担。

交换：把处于等待状态的进程从内存移到辅存，内存空间腾出来，这个叫做换出；把将要调用的进程在从辅存调到内存，这个过程叫做换入。

交换的特点：交换一般是根据交换时长与执行时长比较，前者长的话就不适合；也会根据进程的优先级，首先换出优先级低的进程，但这里要防止低优先级进程饥饿；硬盘其实是分为文件区和交换区的，换出去的进程是存放在交换区。

（覆盖和交换的区别）（1）覆盖是在同一个程序或者进程之间的；（2）交换是在不同进程和程序之间的；（3）所以覆盖技术与交换技术可以一起使用。

## 驻留集的三种分配策略

**固定分配局部置换**：每个进程分配一定的物理块，在整个运行过程中如果发生缺页，只能从这些物理块中去换出，然后再换入

**可变分配全局置换**：为每个进程分配一定数目的物理块，操作系统也会保持出一个空闲的物理块队列。当发生缺页的时候，系统会将空闲的物理块取出给这个进程使用。这个方法是最灵活的，也是最容易实现的。

**可变分配局部置换**：操作系统为每一个进程分配一定数目的物理块，当进程缺页的时候只能从该进程的内部的页面选出一页换出，不会影响其他进程。如果经常出现缺页，系统会为该进程再分配一些物理块供他使用，这里体现了动态性。

## 虚拟内存概念、局部性原理、缺页中断过程

> 概念：基于局部性原理，在程序装入内存时，只需将程序的一部分装入内存，就可以启动程序。在程序执行过程中如果所需要的信息不在内存中，可以由操纵系统将需要的那一部分数据再调入内存。如果操作系统暂时不适用某些内容，可以将其调到外存上，从而腾出空间供别的作业使用。以上就称为虚拟存储器。

> 时间局部性：程序中的一条指令一旦执行，不久后改指令还可能再次被执行。产生时间局部性的原因是程序中存在大量的循环操作。

> 空间局部性：一旦程序访问了某个存储单元，在不久后，其附近的存储单元也会被访问。因为指令的顺序通常是顺序存储、顺序执行的。

> 引入虚拟内存的原因：传统的内存管理方式具有一次性(作业必须一次性的装入内存，其实每次运行的只是一小部分)和驻留性(作业被装入内存后会一直驻留在内存中直到所有的作业结束)。这两个特性导致了内存空间存储效率极低，所以引入了虚拟内存的概念。

> 虚拟内存的特点：

多次性：是指作业在运行的过程中不是一次性全部调入内存，而是分成多次调入内存。

对换性：是指作业在运行过程中不需要一直存放在内存中，需要的作业从外存换入，不需要的可以暂时换出

虚拟性：从逻辑上扩充了内存的容量，使用户看到内存容量很大的程序却很顺利的运行在很小的内存上。

> 虚拟内存的实现方式：

1、请求分页存储管理/请求分段存储管理/请求段页式存储管理

2、硬件支持：（1）一定的内存与外存空间（2）页表机制或者段表机制（3）中断机制，当用户要访问的程序调入内存需要中断（4）地址变换机制，逻辑地址转换成物理地址。

> 缺页中断过程：保护CPU环境、分析中断原因、装入缺页中断处理程序(页面置换算法)、恢复CPU环境

## 使用虚拟地址的好处

1. 读写内存的安全性：避免用户直接访问物理内存地址，防止一些破坏性操作，保护操作系统的安全性。
2. 让每个进程有独立的地址空间（保护进程间的安全）：在某一个进程中访问任何地址都不可能访问到另外一个进程的数据，这样使得任何一个进程如果执行错误指令或恶意代码导致的非法内存访问都不会意外改写其它进程的数据，不会影响其它进程的运行，从而保证整个系统的稳定性。
3. 物理内存映射到虚拟内存会给分配和释放带来很多方便：物理地址不连续的几块内存可以映射成虚拟地址连续的一块内存。比如要用malloc分配一块很大的内存空间，虽然有足够多的空闲物理内存，却没有足够大的连续空闲内存，这时就可以分配多个不连续的物理页面而映射到连续的虚拟地址范围。
4. 一个系统如果同时运行着很多进程，为各进程分配的内存之和可能会大于实际可用的物理内存，虚拟内存管理使得这种情况下各进程仍然能够正常运行。

## 从何处调入页面、调入的时机、抖动现象

从何处调入页面：

请求分页系统的外存其实是分成两个区的，一个文件区(用来存放文件)和对换区(用来存放对换的页面)。对换区通产采用连续分配，文件区采用离散分配，所以对换区的效率更高。 如果系统拥有足够的对换区空间，可以全部从对换区调入内存，以提高调页的速度。这样的话需要在进程运行前将文件区的作业复制到对换区。 如果缺少足够的对换区空间，凡是不会被修改的文件放到文件区，会被修改的页面放到对换区。因为那些不被修改的页面不需要换出。//UNIX方式：与进程有关的页面放在文件区，对换出来的页面放在对换区。

页面调入时机：

为了确定系统进程的运行时所需要的页面调入内存的时机，会有两个策略预调页策略和请求调页策略，一般的系统中会结合使用。 预调页策略：根据局部性原理，一次调入多个页面肯定比一次调入一个页面更高效，但是一次调入太多的页面会浪费内存使用率。所以先预计哪些页面会使用，就先将其调入。这个方法主要用于进程的首次调入，由程序员和用户完成。 请求调页策略：在运行过程中，如果进程需要哪个页面不在内存中，就会提出请求，系统将页面调入内存。这个方式一次只能调入一页，调入调出会加大I/O开销。

抖动现象：

刚刚换出的页面又要换入内存，刚刚换入内存的页面马上又要换出内存。主要原因是某个进程频繁访问的页面数目高于系统给他分配的物理页帧数目。

## SWAP机制及4G内存申请8G

在 32 位操作系统，因为进程最大只能申请 3 GB 大小的虚拟内存，所以直接申请 8G 内存，会申请失败。

在 64 位操作系统，因为进程最大只能申请 128 TB 大小的虚拟内存，即使物理内存只有 4GB，申请 8G 内存也是没问题，因为申请的内存是虚拟内存。

**（1G内存申请2G怎么办）程序申请的虚拟内存，如果没有被使用，它是不会占用物理空间的。当访问这块虚拟内存后，操作系统才会进行物理内存分配。**

**如果申请物理内存大小超过了空闲物理内存大小，如果没有开启 Swap 机制，程序就会直接 OOM；如果有开启 Swap 机制，程序可以正常运行。**

（swap机制）当内存使用存在压力或者内存不够用的时候，会开始触发内存回收行为，会把这些不常访问的内存先写到**磁盘中**，然后释放这些内存，给其他更需要的进程使用。再次访问这些内存时，重新从磁盘读入内存就可以了。这种将内存数据换出磁盘，又从磁盘中恢复数据到内存的过程，就是 Swap 机制负责的。

## 共享/物理/虚拟内存

物理内存：物理内存是在操作系统的内存中分成了许多大小相同的块称为页面，页面根据不同操作系统分得的大小也不同，每个进程保存的逻辑地址会经过MMU转换为物理地址后才会去执行该物理地址上保存的数据。

虚拟内存：虚拟内存是操作系统扩充内存的一种方法，其运用了局部性的原理，每次执行程序时，并不需要将全部的程序装入内存而是装入一部分就开始运行，当所对应的页面不在内存中时会产生缺页中断，根据不同的页面置换算法将页面置换入内存执行，虚拟内存有效地扩充了操作系统的内存大小，能够执行比实际内存更大的程序。

共享内存：由于我们执行程序的时候可能需要很多外部的动态库以及进程通信的一些要求，我们会开辟一块空闲的内存块作为共享内存使用，使用这块内存时不同的逻辑地址会指向一个相同的物理地址去使用，因为这种方法涉及到了多个进程对同一内存块的修改，所以一般配合互斥锁、信号量等方式使用。共享内存和管道、消息队列这样的通信方式相比，由于其是直接对内存进行操作，而其他的通信方式是在内核中的，所以要经过内核进行四次拷贝工作，共享内存效率更高。

## Linux写时复制原理

在 Linux 系统中，调用 fork 系统调用创建子进程时，并不会把父进程所有占用的内存页复制一份，而是与父进程共用相同的内存页，而当子进程或者父进程对内存页进行修改时才会进行复制，这就是写时复制机制。

## 操作系统缓冲原理，cache(高速缓存)

> 使用的原因：由于cpu访问速度与内存访问速度差异较大的原因，为了加快cpu访问主存的速度，在cpu与内存之间加入了cache高速缓存器，以此提高cpu访问内存的速度。

> 程序的执行过程中遵循局部性原理，即时间局部性原理和空间局部性原理。时间局部性是指程序中的一条指令一旦执行，不久后改指令还可能再次被执行。空间局部性指的是一旦程序访问了某个存储单元，在不久后，其附近的存储单元也会被访问

> Cache就是借助局部性原理把主存中被频繁访问的活跃程序块和数据块复制到 cache 中，分为指令cache和数据cache

> 内存放入cache规则：

直接映射（direct）：每个主存块映射到 cache 的固定行中。

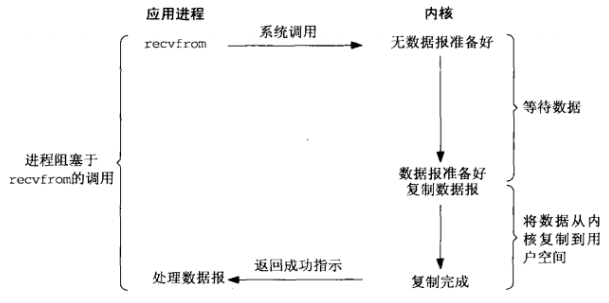
全相连映射（full associate）：每个主存块映射到 cache 的任意行中。

组相连映射（set associate）：每个主存块映射到 cache 的固定组的任意行中

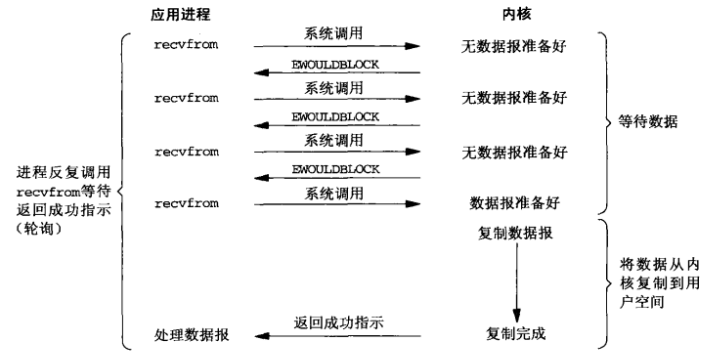
## ----------------------------IO模型及其他--------------------------

## 五种IO模型

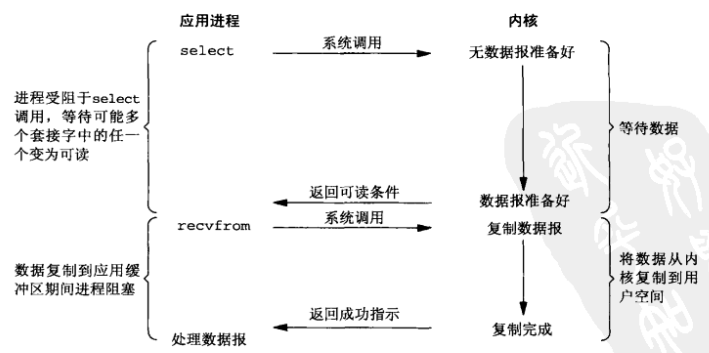
1. **阻塞式IO模型：**应用进程从发起 IO 系统调用，至内核返回成功标识，这整个期间是处于阻塞状态的。



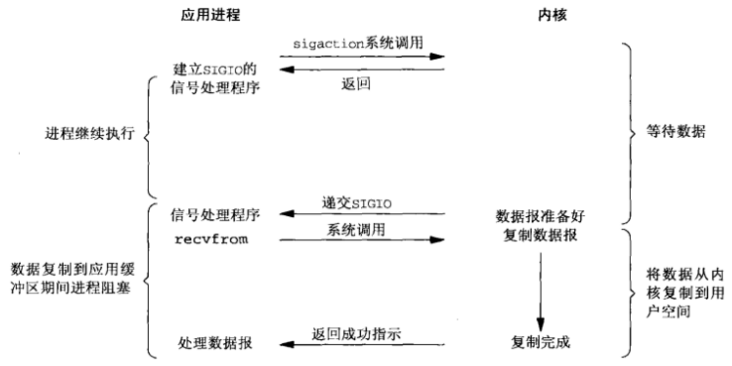
1. **非阻塞式IO模型：**应用进程可以将 Socket 设置为非阻塞，这样应用进程在发起 IO 系统调用后，会立刻返回。应用进程可以轮询的发起 IO 系统调用，直到内核返回成功标识。



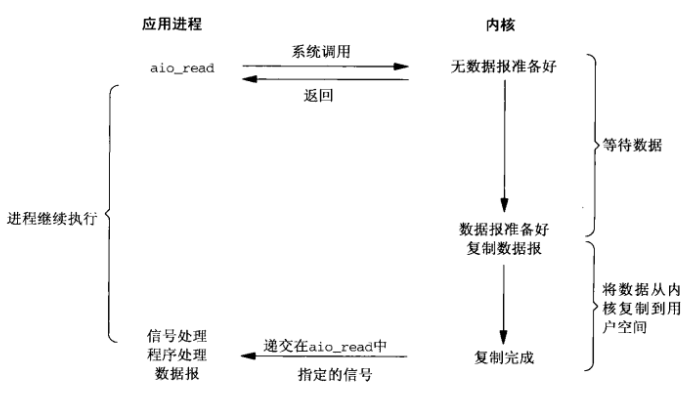
1. **多路IO复用技术：**可以将多个应用进程的 Socket 注册到一个 Select（多路复用器）上，然后使用一个进程来监听该 Select（该操作会阻塞），Select 会监听所有注册进来的 Socket。只要有一个 Socket 的数据准备好，就会返回该Socket。再由应用进程发起 IO 系统调用，来完成数据读取。



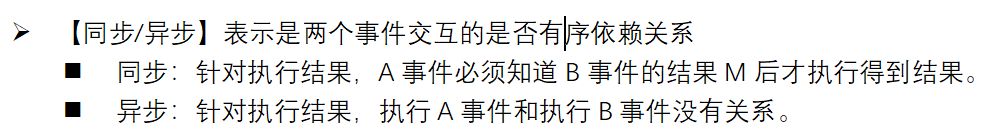
1. **信号驱动IO模型：**可以为 Socket 开启信号驱动 IO 功能，应用进程需向内核注册一个信号处理程序，这个操作会立即返回。然后当内核中有数据准备好，会发送一个信号给应用进程，应用进程便可以在信号处理程序中发起 IO 系统调用，来完成数据读取了。



1. **异步IO模型：**应用进程发起 IO 系统调用后，会立即返回。当内核中数据完全准备后，并且也复制到了用户空间，会产生一个信号来通知应用进程。



## 异步IO



其核心就是两点首先是在不同的线程中执行IO操作，执行完毕后通知管理者接受数据。

1. 注册者申请一个异步IO任务，并且将自己的回调函数发给管理者，当注册者销毁释放的时候，也需要销毁管理者中的回调函数。
2. 当管理者收到异步请求后，创建一个子线程或者从线程池中拿出一个线程处理异步IO任务。
3. 用户进行异步IO操作，操作完成后通知管理者接受数据。
4. 管理者处理封包，通过调用注册者的回调将数据传送下去。

## 多路IO复用技术（同步IO）

1. Select

Select采用了等待队列的数据结构，当新建完多个socket后保存其fd（文件描述符）并放入等待队列，调用select首先会将fd\_set（文件描述符列表）从用户态拷贝到内核态，当有事件就绪或者超时后以轮回的方式查看哪个fd是可操作的，相当于是进行了两次遍历，因此这样会比较浪费时间，默认可监视的数量是1024。**缺点1能够同时监听的用户太少，缺点2以轮回的方式调用太浪费时间，缺点3每次调用select需要将fd集合从用户态拷贝到内核态，开销太大。**

select其实是在**内核**中开辟一块数组空间用来存储用户空间传递的文件描述符进行监控,监控这件事也是内核做的，我们只需要传递文件描述符，**在内核空间进行检测，当有了就绪事件就拷贝到用户空间进行处理。**

1. Poll

Poll和select传参不一样，poll传过去的不是数组结构而是链表结构，打破了select1024的限制。

1. Epoll（1G的内存上能监听约10万个端口）

Epoll在内核里面有一个相应的数据结构去存储数据，这个数据结构就是**eventpoll**，它主要存储两方面的信息：需要监听的fd（文件描述符）集合**（红黑树实现，因为要利于添加和删除，还要便于搜索以免重复添加）**和就绪列表**（双向链表）**用来存放就绪的socket集合。Eventpoll主要采用了回调的方式去代替轮询检查，当在eventpoll中注册一个socket时，会在等待队列保存eventpoll的引用，如果socket读缓存区写入数据后会采用回调的方式将其放入eventpoll中的就绪队列。

所以当调用epoll\_wait检测是否有事件发生时只需要检测eventpoll中的就绪队列是否为空即可。

**叙述阻塞recv下，内核接收数据全过程。**

**如下图所示，进程在recv阻塞期间，计算机收到了对端传送的数据（步骤①）。数据经由网卡传送到内存（步骤②），然后网卡通过中断信号通知cpu有数据到达，cpu执行中断程序（步骤③）。此处的中断程序主要有两项功能，先将网络数据写入到对应socket的接收缓冲区里面（步骤④），再唤醒进程A（步骤⑤），重新将进程A放入工作队列中。**

## Epoll的两个模式水平触发和边沿触发

**水平触发（非阻塞IO浪费资源）**：只要缓冲区有数据，epoll\_wait就会一直被触发，直到缓冲区为空；

**边沿触发（回调形式）**：只有所监听的事件状态改变或者有事件发生时，epoll\_wait才会被触发；

epoll边沿触发时，假设一个客户端发送100字节的数据，而服务器设定read每次读取20字节，那么一次触发只能读取20个字节，

然后内核调用epoll\_wait直到下一次事件发生，才会继续从剩下的80字节读取20个字节，

由此可见，这种模式其工作效率非常低且无法保证数据的完整性，因此边沿触发不会单独使用。边沿触发通常与非阻塞IO一起使用，其工作模式为：epoll\_wait触发一次，在while（1）循环内非阻塞IO读取数据，直到缓冲区数据为空（保证了数据的完整性），内核才会继续调用epoll\_wait等待事件发生。

## Epoll相关函数组成说明

（1）epoll\_create：向内核申请空间，创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大，当创建好epoll句柄后，它就是会占用一个fd值，在linux下如果查看/proc/进程id/fd/，是能够看到这个fd的。

（2）epoll\_ctl：向 epfd 对应的内核epoll 实例添加、修改或删除对 fd 上事件（event）的监听。op操作选项可以为 EPOLL\_CTL\_**ADD**, EPOLL\_CTL\_**MOD**, EPOLL\_CTL\_**DEL**分别对应的是添加新的事件，修改文件描述符上监听的事件类型，从实例上删除一个事件。如果 event 的 events 属性设置了 EPOLLET flag，那么监听该事件的方式是边缘触发。（模板：EPOLL\_CTL\_**ADD|**flag）

（3）epoll\_wait：通过循环，不断地监听暴露的端口，看哪一个fd可读、可写，当 timeout 为 0 时，epoll\_wait 会立即返回。而 timeout 为 -1 时，epoll\_wait 会一直阻塞直到任一已注册的事件变为就绪。当 timeout 为一正整数时，epoll 会阻塞直到计时结束或已注册的事件变为就绪。因为内核调度延迟，阻塞的时间可能会略微超过 timeout （毫秒级）。

epoll文件描述符用完后，直接用close关闭，并且会自动从被侦听的文件描述符集合中删除。

## 操作系统Listen做了那些工作

Listen主要做了两件事情，第一，将套接字文件从主动[文件描述符](https://so.csdn.net/so/search?q=%E6%96%87%E4%BB%B6%E6%8F%8F%E8%BF%B0%E7%AC%A6&spm=1001.2101.3001.7020" \t "_blank)变为被动文件描述符，用于被动监听客户的连接；第二，函数参数backlog 限制全连接队列的大小，还有限制半连接个数。

调用listen函数，会发生如下事情：（1）设置socket的backlog，并创建半连接队列、全连接队列（2）设置TCP状态为TCP\_LISTEN，服务端的socket不能主动连接其他服务，只有socket处于TCP\_LISTEN或者TCP\_CLOSE的时候才能调用listen（3）设定TCP\_LISTEN状态之后还需要校验一次端口，尽管在listen之前bind绑定了端口，但是bind和listen是两个独立的操作，中间的时间段整个系统可能执行了一些影响端口的操作，所以需要一次再校验（4）处于监听状态的socket注册到全局hash表中，32位的哈希桶，用链表的形式串起来包含ip/port的socket信息。

当发生TCP三次握手的时候，会先从哈希桶上找到对应的socket，如果采用了SO\_REUSEPORT，还会再在做一次哈希。

## 惊群效应

多进程/多线程同时监听同一个套接字，当有网络事件发生时，所有等待的进程/线程同时被唤醒，但是只有其中一个进程/线程可以处理该网络事件，其它的进程/线程获取失败重新进入休眠。

惊群问题带来的是 CPU 资源的浪费和锁竞争的开销。根据使用方式的不同，Linux 上的网络惊群问题分为 accept 惊群和 epoll 惊群两种。

## 内核级负载均衡、多个进程监听同一个端口怎么办

为了充分发挥多核 CPU 的性能，多进程的处理网络请求主要有下面两种方式：（1）主进程+多个 worker 子进程监听相同的端口 （2）多进程 + REUSEPORT。

第一种方最常用的一种模式，Nginx 默认就采用这种方式。主进程执行 bind()、listen() 初始化套接字，然后 fork 新的子进程。在这些子进程中，通过 accept/epoll\_wait 同一个套接字来进行请求处理，但是这样的方式容易造成“惊群问题”，惊群问题带来的是 CPU 资源的浪费和锁竞争的开销，使用mutex互斥量可以防止惊群问题。

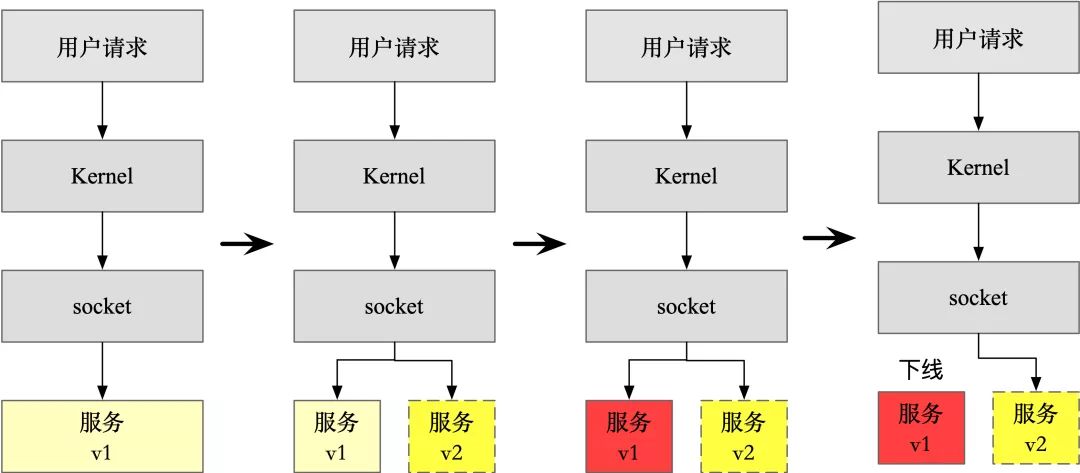
//**计算机中的惊群问题指的是：多进程/多线程同时监听同一个套接字，当有网络事件发生时，所有等待的进程/线程同时被唤醒，但是只有其中一个进程/线程可以处理该网络事件，其它的进程/线程获取失败重新进入休眠。**

对第二种方式来说：内核为处于 LISTEN 状态的 socket 分配了 32 个哈希桶。监听的端口号经过哈希算法运算打散到这些哈希桶中，相同哈希的端口采用拉链法解决冲突。当收到客户端的 SYN 握手报文以后，会根据目标端口号的哈希值计算出哈希冲突链表，然后遍历整条哈希链表得到最匹配的得分最高的 Socket。对于使用 SO\_REUSEPORT 选项的 socket，可能会有多个 socket 得分最高，这个时候经过随机算法选择一个进行处理。这样对于多个socket监听同一个端口来说，冲突太多，即得分高的socket太多查询效率底下，后来引入了SO\_REUSEPORT group的概念，就是监听同一个端口的多个socket有一个专门的数组，当在链表中找到一个socket的时候，直接跳转查询这个数组进行二次哈希选取出对应的socket来处理这次网络数据。  
 //**（SO\_REUSEPORT 与安全性）如果进程监听了某个端口，不怀好意的其他人也可以监听相同的端口来“窃取”流量信息，这种方式被称为端口劫持（port hijacking）。SO\_REUSEPORT 在安全性方面的考虑主要是下面这两点。**

（1）只有第一个启动的进程启用了 SO\_REUSEPORT 选项，后面启动的进程才可以绑定同一个端口。（2）后启动的进程必须与第一个进程的有效用户ID（effective user ID）匹配才可以绑定成功。

（SO\_REUSEPORT）的应用：一是实现了内核级的负载均衡，二是支持滚动升级（见下图）。步骤：（1）新启动一个新版本 v2 ，监听同一个端口，与 v1 旧版本一起处理请求。

（2）发送信号给 v1 版本的进程，让它不再接受新的请求 （3）等待一段时间，等 v1 版本的用户请求都已经处理完毕时，v1 版本的进程退出，留下 v2 版本继续服务。

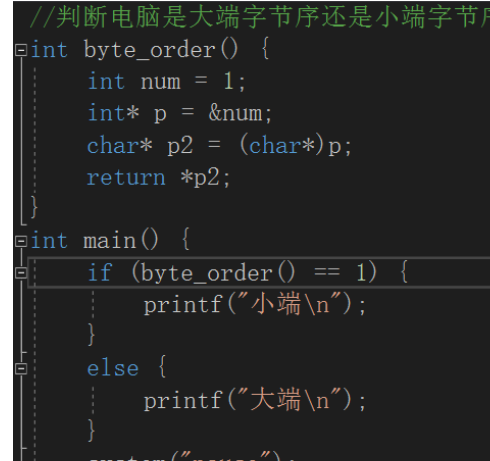


## 大端模式和小端模式及其判断方式

* 大端模式是指数据的低位保存在内存的高地址中，而数据的高位保存在内存的低地址中。
* 小端模式是指数据的低位保存在内存的低地址中，而数据的高位保存在内存的高地址中。

这是根据不同处理器进行决定的。

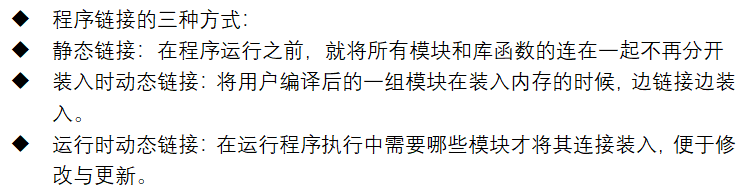
如果是大端字节序，则在内存中存储方式为 0000 0000 0000 0000 0000 0000 0000 0001；如果是小端字节序，则在内存中存储方式为 0000 0001 0000 0000 0000 0000 0000 0000。



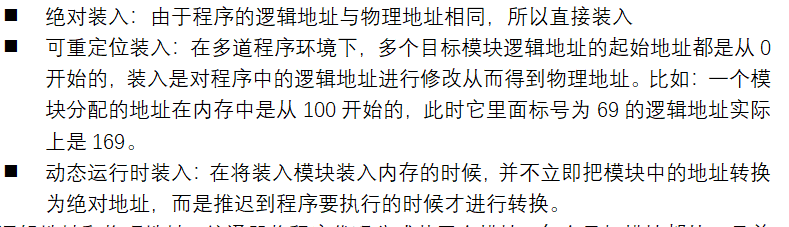
## 程序执行的过程（编译、链接、装入）

编译：由编译程序将用户源代码编译成若干个目标模块。

链接：链接程序将变以后的模块和所需要的库函数连接在一起，形成一个完整的装入模块。



装入：由装入程序将得到的装入模块装入内存运行。



## 零拷贝技术（mmap、sendfile）

当正常进行读操作的时候：

（1）DMA将硬盘上的数据拷贝到内核缓冲区中

（2）CPU将内核缓冲区中的数据拷贝到用户空间进行读操作

当正常进行写操作的时候：

（1）CPU将数据拷贝到socket内核缓冲区

（2）DMA将socket内核缓冲区中的数据拷贝到网卡中进行写操作

所以一次读写共需要两次DMA和两次CPU的拷贝，其中DMA的两次拷贝是没办法省掉的，零拷贝技术就是省掉了CPU的拷贝处理。

现在零拷贝技术主要是使用**mmap和sendfile**，这两个技术一定程度上会减少上下文的切换和CPU的拷贝。

1.使用**mmap是将读缓冲区的地址和用户空间的地址发生映射，实现两个缓冲区共享**，读写过会程被简化为：（省去一次CPU拷贝）

（1）DMA将数据从磁盘拷贝到读内核缓冲区中

（2）CPU将读内核缓冲区中的数据拷贝到socket内核缓冲区中

（3）DMA再将socket内核缓冲区中的数据拷贝到网卡上

2.使用**sendfile的话是将读内核缓冲区的文件描述符和文件大小发给socket内核缓冲区**，实现CPU的零拷贝：

（1）DMA将数据从磁盘拷贝到读内核缓冲区中

（2）CPU把读内核缓冲区的文件描述符和文件大小发送给socket内核缓冲区

（3）DMA根据文件描述符和文件大小从读内核缓冲区中把数据拷贝到网卡以此实现CPU零拷贝

# 计算机网络总结

## 七层、五层、四层

为什么要做分层？

1. 各层次之间是独立的。某一层并不需要知道它的下一层是如何实现的，而仅仅需要知道该层通过层间的接口所提供的服务。
2. 灵活性好。当任何一层发生变化时，只要层间接口关系保持不变，则在这层以上或以下层均不受影响。
3. 易于实现和维护。这种结构使得实现和调试一个庞大又复杂的系统变得易于处理，因为整个的系统已经被分解为若干个相对独立的子系统。
4. 能促进标准化工作。因为每一层的功能及其所提供的服务都已有了精确的说明。

七层：7 应用层 6 表示层 5 会话层 4 传输层 3 网络层 2 数据链路层 1 物理层.

四层：应用层+传输层+网络层+网络接口层

## 各层协议

1. 应用层: 直接为应用进程提供服务(典型设备:应用程序，如DHCP，FTP，SMTP ，HTTP)

DHCP动态主机分配协议，使用 UDP 协议工作，主要有两个用途：给内部网络或网络服务供应商自动分配 IP 地址，给用户或者内部网络管理员作为对所有计算机作中央管理的手段。实 现即插即用连网。

FTP ：文件传输协议<端口号21>减少或消除不同操作系统下处理文件的不兼容性。

HTTP：超文本传输协议 <端口号 80>， 面向事务的应用层协议

SMTP：简单邮件传输协议 <端口号25> 用于发送邮件。

RPC：远程过程调用协议

（2）传输层: 为两台主机中的进程提供端到端的通信(典型设备: 进程和端口) 数据单元：数据段 （Segment）

TCP：传输控制协议提供可靠的面向连接的服务，传输数据前须先建立连接，结束后释放。可靠的全双工信道。可靠、有序、无丢失、不重复。

UDP：用户数据报协议发送数据前无需建立连接，不使用拥塞控制，不保证可靠交付，最大努力交付。

（3）网络层: 为两台主机之间提供服务(典型设备:路由器，防火墙、多层交换机) 数据单元：数据包（Packet ）

IP (IPv4 · IPv6) ：网络之间互连的协议。////\*IP寻址还有动态路由协议，包括RIP协议、OSPF协议、ISIS协议以及BGP协议。\*////

ARP：即地址解析协议，实现通过IP 地址得知其物理地址的功能。

RARP：反向地址转换协议允许局域网的物理机器从网关服务器的ARP表或者缓存上请求其IP地址。

ICMP：控制报文协议。它是TCP/IP 协议族的一个子协议，用于在IP 主机、路由器之间传递控制消息。

IGMP： Internet 组管理协议,是因特网协议家族中的一个组播协议，用于 IP 主机向任一个直接相邻的路由器报告他们的组成员情况。

（4）数据链路层: 对电信号进行封装，定义了分组的方式。数据链路层主要通过校验、确认和反馈重发等方式，将不可靠的物理链路转换为数据链路；并且还需要协调双方的数据传输效率(典型设备: 网卡，网桥，交换机) 数据单元：帧 （Frame）

ARQ：自动重传请求协议，错误纠正协议之一，包括停止等待ARQ 协议和连续ARQ 协议，错误侦测、正面确认、逾时重传与负面确认继以重传等机制。

PPP：点对点协议面向字节，由三部分组成：一个将IP 数据报封装到串行链路的方法；一个用于建立、配置和测试数据链路连接的链路控制协议

CSMA/CD：载波监听多点接入/碰撞检测协议。总线型网络，协议的实质是载波监听和碰撞检测。载波监听即发数据前先检测总线上是否有其他计算机在发送数据，如有暂时不发数据，避免碰撞。碰撞检测为计算机边发送数据边检测信道上的信号电压大小。

（5）物理层:确保数据可以在物理介质上传输，提供稳定可靠的传输环境(典型设备：中继器，集线器、网线、HUB) 协议：以太网协议。

## Socket缓存区（接收缓存区和发送缓存区）

**接收缓冲区**把数据缓存入内核，应用进程一直没有调用recv()进行读取的话，此数据会一直缓存在相应socket的接收缓冲区内。不管进程是否调用recv()读取socket，对端发来的数据都会经由内核接收并且缓存到socket的内核接收缓冲区之中。

进程调用send()发送的数据的时候，将数据拷贝进入socket的内核**发送缓冲区**之中，然后send便会在上层返回。也就是说，send（）返回之时，数据不一定会发送到对端去（和write写文件有点类似）。

一旦将数据成功写入到缓冲区，函数就可以成功返回，不管它们有没有到达目标机器，也不管它们何时被发送到网络，这些都是TCP协议流量控制和拥塞控制要处理的事情。

## 重定位和转发

转发是服务器的行为，它表示服务器会直接访问URL，将目标URL响应的内容读取过来发给浏览器，所以浏览器根本不知道是哪里拿来的内容，因为这个跳转是在服务器实现的，并不是客户端的跳转，所以地址还是那个地址。

重定位是客户端的行为，它表示客户端访问的URL已经失效了，会返回一个新的URL给客户端，客户端再去访问新的URL。

## 数据在网络中的传输方式（单播、组播、广播）

**单播：**主机之间**一对一**的通讯模式，网络中的交换机和路由器对数据只进行转发不进行复制。如果10个客户机需要相同的数据，则服务器需要逐一传送，重复10次相同的工作。但由于其能够针对每个客户的及时响应，所以现在的网页浏览全部都是采用单播模式，具体的说就是IP单播协议。网络中的路由器和交换机根据其目标地址选择传输路径，将IP单播数据传送到其指定的目的地。

**组播：**主机之间**一对一组**的通讯模式，也就是加入了同一个组的主机可以接受到此组内的所有数据，网络中的交换机和路由器只向有需求者复制并转发其所需数据。主机可以向路由器请求加入或退出某个组，网络中的路由器和交换机有选择的复制并传输数据，即只将组内数据传输给那些加入组的主机。这样既能一次将数据传输给多个有需要（加入组）的主机，又能保证不影响其他不需要（未加入组）的主机的其他通讯。

**广播：**主机之间一对所有的通讯模式，网络对其中每一台主机发出的信号都进行无条件复制并转发，所有主机都可以接收到所有信息（不管你是否需要），由于其不用路径选择，所以其网络成本可以很低廉。有线电视网就是典型的广播型网络，我们的电视机实际上是接受到所有频道的信号，但只将一个频道的信号还原成画面。

## 端到端和点到点的区别

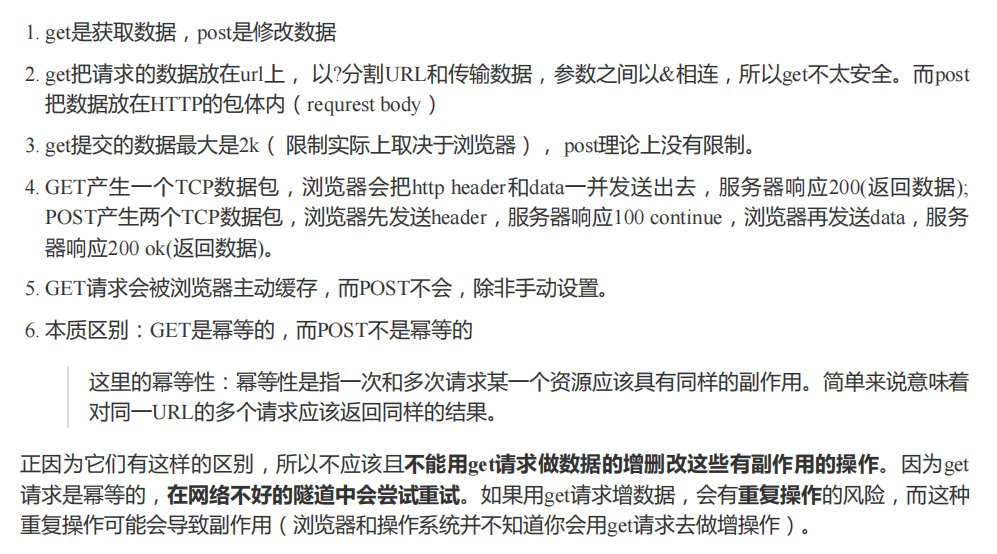
端到端主要涉及传输层之间的通信，传输层为网络提供端到端的通信，点到点主要涉及数据链路层、网络层的通信。

端到端表示在传输层上，会在发送端和接收端之间建立一条链路，然后发送端就能发送消息了，若接收端收到消息则通信完毕。

点到点的传输表示在数据链路层和网络层上，因为数据链路层只负责两个结点之间的通信，一个数据链路层的结点接受ip数据包封装后，就将该数据包发送到与之相邻的下一个结点，然后这个设备在合适的时候又传输给下一个设备，这样就通过一台台相邻的设备发送到了接收端。

## GET 和 POST 是什么？HTTP 协议中的两种发送请求的方法，区别是什么：

都是用的同一个传输层协议：TCP



## -------------------------------HTTP--------------------------------

## http和HTTPSHTTPSHTTPSHTTPS

由于http传输的信息都是未加密的明文传输，所以当传输一些保密信息的时候不是很安全，所以设计了SSL协议对http协议传输的内容进行加密，因此形成了https协议。

**区别：**

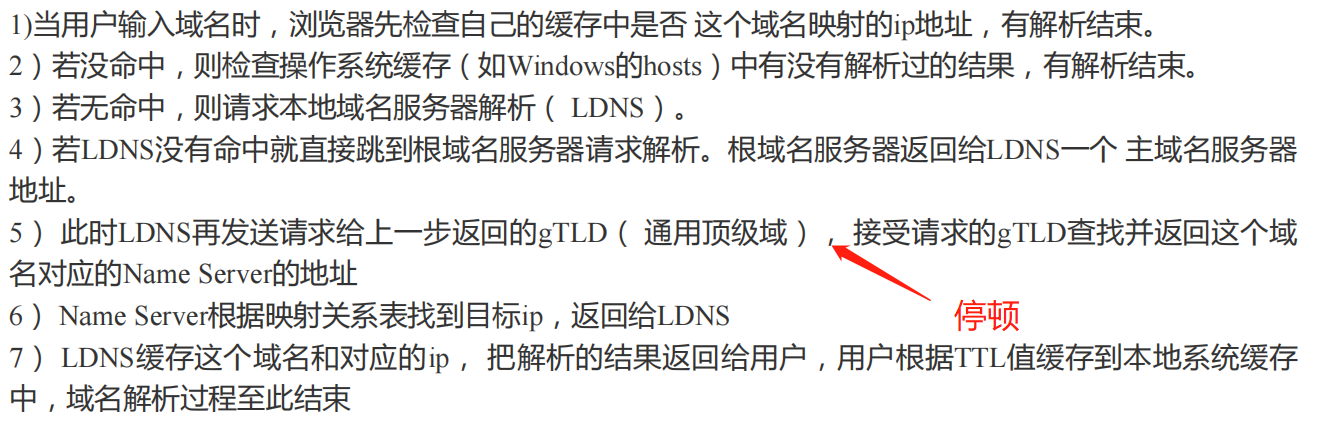
1. https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。
2. http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。
3. http和https使用的是完全不同的连接方式，用的端口也不一样
4. http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

**https加密过程，包括非对称加密和对称加密：**

1. 首先客户端发送自己支持的TLS版本、加密套件以及生成一个第一随机号。
2. 服务器端返回确认支持的TLS版本、加密套件并且生成一个第二随机号发送给客户端。
3. 然后服务器发送证书和公钥给客户端，发送完毕后通知客户端。
4. 客户端生成第三个随机数称为预主密钥，然后使用刚收到的公钥进行加密后发送出去。
5. 服务器收到客户端发送的消息后使用自己的私钥进行解密，这样就收到了预主密钥。
6. 然后双方用第一随机数、第二随机数和预主密钥计算出会话密钥。前面所进行的是非对称加密，最后得到会话密钥后进行对称加密。

## 一次完整的HTTP请求过程包括哪些内容？

1. **浏览器输出一个url，先检查强缓存（本地缓存），如果命中就直接使用，否则进行DNS解析。强缓存也叫本地缓存，强缓存是利用http头中的Expires和Cache-Control两个字段来控制的，用来表示资源的缓存时间。**
2. **首先会对url进行DNS域名解析，得到对应的地址，DNS的过程是：**



1. **根据这个ip，找到对应的服务器，发起TCP三次握手建立浏览器客户端和服务端的连接。**

（TCP：传输控制协议，特点：面向连接、可靠、基于字节流的传输层通信协议）

1. **TCP连接建立后，发送HTTP请求：**

**HTTP请求报文由四部分组成：请求行，请求头、空行、请求正文。**

* 请求行：用于描述客户端的请求方式（GET/POST等），请求的资源名称(URL)以及使用的HTTP协议的版本号
* 请求头：用于描述客户端请求哪台主机及其端口，以及客户端的一些环境信息等
* 空行：空行就是\r\n (POST请求时候有)
* 请求正文：当使用POST等方法时，通常需要客户端向服务器传递数据。这些数据就储存在请求正文中（GET方式是保存在url地址后面，不会放到这里）

1. **服务器响应HTTP请求，浏览器得到html代码:**

**HTTP响应也由三部分组成：状态行，响应头，空格，消息体。**

* 状态行包括：协议版本、状态码、状态码描述
* 其中状态码表示服务器对请求的处理结果，有以下5个大类：
* 1xx：指示信息——表示请求已经接受，继续处理
* 2xx：成功——表示请求已经被成功接收、理解、接受。
* 3xx：重定向——要完成请求必须进行更进一步的操作
* 4xx：客户端错误——请求有语法错误或请求无法实现
* 5xx：服务器端错误——服务器未能实现合法的请求。
* 响应头：响应头用于描述服务器的基本信息，以及客户端如何处理数据。
* 空格：CRLF（即 \r\n）分割。
* 消息体：服务器返回给客户端的数据。

1. **浏览器解析html代码，并请求html代码中的资源:**

浏览器拿到html文件后，就开始解析其中的html代码，遇到js/css/image等静态资源时，就向服务器端去请求下载,这是时候就用上 keep-alive特性了，建立一次HTTP连接，可以请求多个资源，下载资源的顺序就是按照代码里面的顺序，但是由于每个资源大小不一样，而浏览器又是多线程请求请求资源，所以这里显示的顺序并不一定是代码里面的顺序。

1. **浏览器对页面进行渲染呈现给用户**

最后，浏览器利用自己内部的工作机制，把请求的静态资源和html代码进行渲染，渲染之后呈现给用户，浏览器是一个边解析边渲染的过程。

1. **服务器关闭关闭TCP连接**

一般情况下，一旦Web服务器向浏览器发送了请求数据，它就要关闭TCP连接，然后如果浏览器或者服务器在其头信息加入了Connection:keep-alive,TCP连接在发送后将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。

此时就完成了一次完整的HTTP请求。

## http协议的内容（+HTTP状态码）

1. **HTTP请求报文（包括请求行+请求头部+空行+请求数据）80端口**

* **请求行：请求方法、请求地址和协议版本。**
* **请求方法：HTTP/1.1定义的请求方法常用的是POST和GET请求，共8种：GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS、TRACE。**
* **协议版本：常用的是HTTP/1.0和1.1.**

**请求头部：**



1. **HTTP响应报文（状态行+响应头部+空行+响应数据）**

**状态行：**

 1xx：指示信息--表示请求已接收，继续处理。

 2xx：成功--表示请求已被成功接收、理解、接受。

 3xx：重定向--要完成请求必须进行更进一步的操作。

 4xx：客户端错误--请求有语法错误或请求无法实现。

 5xx：服务器端错误--服务器未能实现合法的请求。

**响应头部：**



**详细信息：**

|  |  |  |
| --- | --- | --- |
| 100 | Continue | 客户端应继续请求 |
| 101 | Switching Protocols | 切换协议，服务器根据客户端的请求切换协议。只能切换到更高级的协议，例如，切换到HTTP的新版本协议 |
| 200 | OK | 请求成功。一般用于GET与POST请求 |
| 201 | Created | 已创建。成功请求并创建了新的资源 |
| 202 | Accepted | 已接受。已经接受请求，但未处理完成 |
| 203 | Non-Authoritative Information | 非授权信息。请求成功。但返回的meta信息不在原始的服务器，而是一个副本 |
| 204 | No Content | 无内容。服务器成功处理，但未返回内容。 |
| 205 | Reset Content | 重置内容。服务器处理成功，用户浏览器应重置文档视图。 |
| 206 | Partial Content | 部分内容。服务器成功处理了部分GET请求 |
|  | | |
| 300 | Multiple Choices | 多种选择。请求的资源可包括多个位置，相应可返回一个资源特征与地址的列表用于用户浏览器选择 |
| 301 | Moved Permanently | 永久移动。请求的资源已被永久的移动到新URI，返回信息会包括新的URI，浏览器会自动定向到新URI。今后任何新的请求都应使用新的URI代替 |
| 302 | Found | 临时移动。但资源只是临时被移动。客户端应继续使用原有URI |
| 303 | See Other | 查看其它地址。使用GET和POST请求查看 |
| 304 | Not Modified | 未修改。所请求的资源未修改，服务器返回此状态码时，不会返回任何资源。客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返回在指定日期之后修改的资源 |
| 305 | Use Proxy | 使用代理。所请求的资源必须通过代理访问 |
| 306 | Unused | 已经被废弃的HTTP状态码 |
| 307 | Temporary Redirect | 临时重定向。与302类似。使用GET请求重定向 |
|  | | |
| 400 | Bad Request | 客户端请求的语法错误，服务器无法理解 |
| 401 | Unauthorized | 请求要求用户的身份认证 |
| 402 | Payment Required | 保留，将来使用 |
| 403 | Forbidden | 服务器理解请求客户端的请求，但是拒绝执行此请求 |
| 404 | Not Found | 服务器无法根据客户端的请求找到资源（网页）。 |
| 405 | Method Not Allowed | 客户端请求中的方法被禁止 |
| 406 | Not Acceptable | 服务器无法根据客户端请求的内容特性完成请求 |
| 407 | Proxy Authentication Required | 请求要求代理的身份认证，但请求者应当使用代理进行授权 |
| 408 | Request Time-out | 服务器等待客户端发送的请求时间过长，超时 |
| 409 | Conflict | 服务器完成客户端的 PUT 请求时可能返回此代码，服务器处理请求时发生了冲突 |
| 410 | Gone | 客户端请求的资源已经不存在。 |
| 411 | Length Required | 服务器无法处理客户端发送的不带Content-Length的请求信息 |
| 412 | Precondition Failed | 客户端请求信息的先决条件错误 |
| 413 | Request Entity Too Large | 由于请求的实体过大，服务器无法处理，因此拒绝请求。为防止客户端的连续请求，服务器可能会关闭连接。如果只是服务器暂时无法处理，则会包含一个Retry-After的响应信息 |
| 414 | Request-URI Too Large | 请求的URI过长（URI通常为网址），服务器无法处理 |
| 415 | Unsupported Media Type | 服务器无法处理请求附带的媒体格式 |
| 416 | Requested range not satisfiable | 客户端请求的范围无效 |
| 417 | Expectation Failed | 服务器无法满足Expect的请求头信息 |
|  | | |
| 500 | Internal Server Error | 服务器内部错误，无法完成请求 |
| 501 | Not Implemented | 服务器不支持请求的功能，无法完成请求 |
| 502 | Bad Gateway | 当网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应 |
| 503 | Service Unavailable | 服务器暂时的无法处理客户端的请求，可能是由于超载或系统维护。 |
| 504 | Gateway Time-out | 网关超时，充当网关或代理的服务器，未及时从远端服务器获取请求 |
| 505 | HTTP Version not supported | 服务器不支持请求的HTTP协议的版本，无法完成处理 |

## cookie和session

1. **内容：**

Cookie指的是浏览器cookie，表示服务器发往浏览器并保存在本地的一小块数据，浏览器会在下一次访问同一个服务器时携带上，通常用来告知服务器两个请求是否来自同一个客户端，cookie通常会保存用户登录信息、个性化设置、浏览器记录等信息，cookie使无状态的http协议记录稳定的状态信息成了可能。

Session指的是一次客户端和服务器的会话过程。Session存储指定对象的登录属性等信息，这样当用户在浏览器和应用程序之间切换时，session中的数据就不会丢失，而是在整个用户会话中一直存在下去直到用户关闭客户端。

1. **不同点：**

作用范围不同：session存放在服务器上，cookie存放在客户端上。

存取方式不同：cookie只能存字符串文本，session能存任意数据类型，一般我们在session中会存放常用的变量信息如userid等

有效期不同：cookie是可以设置长期保存的，比如记录用户的默认登录信息，session一般存在时间较短，关闭客户端或者session超时后就会失效。

隐私策略不同：cookie是存在本地的所以有被窃取的风险，session是存放在服务器的，所以相比之下更安全。

存储大小不同：cookie一般在4k以内，而session能存放更多的信息。

1. **为什么需要cookie和session？有什么关联？**

首先因为http协议是无状态的，所以浏览器并不知道是那个用户在访问服务器，所以必须需要一种机制来告诉服务器当前是哪个用户在进行操作，所以此时就需要cookie和session的配合。

当用户第一次访问服务器的时候，服务器会根据请求的用户信息创建一个session，请求返回时将session的唯一标识符sessionid一起返回给浏览器，浏览器将sessionid存入cookie中，同时cookie记录此sessionid属于哪个域名。

当第二次访问服务器的时候，浏览器首先会查询cookie文件下是否有此域名的cookie文件，如果有的话会一起传给浏览器，浏览器再根据接收到的cookie信息提取sessionid，然后查寻该sessionid对应的session信息，如果没有找到就说明登录失效，如果查找到就证明用户已经登录，就可以继续操作了。

由上可知：sessionid是cookie和session之间的桥梁，服务器会根据这两个机制来验证登录信息。

1. **当浏览器禁止cookie的时候，该怎么办？**

第一种办法因为服务器获取的其实是cookie中的sessionid，那么我们可以将每个请求中带一个sessionid的参数，以post方式提交，也可以在请求的地址后面拼接。

第二种办法是使用Token机制，Token机制多用于App客户端和服务器端交互的模式，也可以用于web端做信息管理。Token指的是服务器端生成的一段信息，作为客户端请求的一个标识，token机制和cookie、session机制比较相似。当客户端访问服务器时，服务器产生一个token并返回，当客户端再次访问的时候带上这个token即可。

## 如何考虑session分布式问题？

* **基于Tomcat的Session复制：**就是让这两个服务器之间互相同步session，比如左边服务器之前保存了一个1，右边服务器之前保存了一个2，他们两个一同步，那么左边服务器保存了1,2，右边服务器也保存了1,2。这样做的话，我们无论去哪个服务器，都相当于能拿到全量的session数据。

优点：tomcat原生支持，只需要修改一下配置文件，好多tomcat之间就能复制session

缺点：session同步需要通过网络进行数据传输，就有延迟问题，同时会占用大量带宽，这样会压缩我们整个业务的带宽，会降低我们的处理能力；而且也有可能session过多，一台机器上没办法存下所有session。

总结：如果是大型分布式集群环境，由于所有的web-server都全量保存数据，所以这种方案我们不使用。而如果是小型系统里面，就3/5个tomcat，我们想使用的话，就简单配置一下也还可以。

* **基于Nginx的ip\_hash 负载均衡**：我们可以利用ip的哈希一致性，只要来自于同一个ip的，那我们就永远给它定位到同一个服务器，其实就是对请求过来的ip地址对你的多少台可用的服务器进行取模，然后就会把你的请求通过Nginx的反向代理给分发到对应的服务器上。（这里会把可用的服务器放到一个数组中，如果取模得到的结果是几，就把请求分到服务器数组中的下标为几的服务器上）。
* **Session统一存储（使用Redis类的nosql数据库）：**无论在哪个服务器上，所有的session数据我们统一在后台进行存储，可以利用Redis等nosql数据库进行存储。

**优点：**水平扩展也很容易，无论我们web服务器有多少个，10个，100个，1000个，反正大家都去redis中做存取，即使redis不够用了，我们做redis集群，每个里面存一点，每个里面存一点都能存下来。

我们服务器即使重启、宕机，下次再启动了，我们session也不会丢失，因为session都是redis里面存着，跟我们业务服务器宕机与否没有任何关系。

**缺点：**从内存中取数据是非常快的，也不需要网络交互，而如果我们存储到了redis里面，我们想要从session里面取数据，我们还得连接redis，再来一次网络交互。

## http的跨域问题

HTTP跨域指的是浏览器出于安全考虑，不允许同一个页面向不同源的接口请求数据，这里的不同源指的是两次结构的**协议或主机或端口不同**，浏览器会认为是这是2个不同的服务器，不同的服务器中内容是不可控的，就禁止用户访问了。

HTTP跨域的解决方案

1、CORS：由后端负责解决HTTP的跨域问题，主要是后端的业务在返回结果的时候在HTTP头部加入字段Access-Control-Allow-Origin，告知浏览器不要拦截信息。

2、jsonp，/\*知道名字不了解，涉及到前端的东西\*/

## http断点传输

当上传或者下载的文件中断时，再次连接的时候支持继续传输。http1.1协议开始支持获取文件的部分内容，它通过在header里面的两个参数实现的，客户端请求时对应range，服务器响应content-range，range包括第一个到最后一个字节的大小，content-range包括当前接收范围以及文件总大小，响应完成后返回的响应头内容也不同，返回200表示不使用断点传输方式，返回206表示使用断点传输方式。

## http缓存（强制缓存和对比缓存）

浏览器存在一个缓存数据库，当客户端第一次请求数据的时候，由于缓存数据库中无对应信息，则会访问服务器请求数据，而缓存数据分为两种，一种是强制缓存，一种是对比缓存。

强制缓存指查询缓存时，根据缓存规则检查缓存是否失效，如果未失效则访问数据，如果失效则访问服务器，服务器返回最新的缓存数据和缓存规则存入缓存数据库。

对比缓存是第一次访问缓存数据库得到缓存标识，第二次缓存服务器验证缓存标识是否有效，若有效则获取数据，若无效服务器返回最近数据和缓存规则，然后客户端将其存入缓存数据库。

服务端发送给客户端，文件控制块

对比缓存和强制缓存相比，因为每次传递的是http头部消息，每次不需要传输数据，所以速度会很快。

## 强制缓存和对比缓存的实现

1. 强制缓存的实现通过HTTP协议中的两个字段实现，一个是Expires，另一个是max-age。expires是绝对时间，返回的到期时间是服务器端的时间，这样存在一个问题，如果浏览器所在机器的时间与服务器的时间相差很大，那么误差就很大。max-age是相对时间。强制缓存就是客户端的本地缓存，在客户端第一次请求资源的时候服务器会返回包含Expires和max-age两个字段的资源并缓存在客户端本地，客户端再次请求的时候会校验本地的过期时间，如果资源没有过期则直接使用客户端本地缓存的资源，否则直接请求服务端新资源。
2. 对比缓存的实现是通过response-header中的Last-Modified、request-header中的Last-Modified-Since两个字段以及 ETag服务器返回的资源标识符来实现的。Last-Modified表示资源最后修改的时间（服务器上文件修改的时间），Last-Modified-Since表示上次请求时，资源最后修改的时间（本地文件修改的时间），在实现的时候客户端发送本地文件修改时间，也就是Last-Modified-Since字段，服务器拿这个时间和本地文件的修改时间做比较，如果修改过返回状态码200和资源，如果没有修改则返回状态码304直接使用客户端的本地缓存。Last-Modified/Last-Modified-Since的修改时间只能精确到秒，如果文件在一秒内被修改多次，那么仅靠时间对比无法出区分新旧，所以在http中还用了Etag的资源标识符，同时一些文件会定时刷新，但是内容并不变。使用Last-Modified/Last-Modified-Since判断就无法使用缓存。

## http请求能否通过一tcp连接同时发多个（多路复用的产生原因）

HTTP/1.1 存在一个问题，单个 TCP 连接在同一时刻只能处理一个请求，两个请求的生命周期不能重叠，任意两个 HTTP 请求从开始到结束的时间在同一个 TCP 连接里不能重叠。不可以多个HTTP同时发送的原因是HTTP/1.1是一个文本协议，服务器返回了两个结果，浏览器是没有办法根据响应结果来判断响应对应于哪一个请求的，HTTP/2.0引入了多路复用，可以实现多个HTTP请求同时处理。

## HTTP1.0/1.1/2.0的差别

第一点是1.0使用的是短连接，它表示每获取一次数据就需要重新建立连接然后再断开，这样每次断开、连接.....就太浪费时间了。

HTTP1.1的优化：

* 长连接：如果每次连接都去建立一个新的连接这样消耗时间会很长，因此1.1中使用了长连接，增加了心跳包去检测连接是否通畅，每次传输完成后并没有关闭对应连接而是继续等待后续传输。
* 部分传输：1.1中可以根据范围获取数据，这样就实现了断点传输的功能。
* 缓存处理：引入额外的缓存控制机制。

HTTP传输有大小

HTTP2.0的优化：

在HTTP1.1中由于其解析方式是文本解析，直到读到分隔符才能结束，结束之前无法停止解析，而且服务端无法预知分配的资源数；而2.0中采用了二进制“帧”传输，在传输的包中服务端可以获得流的大小，通过标识Stream ID实现对流的控制，因此可以实现多路复用机制。

* 多路复用：在2.0中通过多路复用技术可以在发送完请求后接着发送后面的请求，只需要将最后的结果进行对应即可。但是多路复用有一个显著的对头阻塞问题，如果对头传输不成功则会阻塞队列中的数据传送，在TCP option里面增加了SACK字段表示了需要重传的包，但是仍无法根本解决队头阻塞问题。
* 头部压缩：在2.0中使用二进制框架对头部信息和数据信息进行了压缩，提高了传输效率。
* 流量控制：在2.0中客户端可以通过发送Window\_update帧来告诉发送方自己想要接受的字节数，从而进行流量控制。
* 服务端推送：当页面还没开始请求的时候，就可以将部分资源推送到客户端了，这样在页面渲染的时候部分资源已经在缓存中，提高了页面打开的速度。

## 可靠udp传输（QUIC协议）

QUIC协议在HTTP2.0的基础上做了一些新的修改：

· **针对队头阻塞问题**，TCP的对头阻塞问题主要是由于数据包超时确认或丢失所以阻塞了当前窗口的滑动，QUIC使用了Packet Number代替了Sequence Number来确认数据的有序到达，Packet Number区别于Sequence Number是严格递增的，相当于就算Packet N数据包丢失了接下来重传的依然是后面的包。

· **数据完整性问题**：QUIC使用Packet Number可以不像TCP那样有序确认，QUIC支持乱序确认，发送方会将没有传输成功的数据包放在待发送队列中，通过Stream ID设置流编号，以及Stream Offset字段设置偏移量组装完整的资源。

· **QUIC的0RTT连接**：在TCP中会通过三次握手确认双方的接收和发送是否正常以及数据开始序列号的确认，由于在第三次握手后不需要回复就能发送消息了，因此首次建立连接会花费1RTT的时间。QUIC采用了TLS简短握手过程，将首次协商的信息记录下来，以Session Ticket的形式传给客户端，如果想恢复之前的连接，可以将Session Ticket发送给服务器，这样就节省了连接建立的RTT开销。

· **降低数据包重传的概率**：通过采用单向递增的Packet Number降低TCP重传的歧义性，从而降低重传概率。

## HTTP和RPC的区别

两者都常用于实现服务，在这个层面最本质的区别是RPC服务主要工作在TCP协议之上（也可以在HTTP协议），而HTTP服务工作在HTTP协议之上。由于HTTP协议基于TCP协议，所以RPC服务天然比HTTP更轻量，效率更胜一筹。

RPC接口即相当于调用本地接口一样调用远程服务的接口，HTTP接口是基于http协议的post接口和get接口（等等，2.0版本协议子支持更多）。

RPC主要用于公司内部服务调用，性能消耗低，传输效率高，服务治理方便。HTTP主要用于对外的异构环境，浏览器调用，APP接口调用，第三方接口调用等等。

## 服务器收不到客户端消息如何排查

1，抓包。分别抓客户端和服务端的网络数据包，看数据是否发送出去或者接收到。（TCP协议是可靠的，不存在丢包问题）

2，如果服务端没有发出响应包，则排查服务端的发包流程是否正常。

3，如果服务端有发出数据包，客户端也有收到，那么排查客户端的收包流程是否正常，特别是粘包，拆包的处理。

## ------------------------HTTPS及公钥密钥-------------------------

## 对称加密和非对称加密

对称加密：指的就是加密和解密使用同一个秘钥，所以叫做对称加密。对称加密只有一个秘钥，作为私钥。

非对称加密指的是：加密和解密使用不同的秘钥，一把作为公开的公钥，另一把作为私钥。公钥加密的信息，只有私钥才能解密。私钥加密的信息，只有公钥才能解密。

非对称加密算法RSA、对称加密算法AES

## 数字签名和数字证书内容（CA被盗/CA伪造）

**什么是数字签名，他的作用是什么？**

* 数字签名就是使用私钥对数据摘要进行签名，并附带和数据一起发送。
* 接收方通过判断发送方的数字签名，可以起到消息防篡改、防伪装、防否认的作用。

**为什么要对数据的摘要进行签名？**

* 数据可能比较大，签名是使用非对称加密算法，比较耗时。
* 防止第三方使用公钥解开签名后，拿到原始数据。

**什么是数字证书，数字证书存在解决了什么问题？**

* 数字证书就是由 CA 机构使用自己私钥，对证书申请者的公钥进行签名认证，能够防止第三方私自伪造公钥。
* 数字证书解决了如何安全分发公钥的问题，也保证了后续加密过程的有效性。

## CA的公钥如何获得

如果用户想得到一份属于自己的证书，他应先向 CA 提出申请。在 CA 判明申请者的身份后，便为他分配一个公钥，并且 CA 将该公钥与申请者的身份信息绑在一起并签字后，便形成证书发给申请者。

如果一个用户想鉴别另一个证书的真伪，他就用 CA 的公钥对那个证书上的签字进行验证，一旦验证通过，该证书就被认为是有效的。

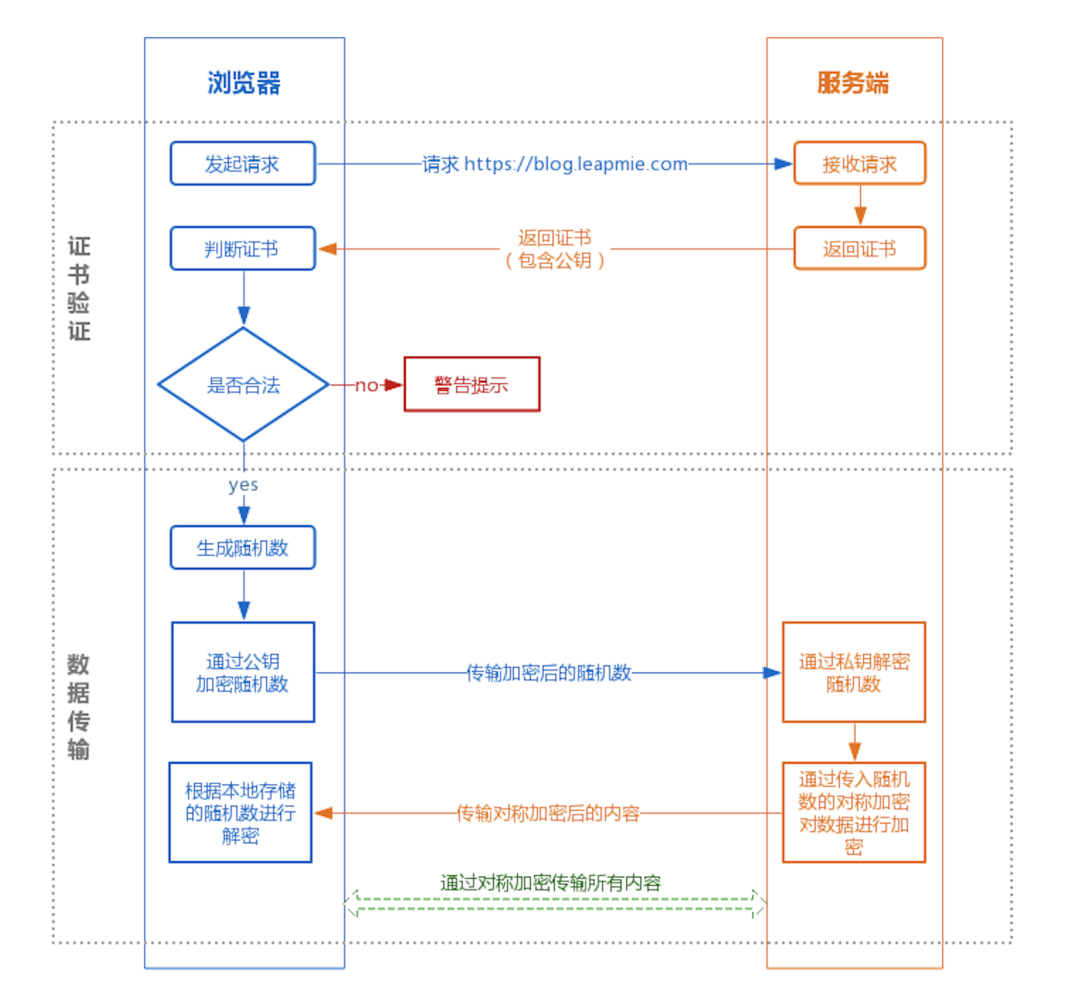
## HTTPS的数据传输过程

HTTPS数据传输过程

① 证书验证阶段：浏览器首先发起 HTTPS 请求，服务端会时首先返回 HTTPS 证书，然后客户端验证证书是否合法，如果不合法则向用户提示告警。

② 数据传输阶段：当证书验证合法后，在本地生成随机数，通过公钥加密随机数，并把加密后的随机数传输到服务端，服务端通过私钥对随机数进行解密

。之后服务端通过客户端传入的随机数构造对称加密算法，对返回结果内容进行加密后传输。



## HTTPS的数据传输为什么是对称加密

非对称加密的加解密效率是非常低的，而 http 的应用场景中通常端与端之间存在大量的交互，非对称加密的效率是无法接受的；另外，在 HTTPS 的场景中只有服务端保存了私钥，一对公私钥只能实现单向的加解密，所以 HTTPS 中内容传输加密采取的是对称加密，而不是非对称加密。

## HTTPS会被抓包吗

HTTPS不能防止抓包的发生。HTTPS 的数据是加密的，常规下抓包工具代理请求后抓到的包内容是加密状态，无法直接查看。但是，只要客户端是我们自己的终端，我们授权的情况下，便可以组建中间人网络，而抓包工具便是作为中间人的代理。通常HTTPS抓包工具的使用方法是会生成一个证书，用户需要手动把证书安装到客户端中，然后终端发起的所有请求通过该证书完成与抓包工具的交互，然后抓包工具再转发请求到服务器，最后把服务器返回的结果在控制台输出后再返回给终端，从而完成整个请求的闭环。

## HTTPS中间人攻击

HTTPs保证安全的方式简单来说就是非对称加密，也就是使用公钥来加密一个随机数内容，并使用本地私钥来进行解密。但是这样的交互过程最大的问题就是，服务器或者说网站的公钥可能被篡改，这就是中间人攻击，就比如**查询请求被中间人拦截，伪造了百度的公钥，换成中间人的公钥**，比如中间人是谷歌，那么就是谷歌的公钥，同时中间人也持有对应的私钥（谷歌的私钥）。之后客户端的一切通信，在中间人（谷歌）那里都会被中间人的私钥（谷歌的私钥）给解开内容，因为中间人通过替换的中间人公钥以及对应的中间人私钥解开了交互密码。

**如何避免？？？**

HTTPS对于获取的公钥的过程做了一个验证，就是证书验证机制。这里的原理简单来说就是，对网站下发的公钥做一次哈希校验，然后与本机预先存储的证书信息做验证，如果一致，说明对方的公钥是可信的。这里的本机预先存储的证书信息来自“证书颁发机构”，保存在操作系统里面。

## 既然 HTTPS 不能防抓包，那 HTTPS 有什么意义？

HTTPS可以防止用户在不知情的情况下通信链路被监听，对于主动授信的抓包操作是不提供防护的，因为这个场景用户是已经对风险知情。要防止被抓包，需要采用应用级的安全防护，例如采用私有的对称加密，同时做好移动端的防反编译加固，防止本地算法被破解。

## HTTPS一定是安全的吗

HTTPS 协议本身到目前为止还是没有任何漏洞的，即使你成功进行中间人攻击，本质上是利用了客户端的漏洞（用户点击继续访问或者被恶意导入伪造的根证书），并不是 HTTPS 不够安全。

## TCP、UDP报头、IP报头、HTTP报头和以太网帧头有哪些内容

**TCP报头：**

源端口号+目的端口号（每个16位）

序列号seq（32位）

确认序列号ack（32位）

数据偏移（32位）表示TCP报文段的数据起始处距离TCP报文段的起始处有多远，其实就是TCP报文段的首部长度

保留位（占6位）保留位今后使用，当前应置为0

6个标识位：URG：标记紧急指针、ACK、PSH：表示紧急传输，不用等缓冲区放满就要发送、RST：异常关闭响应码、SYN：标识连接请求、FIN：表示断开连接请求

窗口大小（16位）：用于控制滑动窗口的大小进行流量控制

校验和（16位）：用于检测数据包数据是否异常等

紧急指针（16位）：表示该数据报的优先级较高，要紧急发送，同时标识位中的PSH=1

**UDP报头：**

源端口号+目的端口号（每个16位）

长度字段（16位）

校验和（16位）

**IP报头**：



**HTTP报头：**

1. **HTTP请求报文（包括请求行+请求头部+空行+请求数据）80端口**

* **请求行：请求方法、请求地址和协议版本。**
* **请求方法：HTTP/1.1定义的请求方法常用的是POST和GET请求，共8种：GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS、TRACE。**
* **协议版本：常用的是HTTP/1.0和1.1.**

**请求头部：**



1. **HTTP响应报文（状态行+响应头部+空行+响应数据）**

**状态行：**

 1xx：指示信息--表示请求已接收，继续处理。

 2xx：成功--表示请求已被成功接收、理解、接受。

 3xx：重定向--要完成请求必须进行更进一步的操作。

 4xx：客户端错误--请求有语法错误或请求无法实现。

 5xx：服务器端错误--服务器未能实现合法的请求。

**响应头部：**



**以太网帧头：（共18位）**

**目的mac、源mac地址（每个6字节）**

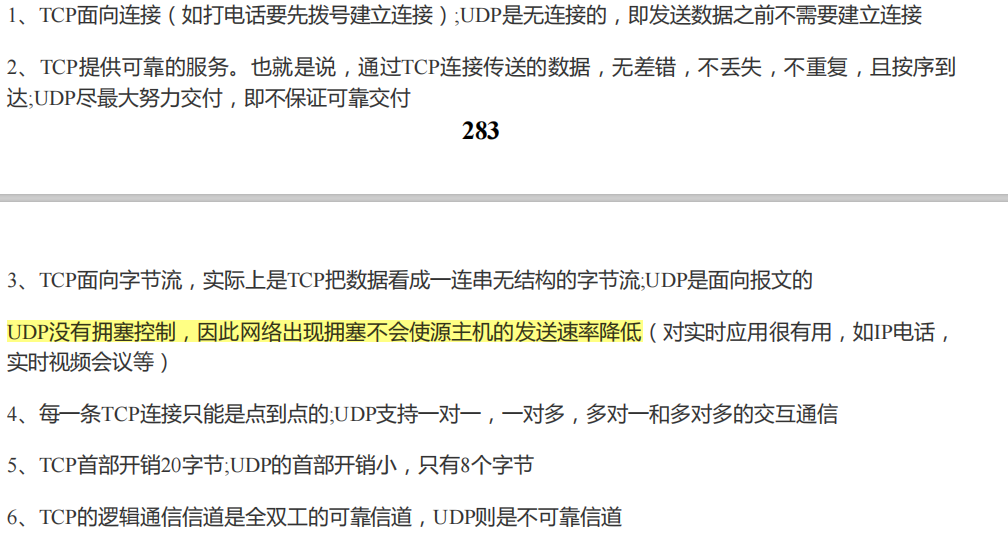
**帧类型（2字节）**

**CRC校验（4字节）**

## -------------------------------TCP---------------------------------

## Tcp和Udp的区别

TCP为通信双方维持一个连接，并且在内核中存储相关数据；UDP无需为数据保存副本，当发送成功后内核缓冲区中的数据报就被丢弃了。





**面向报文的传输方式**是应用层交给UDP多长的报文，UDP就照样发送，即一次发送一个报文。因此，应用程序必须选择合适大小的报文。若报文太长，则IP层需要分片，降低效率。若太短，会是IP太小。UDP对应用层交下来的报文，既不合并，也不拆分，而是保留这些报文的边界。这也就是说，应用层交给UDP多长的报文，UDP就照样发送，即一次发送一个报文。

**面向字节流的话**，虽然应用程序和TCP的交互是一次一个数据块（大小不等），但TCP把应用程序看成是一连串的无结构的字节流。TCP有一个缓冲，当应用程序传送的数据块太长，TCP就可以把它划分短一些再传送。如果应用程序一次只发送一个字节，TCP也可以等待积累有足够多的字节后再构成报文段发送出去。

## UDP一定比TCP快吗，不一定

UDP不一定比TCP快，原因如下：

（1）UDP无法动态调整发包大小，无法有效适应动态变化网络情况，可能导致网络带宽的利用率低下。UDP没有限制发包大小，但是网络传输过程中，以太网协议限制了发送帧的大小，因此如果UDP包太大在IP层会进行大量的分片操作，影响传输的效率，而且其中某个分片丢失了，就会导致整个UDP包的无效。同时如果UDP包太小的话，UDP包中的有效数据占比较小（包头大小和数据的比例），这个时候发送效率低下。所以在动态变化的网络情况中，UDP包过大过小都会导致带宽的利用率低下。而TCP有一套智能算法，会在延迟和吞吐量之间达到一个很好的平衡。

（2）同时因为UDP没有流量控制和拥塞控制机制，网络出现拥塞或通信两端处理能力不匹配的时候，UDP并不会进行调整发送速率，从而导致大量丢包。更加严重的是，UDP的无秩序性和自私性，一个疯狂的UDP程序可能会导致这个网络的拥塞，挤压其他程序的流量带宽，导致所有业务质量都下降。

## 三次握手，四次挥手

**三次握手：**

初始状态：客户端：Closed，服务器端：Listen

第一次：客户端发送请求报文将同步序列号SYN=1及初始化序列号seq=x发送给服务器端，此时客户端处于SYN\_send状态，此时也验证了客户端的发送能力和服务器端的接收能力。

第二次：服务器端收到客户端的请求后，若同意连接，会将自己的同步序列号SYN=1、ACK=1、确认序列号ack=x+1以及初始化序列号seq=y报文发送给客户端，此时服务器端处于SYN\_receive状态，此时客户端知道了服务器端的发送和接收能力，服务器端知道了客户端的发送能力。

第三次：客户端接收到服务器端的SYN+ACK信息后，就知道了下一次可以传输数据序列号了，所以将确认序列号ACK=y+1，数据包序列号=x+1以及确认号ACK=1报文发送给服务器端，此时客户端是Established状态，经过三次握手后，客户端和服务器端都知道了双方的接收和发送能力。

**四次挥手：**

初始状态：双方都是Established状态，双方都可以发起关闭请求，因为TCP是全双工的。

第一次：当客户端发起关闭请求时，会向服务器端发送一个FIN=1，表示客户端不会继续传输数据了，此时客户端进入FIN\_wait1状态。

第二次：当服务器端收到客户端的关闭请求后，会发送一个ACK=1告诉客户端已经收到了客户端的请求，此时服务器端会进入close\_wait状态，继续将剩余的数据发送完毕，客户端收到服务器端的ACK=1消息后，会进入SYN\_wait2状态。

第三次：当服务器中的数据传输完毕之后，服务器会发送FIN=1告诉客户端数据传送完毕了，可以关闭连接了，此时服务器端会进入last\_wait状态。

第四次：客户端收到服务器端的消息后，发送ACK=1给服务器端表示已经收到了服务器端的消息，此时客户端会进入time\_wait状态，经过2MSL时间后会进行断开，而服务器端收到客户端的ACK=1消息后就会立马断开连接。

**Tcp连接过程中有哪些状态？**

Closed、Established、SYN\_wait1、SYN\_wait2、time\_wait、closed\_wait、last\_wait、SYN\_send、SYN\_receive

**什么是半连接队列？**

服务器端第一次收到SYN报文后会处于SYN\_receive状态，此时双方还没完全建立连接，我们会把此状态下的连接请求放到一个队列里，此队列就叫半连接队列。

## TCP为什么不能两次握手

三次握手完成两个重要的功能，既要双方做好发送数据的准备工作(双方都知道彼此已准备好)，也要允许双方就初始序列号进行协商，这个序列号在握手过程中被发送和确认。

（1）三次握手可以防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。

具体例子：已失效的连接请求报文段”的产生在这样一种情况下：client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。

（2）三次握手也可以避免死锁问题。

比如要考虑计算机A和B之间的通信，假定B给A发送一个连接请求分组，A收到了这个分组，并发 送了确认应答分组。按照两次握手的协定，S认为连接已经成功地建立了，可以开始发送数据分组。可是，B在A的应答分组在传输中被丢失的情况下，将不知道A是否已准备好，不知道A建立什么样的序列号，B甚至怀疑A是否收到自己的连接请求分组。在这种情况下，B认为连接还未建立成功，将忽略A发来的任何数据分组，只等待连接确认应答分组。而A在发出的分组超时后，重复发送同样的分组。这样就形成了死锁。

## 为什么要进行三次握手、四次挥手以及2MSL

**为什么要进行三次握手？**

* 通过三次握手才能阻止重复历史连接的初始化（主要）：三次握手在连接时引入了 RST 这一控制消息，接收方当收到请求时会将发送方发来的 SEQ+1 发送给对方，这时由发送方来判断当前连接是否是历史连接，从而判断是否需要中断。
* 通过三次握手才能对通信双方的初始序列号进行初始化，以及对双方接收能力、发送能力进行一个确认。

**为什么要进行四次挥手？**

TCP协议是一种面向连接的、可靠的、基于字节流的运输层通信协议，是全双工模式，这就意味着，当主机1发出FIN报文段时，只是表示主机1已经没有数据要发送了，主机1告诉主机2，它的数据已经全部发送完毕了；但是，这个时候主机1还是可以接受来自主机2的数据；当主机2返回ACK报文段时，表示它已经知道主机1没有数据发送了，但是主机2还是可以发送数据到主机1的；当主机2也发送了FIN报文段时，这个时候就表示主机2也没有数据要发送了，就会告诉主机1，我也没有数据要发送了，之后彼此就会中断这次TCP连接。

**为什么要进行第四次挥手？**

其实当TCP进行连接的时候也是进行了四次握手，但是服务器端发送ACK和SYN信息的时候会一起发送过去，而断开的时候不能一起发送，当发送ACK确认消息后，可能还有未传输完毕的数据要继续传输，当传输全部完成以后才能发送FIN=1消息关闭连接，所以才会进行四次挥手，而且在客户端第四次挥手后还会有2MSL等待时间才算完成断开。

**为什么有2MSL？**

MSL表示的是一个报文的最大生存时间，当客户端发出ACK确认关闭后，无法确定服务器端是否收到了消息，如果服务器端没有收到客户端的ACK报文，那么超时后会重发FIN报文，这样一个FIN报文和一个最后的ACK报文的最长生存时间刚好是2MSL，这样客户端就能保证服务器端收到了自己的ACK报文，还有一个原因就是假如关闭连接后又再次重新建立了新的连接，如果上一个客户端有一个延迟的消息被新的连接接收到了，这样服务器端会认为消息是新的客户端发送过来的，所以要经过2个MSL确保网络中的报文信息全被丢弃了。

## 三次握手和socket编程的对应

服务端首先通过socket建立套接字，然后bind绑定套接字，然后开启listen()监听套接字；客户端发起连接时首先通过socket建立套接字，然后connect初始化套接字，也就是发起了第一次握手，当服务器端收到SYN后，进入SYN\_recv状态，并且将该连接放入连接未完成队列中，返回给客户端一个SYN+ACK，客户端收到ACK后，状态转为establish状态并发送给服务器端一个ACK，服务器端收到ACK后成为establish状态，并且将该连接放入连接已完成队列，然后accept生成连接套接字，然后客户端和服务器端之间就能通过send和recv函数进行消息发送了。

## TCP第三次握手失败了会发生什么情况

当客户端收到服务端的SYNACK应答后，其状态变为establish，并会发送ACK包给服务端，准备发送数据了。如果此时ACK在网络中丢失，过了超时计时器后，那么服务端会重新发送SYNACK包，重传次数能够进行指定(/proc/sys/net/ipv4/tcp\_synack\_retries来指定)，默认是5次。如果重传指定次数到了后，仍然未收到ACK应答，那么一段时间后，Server自动关闭这个连接。

但是此时客户端已经认为连接建立，而服务端则可能处在SYN-RCVD或者CLOSED状态:

**当服务端处于CLOSED状态时**，当接收到连接已经关闭的请求时，服务端会返回RST 报文，客户端接收到后就会关闭连接，如果需要的话则会重连，重新进行三次握手。

**当服务端处于SYN-RCVD状态**，此时如果接收到正常的ACK 报文，连接恢复，继续传输数据；如果接收到写入数据等请求，此时写入数据等请求也是带着ACK 报文的，实际上也能恢复连接，使服务器恢复到establish状态，继续传输数据。

## Time\_wait状态太多怎么回事？

操作系统默认的连接是有上限的，过多连接断开后不释放资源会导致新的连接无法建立，会显示连接被占满了。

出现的主要原因是短连接太多导致的，每次断开连接都需要等待2倍的MSL，因此会存在大量的Time\_wait状态。

服务器可以设置 SO\_REUSEADDR 套接字选项来通知内核，如果端口被占用，但 TCP 连接位于 TIME\_WAIT 状态时可以重用端口。如果你的服务器程序停止后想立即重启，而新的套接字依旧希望使用同一端口，此时 SO\_REUSEADDR 选项就可以避免 TIME-WAIT 状态。

也可以采用长连接的方式减少 TCP 的连接与断开，在长连接的业务中往往不需要考虑 TIME-WAIT 状态，但其实在长连接的业务中并发量一般不会太高。

## Close\_wait状态太多怎么回事？

close\_wait状态是在TCP四次挥手的时候收到FIN但是没有发送自己的FIN时出现的，一般原因有两种：

1.服务器内部业务处理占用了过多时间，都没能**处理完**业务；或者还有数据需要发送；或者服务器的业务逻辑有问题，比如没有执行close()方法；或者是程序Bug。

2.服务器的父进程派生出子进程，子进程继承了socket，收到FIN的时候子进程处理但父进程没有处理该信号，这时候就会导致socket的引用不为0无法回收。

解决方法：第一步停止应用程序，第二步修改程序里的bug。

**如何快速关闭close\_wait状态的连接？**使用gdb关闭，首先使用gdb -p 连接到对应进程中，如何使用call close(27u)，27u就表示close\_wait 状态连接的文件描述符。

## 四次挥手的时候双方同时发出FIN该怎么办

由于 TCP 是双全工的协议，所以是会出现两方同时关闭连接的现象，也就是同时发送了 FIN 报文。两方发送 FIN 报文时，都认为自己是主动方，所以都进入了 FIN\_WAIT1 状态，FIN 报文的重发次数仍由 tcp\_orphan\_retries 参数控制，参数的默认大小是7。

接下来，双方在等待 ACK 报文的过程中，都等来了 FIN 报文。这是一种新情况，所以连接会进入一种叫做 CLOSING 的新状态，它替代了 FIN\_WAIT2 状态。接着，双方内核回复 ACK 确认对方发送通道的关闭后，进入 TIME\_WAIT 状态，等待 2MSL 的时间后，连接自动关闭。

## syn cookie(SYN 注入攻击、SYN 洪泛攻击)

syn cookie：在收到SYN包后，服务器根据一定的方法，以数据包的源地址、端口等信息为参数计算出一个cookie值作为自己的SYNACK包的序列号，回复SYN+ACK后，服务器并不立即分配资源进行处理，等收到发送方的ACK包后，重新根据数据包的源地址、端口计算该包中的确认序列号是否正确，如果正确则建立连接，否则丢弃该包。

## TCP保证数据正确的方法（正确性）

TCP通过**“首部校验”**来保证数据的正确性，但是这个机制并不能保证数据传输过程中一定不会出现差错。

**首部校验的实现**：TCP的首部字段中有一个字段是校验和，发送方将伪首部、TCP首部、TCP数据使用累加和校验的方式计算出一个数字，然后存放在首部的校验和字段里，接收者收到TCP包后重复这个过程，然后将计算出的校验和和接收到的首部中的校验和比较，如果不一致则说明数据在传输过程中出错。

**首部校验存在的问题**：校验的方式是累加和，TCP协议规定的是数据中的每16个比特位数据作为一个数字求和后取末位，但是对于加法来说A+B和B+A的结果是一样的，所以在传输过程中如果有前后两个16比特位的数据前后颠倒了，那么就会出现数据错误的情况，但是接收端无法检查出这样的错误。

**TCP数据正确性保证方法**：最简单的就是使用MD5校验，/\*这也是亚马逊公司的做法\*/，在发送数据前将数据使用MD5加密，并将MD5摘要一起发送，接收端接收数据后将数据再次用MD5加密，如果得到的摘要和收到的摘要一致说明数据正确。

**MD5校验**：MD5消息摘要算法，属Hash算法一类。MD5算法对输入任意长度的消息进行运行，产生一个128位的消息摘要(32位的数字字母混合码)。该算法的主要特点是不可逆，相同数据的MD5值肯定一样，不同数据的MD5值不一样。

## TCP保证数据传输的方法（可靠性）

**检验和、序列号/确认应答、超时重传、最大消息长度、滑动窗口控制等方法。**

**检验和：**通过检验和的方式，TCP可以检测出来数据是否有差错和异常，如果有异常则丢弃并且重新发送，TCP在计算检验和时，会在TCP首部加上一个12字节的伪首部。检验和总共三个部分：TCP伪首部+TCP首部+TCP数据。

**序列号/确认应答：**当通信过程ACK确认包，则会重传数据。

当发送端发出一个包后，接收方会回应一个ACK确认包，如果发送方没收到超时重传：超时重传表示的是数据发送后到接收到确认包之间的RTT往返时间，如果超过了这个时间则是认为丢包了，需要重传。

**最大消息长度：**在建立TCP连接时，双方约定一个最大的长度（MSS）作为发送的单位，重传的时候也是以这个单位来进行重传，理想的情况下该长度的数据刚好不被网络层分块。

**滑动窗口控制**：窗口的大小表示的是在未收到确认包的情况下最多能发送的数据量，这个机制就是使用了大量的缓冲区，通过对多个端进行确认应答的功能。通过下一次的确认包可以判断接收端是否收到了数据，如果已经收到了就冲缓冲区里面删除数据。当发生丢包的时候，发送端会重复收到三次重传ack，此时会进入快速重传算法重传丢失的数据包。

**拥塞窗口：**用了四个拥塞控制算法来获得最佳的传输窗口大小。

## TCP为什么会存在半连接队列

因为根据TCP协议的特点，会存在半连接这样的网络攻击存在，即不停的发SYN包，而从不回应SYN\_ACK。如果发一个SYN包就建立一个消耗极大的socket，那么很容易就内存耗尽。所以内核在三次握手成功之前，只分配一个占用内存极小的request\_sock，以防止这种攻击的现象，再配合syn\_cookie机制，尽量抵御这种半连接攻击的风险。

## TCP连接后拔了网线会怎么样？

TCP 连接在 Linux 内核中是保持了一个socket 的结构体，该结构体的内容包含 TCP 连接的状态等信息，所以当拔掉网线的时候，操作系统并不会变更该结构体的任何内容，所以 TCP 连接的状态也不会发生改变。

而此时应分为拔掉网线后**有数据传输和无数据传输**两种情况。

1. **当拔掉网线有数据传输时：**

如果是客户端被拔掉网线后，服务端向客户端发送的数据报文会得不到任何的响应，在等待一定时长后，服务端就会触发TCP协议的超时重传机制，但是此时重传并不能得到响应的数据报文。

如果在服务端重传报文的过程中，客户端**恰好把网线插回去了**，由于拔掉网线并不会改变客户端的 TCP 连接状态，并且还是处于 ESTABLISHED 状态，所以这时客户端是可以正常接收服务端发来的数据报文的，然后客户端就会回 ACK 响应报文。

而如果**客户端一直没有将网线插回去**，那么服务端超时重传报文的次数达到一定阈值后（默认15次，第一次2s，每次两倍），内核就会判定出该 TCP 有问题。然后就会通过 Socket 接口告诉应用程序该 TCP 连接出问题了，于是服务端的 TCP 连接就会断开。之后接如果客户端再插回网线并向服务端发送了数据后，由于服务端已经没有与客户端匹配的 TCP 连接信息了，因此服务端内核就会回复 RST 报文，客户端收到后就会释放该 TCP 连接。

1. **当拔掉网线没有数据传输时：**

* 如果没有开启 TCP KeepAlive 机制：在客户端拔掉网线后，并且双方都没有进行数据传输，那么客户端和服务端的 TCP 连接将会一直保持存在。
* 如果开启了 TCP KeepAlive 机制：在客户端拔掉网线后，即使双方都没有进行数据传输，在持续一段时间后，TCP 就会发送KeepAlive探测报文，此时服务端依然会将该TCP连接断开。

## 在客户端宕机或者杀死进程会怎么样？

**客户端宕机：**客户端宕机跟拔掉网线是一样无法被服务端的感知的，所以如果在没有数据传输，并且没有开启 TCP keepalive 机制时，服务端的 TCP 连接将会一直处于 ESTABLISHED 连接状态，直到服务端重启进程。

**客户端杀死进程时**：在客户端杀掉进程的话，就会发送 FIN 报文，来断开这个客户端进程与服务端建立的TCP 连接，这种方式影响范围只有这个客户端进程所建立的连接，而其他客户端或进程不会受影响。

## 服务器端宕机或者杀死进程会怎么样？

**服务器宕机：**服务器宕机后如果无数据传输客户端是无法立马感知的，依然是establish状态，如果双方都开启了keep-alive则会在到期后感知并断开TCP连接；如果有数据传输则会经过规定次数的超时重传后，发现无法传输数据，因此也会断开TCP连接。

**服务器杀死进程时**：可客户端杀死进程不一样，服务器端杀死进程会发出FIN报文关闭其所有的TCP连接。

## 服务器端如何正确地关闭一个TCP连接

如果杀死服务器端进程的话影响会比较大，因为此时所有的TCP连接都会被关闭，服务器端无法继续提供服务，所以要精准的关闭某个连接。

正确的方法是可以伪造一个四元组相同（源目IP，源目端口号）的RST报文，而且要保证RST报文序列号要被对方接收，因此可以首先伪造一个四元组相同的 SYN 报文，来拿到“合法”的序列号，因为如果处于 establish 状态的服务端，收到四元组相同的 SYN 报文后，会回复一个 ACK，这个 ACK 报文里的「确认号」，正好是服务端下一次想要接收的序列号，其实就是可以通过这一步拿到服务端下一次预期接收的序列号，然后用这个确认号作为 RST 报文的序列号，发送给服务端，此时服务端会认为这个 RST 报文里的序列号是合法的，于是就会释放连接!

在 Linux 上有个叫 **killcx 的工具**，就是基于上面这样的方式实现的。

## 当连接一个ip不存在的主机会怎么样？或者ip存在，端口号不存在会怎么样？

当TCP进行三次握手时，首先要将一个tcp报头syn位置为1，设置原端口号和目的端口号，此报文进入网络层后，使用ip头进行封包，然后会调用ARP协议获得此ip地址的mac地址，此时会分为局域网和非局域网两种情况。

1. 当目的ip在局域网内时：

首先查询ARP表无对应mac地址，没有的话会发送一个ARP广播来获得对应的mac地址，但是由于目的ip地址不存在，所以ARP广播无法得到相应的mac地址，所以tcp的第一次握手就会卡在ARP协议获取mac地址的阶段。由于客户端没有得到ack确认，所以会不断重发，但是每次重发都会卡在ARP协议获取mac地址的阶段，所以第一次握手的信息一直发不出去。

1. 当目的ip不在局域网内时：

ARP协议会获得默认网关的mac地址，然后数据链路层加入帧头，消息通过网卡发送到网关路由器上，消息会通过互联网一直传到目的ip地址的局域网上，然后也会因为找不到mac地址卡住，客户端依然收不到ack确认，经过一段时间后还会重传。

1. 当ip存在端口不存在时：

由于目的ip地址存在，第一次握手就能顺利达到目的主机上，但是当进入传输层时，会因为目的端口不存在，服务器端会把此数据丢弃，回应一个RST消息（异常关闭响应码）给客户端。

## 服务器收不到客户端消息如何排查

1，抓包。分别抓客户端和服务端的网络数据包，看数据是否发送出去或者接收到。（TCP协议是可靠的，不存在丢包问题）

2，如果服务端没有发出响应包，则排查服务端的发包流程是否正常。

3，如果服务端有发出数据包，客户端也有收到，那么排查客户端的收包流程是否正常，特别是粘包，拆包的处理。

## TCP超时重传的时间计算

TCP采用了一种自适应算法。在TCP协议中，一个报文从发出到收到相应确认的时间之差叫做RTT，RTT的值会随着网络的状况而动态的变化，所以可以计算得到一个平均的加权往返时间，RTTS，按照权重×旧的RTTS+权重×新的RTT求出，这里面的权重记得好像是旧的加权往返时间为八分之七，新的RTT为八分之一。在计算得到加权往返时间之后还需要计算一个超时重传的偏执量，RTT的差值的加权平均值，计算方式也和加权往返时间是一样的，最后的超时重传时间其实就是加权平均往返时间+四倍的差值的加权结果。

## TCP的心跳检测

TCP的KeepAlive保活机制：

1.客户端每隔一个时间间隔发一个探测包给服务器，

2.客户端发包时启动一个超时定时器，

3.服务器端接收到检测包，回应一个包，

4.如果客户机收到服务器的应答包，则说明服务器正常，删除超时定时器，

5.如果客户端的超时定时器超时，依然没有收到应答包，则说明服务器挂了。

KeepAlive超时需要2小时，每次间隔是75秒，探测次数为9次。对于很多服务端应用程序来说，2小时的空闲时间太长。因此，我们需要手工开启KeepAlive功能并设置合理的KeepAlive参数。

## TCP超时重传机制

**超时重传（根据超时周期进行重传）**

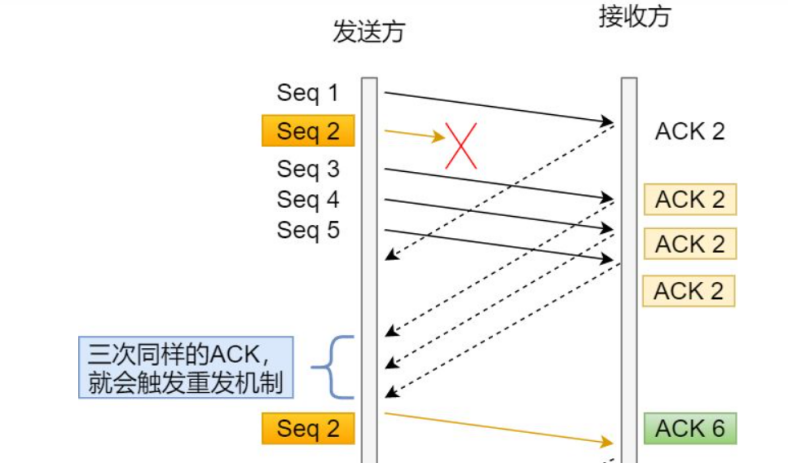
* TCP会在**数据包丢失或者确认应答丢失**的时候进行超时重传，这里的超时时间RTO设置为比RTT大一点的时间间隔（RTO会随着网络的变化而不停变化的）
* 当遇到一次超时重传时，会将**下次的超时时间间隔设置为原先的两倍**，多次超时说明网络环境较差，不宜频繁发送。

**快速重传（根据重复收到的ack进行重传）**

* 当收到三个相同的ACK的时候，则会立即触发重传机制。但是重传对应缺失的数据包还是重传所有数据包呢？因此有了SACK方法来进行控制。

**SACK方法**

* 这种方式需要在TCP头部【选项字段】中增加SACK的东西，它可以将缓存的地图发送给发送方，这样就可以只重传丢失的数据。



1. **SACK**

* 使用SACK告诉发送方有哪些数据被重复接收了。

## TCP粘包的原因

**发送端原因:** 由于TCP协议本身的机制（面向连接的可靠地协议-三次握手机制）客户端与服务器会**维持一个连接**（Channel），数据在连接不断开的情况下，可以**持续不断地将多个数据包发往服务器**，但是如果发送的网络数据包太小，那么TCP会启用Nagle算法（可配置是否启用）**对较小的数据包进行合并**（基于此，TCP的网络延迟要UDP的高些）然后再发送（超时或者包大小足够）。那么这样的话，服务器在接收到消息（数据流）的时候就**无法区分哪些数据包是客户端自己分开发送的**，这样产生了粘包。

**接收端原因:** 服务器在接收到数据库后，**放到缓冲区**中，如果**消息没有被及时从缓存区取走**，下次在取数据的时候可能就会出现**一次取出多个数据包**的情况，造成粘包现象。

## TCP粘包的解决办法

* 固定缓冲区大小：只需要控制服务器端和客户端发送和接收字节的（数组）长度相同即可。
* 可以采用结尾符标记，例如可以使用“\r\n”为结尾，直至获取到结尾标记再进行转换，未获取结尾标记时则需要继续读取。
* 发送端给每个数据包添加包首部，首部中应该至少包含数据包的长度，这样接收端在接收到数据后，通过读取包首部的长度字段，便知道每一个数据包的实际长度了。

## 拥塞控制和流量控制的区别

**流量控制（滑动窗口）**：主要处理的是**端到端之间传输速度快慢的问题**，如果发送方传输数据速度太快，会导致接收方没办法接收，所以要去限制发送端发送数据的速率，一般根据接收方的窗口大小来对传输数据进行控制。

**拥塞控制**：**拥塞控制涉及到网络全局的控制**，包括路由器、交换机以及涉及到网络传输效率的设备，表示的是AB两台机器之间网络发生拥塞导致传输快慢或者丢包，所以要防止过多的数据注入到网络中导致传输速率下降和丢包问题严重，我们必须要进行拥塞控制，拥塞控制我们主要有四个算法来进行控制。拥塞控制是全局性的，和网络中的所有主机、路由器以及与降低网络速率有关的设备有关。

## 拥塞控制算法

**拥塞控制：慢启动算法、拥塞避免算法、快速重传算法、快速恢复算法**

1. **慢启动算法（当发送方每收到一个ACK，拥塞窗口的大小就会加一）**

* 连接建立后，初始化窗口大小为1，表示可以传一个MSS大小的数据。
* 当收到一个ACK后，窗口大小+1，表示可以发送2个数据。
* 接着收到2个ACK确认后，窗口大小+2，因此**窗口大小在慢启动阶段是以指数形式上升。**
* 当窗口大小达到慢启动门限时，就会采用拥塞避免算法。

1. **拥塞避免算法（每收到一个ACK，窗口大小增加窗口大小分之1）：**

* 因此当达到慢启动门限时，此时每个ACK增加窗口大小分之1，全部ACK加起来使窗口大小增加1，因此**此时成为了线性增长。**
* 在增长了一段时间后，网络就会慢慢进入拥塞的情况了，于是出现了丢包现象，因此就会进入拥塞发生算法。

1. **拥塞状态发生时的算法：**

* 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。对于丢包有两种判定方式，一种是超时重传RTO超时，另一个是收到三个重复确认ACK。
* **超时重传**：超时重传RTO超时，TCP会重传数据包。由于发生丢包，将慢启动阈值ssthresh设置为当前窗口大小的一半，窗口大小重置为1，进入慢启动过程。

1. **快速恢复**

* 快速恢复算法一般是和快速重传算法一起使用的，快速重传算法表示当收到三个重复ACK后，将当前窗口大小降低为原来的一半，然后令慢启动门限设置为当前的窗口大小，然后进入**快速恢复算法**。
* 将当前的窗口大小+3，重传丢失的数据包。
* 重传丢失的数据包。
* 如果再收到重复的ACK，将窗口大小增加1。
* 如果收到新数据的ACK后，将窗口大小设置为慢启动门限大小，从而进入拥塞避免算法。

## ------------------------------协议内容-----------------------------

## Nginx常用的几种反向代理策略

（1）轮询（默认），请求依次轮流往每个应用服务器上进行分配，分配策略比较简单。缺点：不均匀，可能会出现，某些服务器接受的请求较重，负载压力重，有些负荷小，不可控。另外服务器之间需要进行session同步。

（2）权重轮询（权重越高，进入的几率越大）。优点：可以根据情况进行调整。可控，仍然需要进行session同步。

（3）IP-hash。优点：无需进行session同步，固定IP会固定访问一台服务器。缺点：恶意攻击，会造成某台服务器压垮。提供的服务不同，面向的地区不同，IP可能会出现集中，造成不均匀，不可控。

（4）URL-Hash。根据URL进行hash，这样某些请求永远打某台服务器。利于利用服务器的缓存，但是可能由于URL的哈希值分布不均匀，以及业务侧重造成某些服务器压力大，某些负荷低。这种也需要进行session同步。

## 为什么要在传输层/网络层进行分片（MSS和MTU）

**MSS就是TCP数据包每次能够传输的最大数据分段，MTU就是网络的最大传输单元（以太网）。**在网络物理链路中，从一个节点向另一个节点传送的是比特流。但是单纯的电信号0和1没有任何意义，必须规定电信号多少位一组，每组什么意思，同时数据传输过程中还需要协调收发双方的数据传输速率，就需要控制发送的大小和方式。因此在数据链路层就出现了以太网协议，规定了电信号的组包方式，称为一个数据帧，以太网协议规定了数据帧的大小不能超过1518字节，其中以太网协议头部标识收发端口占18字节，剩下的1500字节就是MSS，网络的最大传输单元，考虑到传输限制，数据链路层其实就限制了IP包的大小，即IP包最大1500字节，去掉头部就是1480字节，同样的IP数据包的大小也限制了TCP的传输单元大小，这个大小就是MTU，去掉头部字段20字节，最大是1460字节。

## MTU和MSS解释及其联系

MTU：最大传输单元，应用于五层模型的第二层数据链接层，是由硬件规定，如以太网的MTU为1500字节。

MSS：最大报文段长度，表示为TCP数据包每次传输的最大数据分段大小，会在SYN报文中使用MSS选项功能，协商交互双方能够接收的最大段长MSS值，一般由发送端向对端TCP通知对端在每个分节中能发送的最大TCP数据。MSS值为MTU值减去IPv4 Header（20 Byte）和TCP header（20 Byte）得到。

最大报文段长度（MSS）与 最大传输单元（MTU）均是协议用来定义最大长度的。其中MTU应用于数据链接层，并无具体针对的协议。MTU限制了数据链接层上可以传输的数据包的大小，也因此限制了上层（网络层）的数据包大小。例如，如果已知某局域网的MTU为1500字节，则在网络层的 IP协议中最大的数据包大小为1500字节（包含IP协议头）。MSS针对的是传输层的TCP协议，因为传输层在数据链接层的上层，MSS会受到MTU的限制。

## 交换机和路由器有什么区别

交换机工作在数据链路层，路由器工作在网络层。

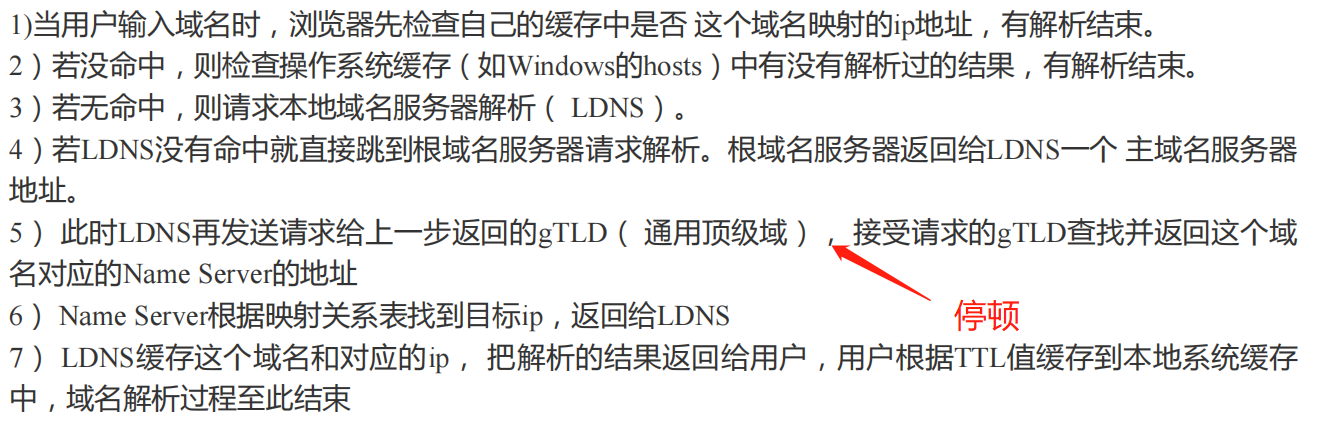
交换机通过MAC地址寻址，路由器通过IP地址寻址。

交换机可以把多个主机连接起来，每个主机都有自己的ip地址，路由器是一个ip分给多个主机用，这些主机对外只表现出一个ip。

## 有了mac地址为什么还需要IP地址

世界上的每个计算机都有其唯一的mac地址，当采用广播的方式发包时，如果都采用以太网的广播形式，那么一台机器发送的包全世界都能收到，这是显然不可以发生的。因此采用了IP地址来区分不同的子网，子网之间采用路径选择算法（路由）进行传递。

## DNS协议（端口53）



DNS占用53号端口，**同时使用TCP和UDP协议**。DNS在区域传输的时候使用TCP协议，其他时候使用UDP协议。

辅域名服务器会定时（一般3小时）向主域名服务器进行查询以便了解数据是否有变动。如有变动，会执行一次区域传送，进行数据同步。区域传送使用TCP而不是UDP，因为数据同步传送的数据量比一个请求应答的数据量要多得多，要保证可靠。

域名解析时使用UDP协议，客户端向DNS服务器查询域名，一般返回的内容都不超过512字节，用UDP传输即可。不用经过三次握手，这样DNS服务器负载更低，响应更快。

**DNS刷新缓存命令：**

1. Windows下使用ipconfig/flushdns。
2. Ubuntu下可以使用nscd，sudo /etc/init.d/nscd restart。

**DNS优化：**

1. dns-prefetch(DNS预获取)是前端网络性能优化的一种措施。它根据浏览器定义的规则，提前解析之后可能会用到的域名，使解析结果缓存到系统缓存中，缩短DNS解析时间，进而提高网站的访问速度。缺点：过多的预获取会导致过量的DNS解析，对网络是一种负担。
2. 可以延长浏览器缓存时间（chrome默认1分钟）和系统缓存时间（默认1小时）。

**DNS劫持：**

DNS劫持指的是攻击者利用其他攻击手段，篡改了某个域名的解析结果，使得指向该域名的IP变成了另一个IP，导致对相应网址的访问被劫持到另一个不可达的或者假冒的网址，从而实现非法窃取用户信息或者破坏正常网络服务的目的。

**DNS劫持发生的情况：**

1. 中间人攻击，拦截用户和DNS服务器之间的通信并提供不同的目标IP地址，从而将用户重定向到恶意站点。
2. 攻击者破解了对应的DNS服务器，更改DNS记录以将DNS请求重定向到恶意站点。
3. 攻击者在用户系统上植入恶意软件并修改本地DNS设置，因此用户的系统现在使用由攻击者控制的DNS服务器。攻击者控制的DNS服务器将网站域请求转换为恶意站点的IP地址，从而将用户重定向到恶意站点。
4. 对路由器做出更改，引导恶意的寻址来影响所有连接的用户。

## Arp和rarp：地址解析协议和反地址解析协议

Arp：主机发送信息时，将包含目标ip地址的arp请求广播到网络上的所有主机，并接收返回消息，以此确定目标主机的物理地址，然后将ip地址和物理地址存入本地的arp缓存中并保留一段时间，下次查询时会先检测本地缓存。

Rarp：反地址解析协议：首先主机从网卡上读取物理地址，然后在网络中发送一个rarp请求的广播数据包，向rarp服务器发出请求。Rarp服务器接收请求为该主机申请ip并返回rarp回应，主机得到回应后使用此ip地址通信。

## ARQ协议（滑动窗口）

它是在数据链路层和物理层上的错误纠正协议之一，他通过确认和超时两个机制在不可靠的服务基础上实现可靠的信息传输，该协议在发送方发送数据后一段时间内没有收到确认的话，它通常会重新发送。ARQ包括停止等待ARQ协议和连续ARQ协议（累计确认），拥有错误检测、超时重传、正面确认和负面确认及确认的机制。

## DHCP协议，为什么需要子网掩码？

子网就是一个局域网。子网掩码就是用来划分子网的，网络位可以向主机位借位以达到划分更小粒度子网的作用，不同的IP地址可以容纳的主机个数是不一致的，A类地址可以容纳一亿多个主机，B类地址可以容纳6万多个主机，C类地址可以254容纳254个主机，在子网中，也就是一个局域网中，计算机通信的基本靠吼，也就是广播的方式，假设现在是B类网络地址，有六万多个主机，靠广播的范式，6万多个主机必须被动听着，很容易就造成网络瘫痪，这个就是广播风暴，同时如果不进行子网切分，也容易造成网络地址的浪费，每个子网有一个网关，网关用来标识子网处于的网段，也就是网络位，判断是不是同一个子网，进而转发IP包。

**DHCP(Dynamic Host Configuration Protocol)中文全称为动态主机配置协议**。它是一个局域网的网络协议，它的主要作用就是管理和分配IP地址，有三种分配方式：

（1）自动分配方式：DHCP服务器可以为主机分配一个永久性的IP地址。当DHCP客户端第一次从DHCP服务端租用到一个IP地址时，就可以永久性的使用该IP地址。

（2）动态分配方式：DHCP服务器为主机分配一个具有时间限制的IP地址，当时间达到限制或者主机主动放弃该地址时，该地址才能被其他主机使用。

（3）手动分配方式：客户端的IP地址由网络管理员指定，DHCP服务器只是将一个指定的IP地址告诉主机。

## IPV4不够用怎么办？（NAT和NAPT）

**NAT网络地址转换是一种网络地址翻译技术，主要是将内部的私有IP地址转换为能在公网使用的IP。**此功能常通过**路由器或者防火墙**来实现。

由于IPV4逐渐不够用，所以NAT让一群机器共用一个IP，暂时解决了地址短缺的问题。

1. **NAT的实现方式：**

**静态NAT：**也就是静态地址转换，是指一个公网IP对应一个私有IP，是一对一的转换，而且这里**只进行了IP转换并没有进行端口转换**。



**NAPT：端口多路复用技术**，和静态NAT的区别是，NAPT不但要进行IP的转换，还需要进行传输层的端口转换，具体来说对外只有一个公网IP，通过端口来区别不同私有IP主机的数据。

NAPT又分为对称性NAT和锥型NAT，对称性NAT是一个请求对应一个端口；锥型NAT是多个请求对应一个端口，只要源端口不变，无论发往的目的IP是否相同，在NAT上都映射为一个端口，形容起来就像锥子一样。



1. **NAT的优点**

* NAT能同时让多个计算机联网，并隐藏其内网IP，因此也增加了内网的安全性。
* NAT对来自外部的数据查看其NAT映射记录，对没有相应记录的数据包进行拒绝，提高了网络安全性。

1. **NAT缺点**

* NAT设备会对数据包进行编辑修改，这样就降低了发送数据的效率。
* 有的协议是无法通过NAT的，这就需要穿透技术来解决。

## 动态路由协议

RIP：适用于小型网络，RIP采用周期性广播的方式更新其路由表，即相互交换路由表学习自己不知道的网段（新路由加入时，初始化的路由表上有该路由直通的其他路由信息）。RIP需要经过N轮学习完成路由器的所有网段位置的学习进而收敛，实现内部网关全网可达。

OSPF：开放最短路径优先协议，OSPF接口发送Hello包，**建立邻居关系**，之后学习**链路状态信息**（互相发送LSA链路状态通告相互通告路由），形成**链路状态数据库**（LSDB） 。再通过Dijkstra算法，计算最短路径（cost最小）后放入路由表。

BGP协议：边界网关协议，是一种外部网关协议，在电信、网通之间实现路由的相互传递。

## RIP的防环机制

（1）规定最大跳数为16，凡是达到16跳为不可达路由。

（2）水平分割：收到广播信息的端口不进行广播。不加限制的时候，RIP环产生的一个重要原因是路由器所有信息发往所有相邻路由器。经过水平分割之后，路由器必须有选择的发送路由信息，即当向某个相邻路由器发送更新消息时，绝不能包含从从该路由得到的信息。

（3）路由毒化或毒性反转：如果某个路由挂掉了，立刻将该目标的距离设为16不可达，并泛洪告诉其他路由，其他路由收到消息之后会从自己的路由表中删除（此处的删除非真正的删除，只是隐藏起来在RIP的数据库中，并启动一个**抑制计时器**，在有效时间范围之内若该掉线的路由恢复可重新放回到路由表中）。

## ICMP协议（ping的工作原理）

IP协议不是一个可靠的协议，无法保证数据被送达，因此保证数据送达的工作需要由其他模块来完成，其中一个重要的模块就是ICMP协议。

当IP数据包发生错误，比如主机不可达，路由不可达等，ICMP会把错误信息封包，然后传送回给主机；我们一般用ICMP协议检测网络是否畅通，基于ICMP协议的工具主要有**ping和traceroute。**

## CSMA/CD协议

提供寻址和媒体存取的控制方式，使得不同设备或网络上的节点可以在多点的网络上通信而不相互冲突。

## RPC协议

Rpc是一种远程过程调用，它是一种通过网络从远程服务器程序上请求服务，而不需要了解底层网络技术的协议，他的提出是为了在公司内部的服务调用效率更高一些，比如在分布式中不同服务之间的调用像本地调用一样简单。假如此时用http请求的话，因为包含的头部信息太多就会比较慢。

Rpc协议一般会包含序列化协议和传输协议，其中传输协议一般可以选tcp协议，tcp的话可以自定义上层协议（应用层的），效率会很高，rpc主要是在序列化和反序列化上耗时多，因为将数据存入文件或者通过网络进行发送，就需要将数据对象转化为字节流。

## RPC协议介绍（Thrift）

[Thrift](https://thrift.apache.org/" \t "/Users/learna/Documents\x/_blank)是一个跨语言的服务部署框架，thrift通过IDL接口定义语言来定义[RPC](http://www.ibm.com/developerworks/cn/aix/library/au-rpc_programming/" \t "/Users/learna/Documents\x/_blank)的接口和数据类型，然后通过thrift编译器生成不同语言的代码，并由生成的代码负责RPC协议层和传输层的实现。

优点：

1. 一站式解决序列化和RPC支持
2. 数据类型丰富，基本数据类型包括bool、byte、i16（short）、i32（int）、i64（long）、double、string，特殊的数据类型binary、struct类型、容器list、set、map
3. RPC和序列化性能可以
4. 很多项目支持thrift，包括hive、spark

缺点：

1. 文档少（2）兼容性差，版本可能出现不兼容

## HTTP和RPC的区别

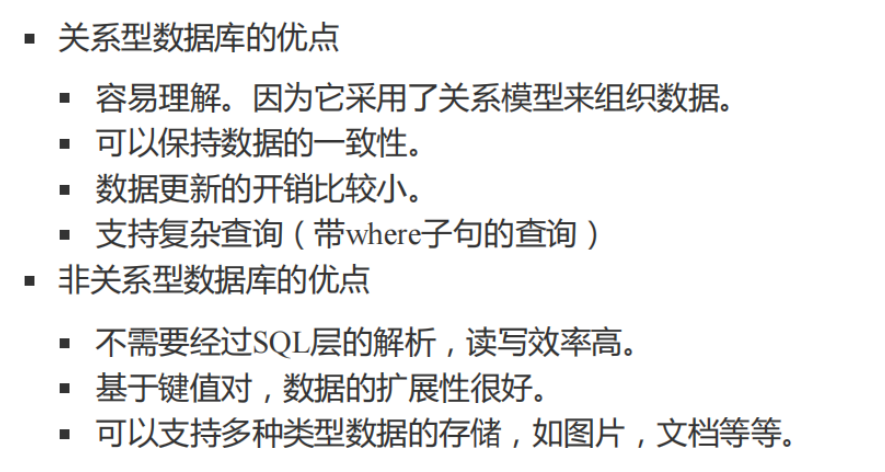
两者都常用于实现服务，在这个层面最本质的区别是RPC服务主要工作在TCP协议之上（也可以在HTTP协议），而HTTP服务工作在HTTP协议之上。由于HTTP协议基于TCP协议，所以RPC服务天然比HTTP更轻量，效率更胜一筹。

RPC接口即相当于调用本地接口一样调用远程服务的接口，HTTP接口是基于http协议的post接口和get接口（等等，2.0版本协议子支持更多）。

RPC主要用于公司内部服务调用，性能消耗低，传输效率高，服务治理方便。HTTP主要用于对外的异构环境，浏览器调用，APP接口调用，第三方接口调用等等。

# 数据库总结

## 关系数据库和非关系数据库



## MySQL存储引擎

InnoDB：支持事务处理，支持外键，支持崩溃修复能力和并发控制。如果需要对事务的完整性要求比较高，要求实现并发控制，那选择InnoDB有很大的优势。如果需要频繁的更新、删除操作的数据库，也可以选择InnoDB，因为支持事务的提交和回滚。

MyIsam：插入数据快、空间和内存使用比较低。如果表主要用于插入新纪录和读出记录，那么可以提高效率。

**InnoDB和MyISAM的差别：前者B+树叶结点存储数据本身，后者B+树叶节点存储的是数据的物理地址。**

Memory：所有数据都存在内存里面，读取数据快，但是安全性不高。如果要求很快的读写速度，安全性要求不高的话可以选择。

## --------------------------B+树相关问题--------------------------

## B+树的原理（索引结构）

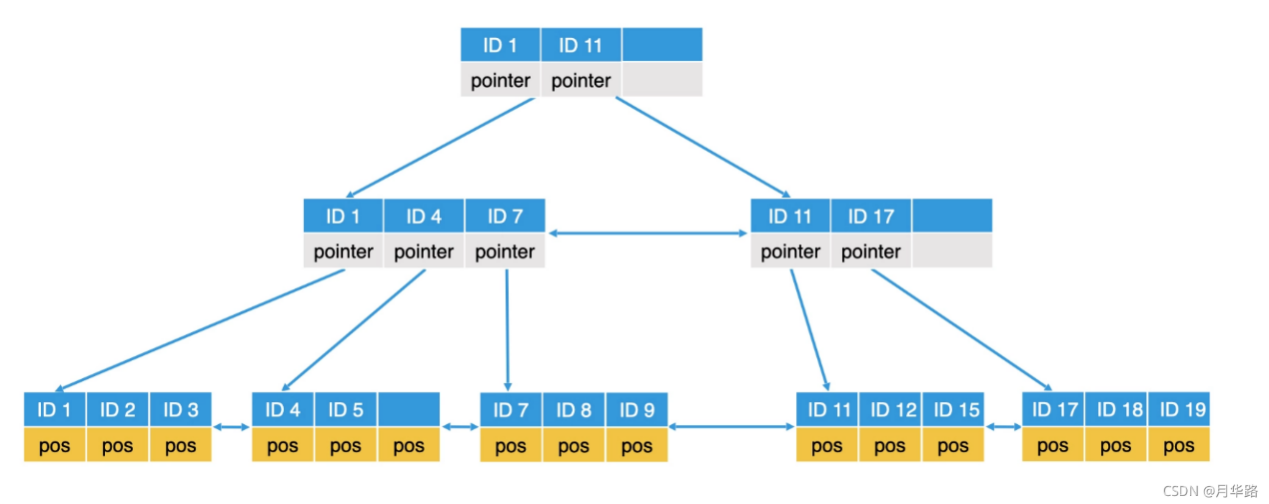
B+树是应文件系统所需而出的一种B树的变型树。

（1）B+有n棵子树的结点中含有n个关键字，每个关键字不保存数据，只用来索引，所有数据都保存在叶子节点，因此在放入数据的时候，B+树不将数据信息放入结点中，因此一次性放入内存的结点数变多，IO次数就会减少，以此提高性能，而且在查询的时候也能更好地利用空间局部性原理。

（2）所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小从小到大顺序链接。

（3）B+树还将同层的节点串链起来形成双向链表，这样就具有了范围查询和灵活调整的能力。

（4）所有的非叶子节点可以看成是索引部分。通常在B+树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点。



## MySQL采用B+树比B树的优势

1、B+树查询速度更稳定：B+所有关键字数据地址都存在叶子节点上，所以每次查找的次数都相同所以查询速度要比B树更稳定。

1. B+树天然具备排序功能：B+树所有的叶子节点数据构成了一个有序链表，在查询大小区间的数据时候更方便，数据紧密性很高，缓存的命中率也会比B树高。
2. B+树全节点遍历更快：B+树遍历整棵树只需要遍历所有的叶子节点即可，而不需要像B树一样需要对每一层进行遍历，这有利于数据库做全表扫描。

## 文件索引和数据库索引为什么使用B+树

1. 因为数据库查询的文件都很多一般不能全放入内存中而要放入磁盘，所以数据库应用了局部性原理，将每个结点的大小等于一个页，这样每次io就能读入结点的全部数据，而红黑树的逻辑相似的结点在物理上不一定相似，无法利用局部性。
2. B+树相比于b树最大的优点就是方便扫库而且查询稳定，因为数据库会有很多的范围操作，b+树的话扫一遍叶子结点就完成了遍历，而b树只能用中序遍历，相比之下做范围扫描会更麻烦，而且b+树将关键字信息放在叶子节点中，每次查询遍历的结点数也一样，查询效率更稳定。
3. B树由于每个结点都存在指向关键字的指针信息，所以相比于b+树结点要更大，而当加载一个结点的全部关键字时，由于b+树结点更小，所以盘块中存放的关键字就更多，所以进行操作时io次数也降低了。

## Mysql的索引为什么使用B+树而不使用跳表

B+树是多叉树结构，每个结点都是一个16k的数据页，能存放较多索引信息，三层左右就可以存储2kw左右的数据。也就是说查询一次数据，如果这些数据页都在磁盘里，那么最多需要查询三次磁盘IO。

跳表是链表结构，一条数据一个结点，如果最底层要存放2kw数据，且每次查询都要能达到二分查找的效果，跳表大概高度远在B+树之上，大概需要24层。最坏情况下，这24层数据会分散在不同的数据页里，也即是查一次数据会经历24次磁盘IO。

因此存放同样量级的数据，B+树的高度比跳表的要少，如果放在mysql数据库上来说，就是磁盘IO次数更少，因此B+树查询更快。

而针对写操作，B+树需要拆分合并索引数据页，跳表则独立插入，并根据随机函数确定层数，没有旋转和维持平衡的开销，因此跳表的写入性能会比B+树要好。

## MySQL索引为什么使用B+树不使用Hash

Hash 索引结构的特殊性，其检索效率非常高，索引的检索可以一次定位，不像B+Tree 索引需要从根节点到枝节点，最后才能访问到页节点这样多次的IO访问，所以 Hash 索引的查询效率要远高于 B+Tree 索引。

虽然 Hash 索引效率高，但是 Hash 索引本身由于其特殊性也带来了很多限制和弊端，主要有以下这些：

（1）哈希表不能实现范围查询。由于 Hash 索引比较的是进行 Hash 运算之后的 Hash 值，所以它只能用于等值的过滤，不能用于基于范围的过滤，因为经过相应的 Hash 算法处理之后的 Hash 值的大小关系，并不能保证和Hash运算前完全一样。

（2）Hash 索引无法被用来避免数据的排序操作。由于 Hash 索引中存放的是经过 Hash 计算之后的 Hash 值，而且Hash值的大小关系并不一定和 Hash 运算前的键值完全一样，所以数据库无法利用索引的数据来避免任何排序运算；

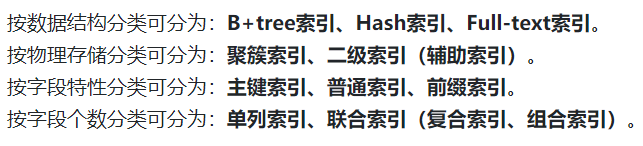
（3）Hash 索引不能利用部分索引键查询。对于组合索引，Hash 索引在计算 Hash 值的时候是组合索引键合并后再一起计算 Hash 值，而不是单独计算 Hash 值，所以通过组合索引的前面一个或几个索引键进行查询的时候，Hash 索引也无法被利用。

（4）Hash 索引在任何时候都不能避免表扫描。Hash 索引是将索引键通过 Hash 运算之后，将 Hash运算结果的 Hash 值和所对应的行指针信息存放于一个 Hash 表中，由于不同索引键可能存在相同 Hash 值，所以即使取满足某个 Hash 键值的数据的记录条数，也无法从 Hash 索引中直接完成查询，还是要通过访问表中的实际数据进行相应的比较，并得到相应的结果。

（5）Hash 索引遇到大量Hash值相等的情况后性能并不一定就会比B+Tree索引高。对于选择性比较低的索引键，如果创建 Hash 索引，那么将会存在大量记录指针信息存于同一个 Hash 值相关联。这样要定位某一条记录时就会非常麻烦，会浪费多次表数据的访问，而造成整体性能低下。

## -----------------------------索引部分------------------------------

## 索引都有哪些分类



Full-test索引=底部原理是倒排索引。

## 两种索引方式（聚簇索引和非聚簇的区别）

聚簇索引就是按照每张表的主键构造一颗B+树，同时叶子节点中存放的就是整张表的行记录数据，所以也将聚簇索引的叶子节点称为数据页，这个特性决定了索引组织表中数据也是索引的一部分，每张表只能拥有一个聚簇索引。

非聚簇索引，在聚簇索引之上创建的索引，也称为辅助索引，非聚簇索引访问数据总是需要二次查找，因为非聚簇索引的叶子节点存储的不再是行的物理位置，而是主键值。通过辅助索引首先找到的是主键值，再通过主键值找到数据行的数据页，相当于进行了一个回表的过程。

聚簇索引优点：A、数据访问更快，因为聚簇索引将索引和数据保存在同一个B+树中，因此从聚簇索引中获取数据比非聚簇索引更快。B、聚簇索引对于主键的排序查找和范围查找速度非常快。

聚簇索引缺点：A、插入速度严重依赖于插入顺序，按照主键的顺序插入是最快的。B、更新主键的代价很高，会导致被更新的行移动

## Hash索引和B+树索引的区别

（1）Hash索引不能进行范围查询，而B+树可以。

这是因为Hash索引指向的数据是无序的，而B+ 树的叶子节点是个有序的链表。

（2）Hash索引不支持联合索引的最左侧原则，而B+树可以。

对于联合索引来说，Hash索引在计算Hash值的时候是将索引键合并后再一起计算Hash值，所以不会针对每个索引单独计算Hash值。因此如果用到联合索引的一个或多个索引时，联合索引无法被利用。

（3）Hash索引不支持Order BY排序，而B+树支持。

因为Hash索引指向的数据是无序的，因此无法起到排序优化的作用，而B+树索引数据是有序的，可以起到对该字段Order By 排序优化的作用。

（4）Hash索引无法进行模糊查询。而B+ 树使用 LIKE 进行模糊查询的时候，LIKE后面前模糊查询（比如%开头）的话可以起到优化的作用。

（5）Hash索引在等值查询上比B+树效率更高。

不过也存在一种情况，就是索引列的重复值如果很多，效率就会降低。这是因为遇到Hash冲突时，需要遍历桶中的行指针来进行比较，找到查询的关键字非常耗时。所以Hash索引通常不会用到重复值多的列上，比如列为性别，年龄等。

## MySQL唯一索引和普通索引的选择

**这两个索引的差别主要在更新时候的change buffer过程中。**

**在查询上：普通索引**找到某个ID后会继续判断下一位是否符合要求，不符合则退出，从性能上看它并没有造成太大的影响；**唯一索引**找到某个ID则直接返回结果。

**在修改上：普通索引**的修改有**change buffer机制，**如果修改的内容在内存中则直接进行修改，如果不在内存中则写入change buffer即可；而**唯一索引**修改的第一步要判断对应ID是否存在，若存在则无法进行新增，所以每次都需要读取，很消耗时间。

因此针对**多改少读**情况change buffer带来的收益很可观，而针对**多读少改**的情况要去根据实际情况判断，因为有可能IO次数并没有减少，反而增加了change buffer的维护代价。

## 什么字段适合建索引、什么时候不适合建索引

1. 表的主键、外键必须有索引。
2. 经常与其他表进行连接的表，在连接字段上需要建索引。
3. 经常出现在where子句中的字段，应该建索引；where里面用不到的字段不要建索引。
4. 经常选择的字段应该建立索引。
5. 索引应该建立在小字段上，大字段一般不要去建索引。
6. 表记录太少或者经常经常增删改的表（更新时mysql不仅要保存数据，还会保存索引，会加重IO负担）。

## 自增主键问题，为什么要使用自增主键

**自增主键的优点：**B+树为了维护索引有序性，在插入新值的时候需要做必要的维护。如果插入的值比最大值id大，则只需要最后记录后面插入一个新记录。如果新插入的ID值在原先的有序中间，就麻烦了，需要逻辑上挪动后面的数据，空出位置进行插入。如果所在的数据页已经满了，根据 B+ 树的算法，这时候需要申请一个新的数据页，然后挪动部分数据过去，这个过程就是页分裂的过程，因此在这种情况下，性能自然会受影响；除了性能外，页分裂操作还影响数据页的利用率。原本放在一个页的数据，现在分到两个页中，整体空间利用率降低大约 50%；自增主键的插入数据模式，正符合了递增插入的场景。每次插入一条新记录，都是追加操作，都不涉及到挪动其他记录，也不会触发叶子节点的分裂。

**主键不自增行不行：**可以把建表sql里的AUTO\_INCREMENT去掉，但是此时在写数据的时候需要自己指定id的值是多少。

## 数据库选择什么作为表的主键

1. 自增主键设计：

首先可以使用BIGINT 的自增类型作为主键，同时由于整型的自增性，数据库插入也是顺序的，性能较好，但是一般使用 BIGINT 的自增类型作为主键的设计仅仅适合非核心业务表，比如告警表、日志表等。真正的核心业务表，一般不会用自增键做主键。

**主要有以下几个原因：**

* **自增值在服务器端产生，存在并发性能问题**：自增值是在 MySQL 服务端产生的值，需要有一把自增的锁进行保护，若这时有大量的插入请求，就可能存在自增引起的性能瓶颈，在 MySQL 数据库中，参数 innodb\_autoinc\_lock\_mode (0,1,2)用于控制自增锁持有的时间。
* **自增值做主键，只能在当前实例中保证唯一，不能保证全局唯一**；
* **公开数据值，容易引发安全问题**，例如知道地址www.abc.com/User/10/，很容猜出 User 有 11、12 依次类推的值，容易引发数据泄露；

1. UUID主键设计:

UUID代表全局唯一标识 ID，由于其全局唯一性，所以能够把它用来作为数据库分库分表的主键。

UUID = 12位时间+10位的IP地址+6位生成序列号+最后4位的分库和分表id。

在存储时间时，UUID 是根据时间位逆序存储，也就是低时间低位存放在最前面，高时间位在最后，UUID 的前 4 个字节会随着时间的变化而不断“随机”变化，并非单调递增。在插入时会产生离散 IO，从而产生性能瓶颈。为了解决这个问题，MySQL8.0推出了UUID\_TO\_BIN函数：

* **能够把UUID字符串的时间高位放在前面，解决了UUID插入乱序的问题；**
* **去掉了无用的字符串"-"，精简存储空间；**
* **将字符串其转换为二进制值存储，空间最终从之前的 36 个字节缩短为了 16 字节。**

## 为什么要建立索引

1. 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
2. 可以大大加快数据的检索速度，这也是创建索引的最主要的原因。
3. 帮助服务器避免排序和临时表。
4. 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

## 建立索引的优缺点

* 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
* 建立索引可以有效缩短数据的检索时间。
* 建立联合索引可以加快表与表之间的连接。
* 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。
* 创建索引和维护索引要耗费时间，随着数据量的增加耗费的时间也会增加。
* 索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间。
* 当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

## 复合索引

包含两个或以上字段的索引叫复合索引，它的存在主要是为了多条件的查询，比如说where后面加了好几个条件，一个是A字段相关的，一个是B字段相关的。在复合索引中，记录首先按照第一个字段排序。对于在第一个字段上取值相同的记录，系统再按照第二个字段的取值排序，以此类推。所以进行查询的时候必须包含第一个字段，这就是复合索引的最左前缀原则。

## 复合索引的最左匹配原则

在数据库联合索引中，只不过联合索引的健值数量不是一个，而是多个。构建一颗B+树只能根据一个值来构建，因此数据库依据联合索引最左的字段来构建B+树。因此在搜索的时候如果符合最左匹配原则去使用对应的值进行搜索，但是只限于等值查询，如果有了范围查询，后面的索引就用不到了。

## 索引下推

* 不使用索引条件下推优化时存储引擎通过索引检索到数据，然后返回给MySQL服务器，服务器然后判断数据是否符合条件。
* 当使用索引条件下推优化时，如果存在某些被索引的列的判断条件时，MySQL服务器将这一部分判断条件传递给存储引擎，然后由存储引擎通过判断索引是否符合MySQL服务器传递的条件，只有当索引符合条件时才会将数据检索出来返回给MySQL服务器。
* 例如在复合索引结构中，如果第一个字段采用了模糊匹配，如果没有索引下推，那么会忽略第二个字段直接进行回表判断，有了索引下推的优化，则会判断完第二个字段是否复合在进行回表操作，减少了回表次数，提高了查询销量。

## 前缀索引

前缀索引也叫局部索引，比如给身份证的前 10 位添加索引，类似这种给某列部分信息添加索引的方式叫做前缀索引；能有效减小索引文件的大小，让每个索引页可以保存更多的索引值，从而提高了索引查询的速度，使用前缀索引要降低重复的索引值。当字符串本身可能比较长，而且前几个字符就开始不相同，适合使用前缀索引；相反情况下不适合使用前缀索引。

## 索引失效的情况

在MySQL中，索引失效通常有如下八种情况：

1. 查询条件中存在OR语句，MySQL会放弃索引走全线扫描。
2. LIKE查询语句中以%开头的，这个时候表示一定条件的任一字符串，也只能全线扫描。
3. 对查询的列上有运算或者函数的。
4. 如果列的类型是字符串，那一定要在条件中使用引号，否则不使用索引。
5. 左联结查询或者右联结查询，查询关联的字段编码风格不一样（GBK对应UTF-8）
6. MySQL如果估计使用全表扫描要比使用索引快的时候，不使用索引。
7. 联结查询中，按照优化器顺序的第一张表不会走索引。比如A left join B，此时优化器执行的顺序是A、B，首先全表扫描A，在根据A中的键查找B中的值。
8. 如果查询中没有用到复合索引的第一个字段，则不会走索引。（按照上面的例子，没有姓，怎么按照索引找到名呢）

## 分库分表情况下的id设置

采用uuid算法：UUID代表全局唯一标识 ID，由于其全局唯一性，所以能够把它用来作为数据库分库分表的主键。

UUID = 12位时间+10位的IP地址+6位生成序列号+最后4位的分库和分表id。

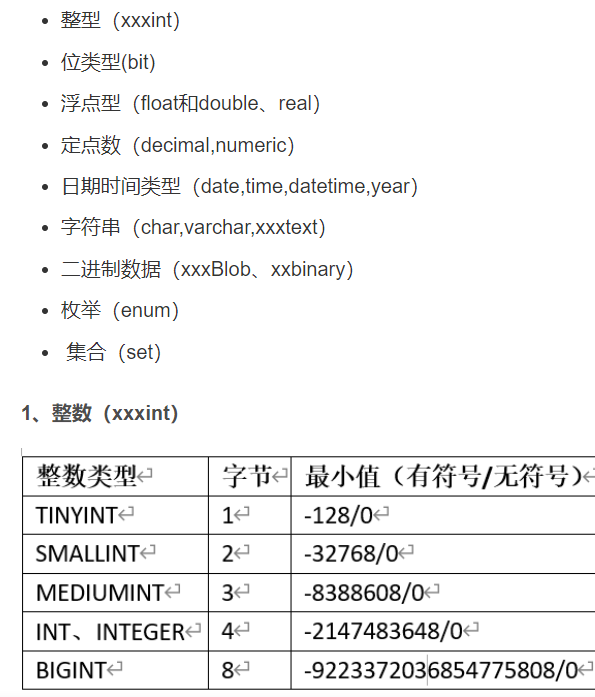
在存储时间时，UUID 是根据时间位逆序存储，也就是低时间低位存放在最前面，高时间位在最后，UUID 的前 4 个字节会随着时间的变化而不断“随机”变化，并非单调递增。在插入时会产生离散 IO，从而产生性能瓶颈。为了解决这个问题，MySQL8.0推出了UUID\_TO\_BIN函数：

* **能够把UUID字符串的时间高位放在前面，解决了UUID插入乱序的问题；**
* **去掉了无用的字符串"-"，精简存储空间；**

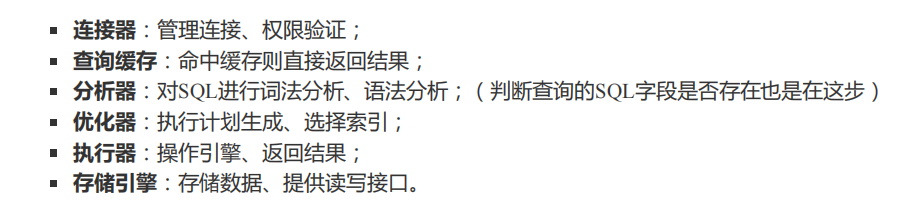
**将字符串其转换为二进制值存储，空间最终从之前的 36 个字节缩短为了 16 字节。**

## ---------------------------MySQL专属---------------------------

## MySQL常见数据类型



## 说一下MySQL是如何执行一条SQL的



## MySQL中InnoDB的当前读和快照读

读取的是最新版本，像**UPDATE、DELETE、INSERT、SELECT ... FOR UPDATE**这些操作都是一种**当前读**，就是它读取的是记录的最新版本，读取时还要保证其他并发事务不能修改当前记录，会对读取的记录进行加锁。

读取的是快照版本，也就是历史版本，像不加锁的SELECT操作就是**快照读**，即不加锁的非阻塞读；快照读的前提是隔离级别是读已提交和可重复读，因为未提交读总是读取最新的数据行，而不是符合当前事务版本的数据行，而可串行化则会对表加锁。

## MySQL事务并发的（脏读、不可重复、幻读）

1. 脏读：表示一个事务处理过程中读取了另一个事务未提交的数据。
2. 不可重复读：表示的是事务在读取过程中多次读取返回了不同的结果（针对同一数据项）。
3. 幻读：表示在一个事务读的过程中，可能插入了另一个事务的记录，影响了事务读的结果（针对的是表项的插入/删除操作）。

## MySQL的隔离模式

|  |  |  |  |
| --- | --- | --- | --- |
| （+表示会出现） | 脏读 | 不可重复读 | 幻读 |
| 读未提交 | + | + | + |
| 读已提交 | - | + | + |
| 可重复读 | - | - | + |
| 可串行化 | - | - | - |

**读未提交：**

* 事务对当前被读取的数据不加锁，都是当前读；
* 事务在更新某数据的瞬间（就是发生更新的瞬间），必须先对其加行级共享锁，直到事务结束才释放。

**读已提交：**

* 事务对当前被读取的数据不加锁，且是快照读；
* 事务在更新某数据的瞬间（就是发生更新的瞬间），必须先对其加行级排他锁（Record），直到事务结束才释放。

**可重复读：**

* 事务对当前被读取的数据不加锁，且是快照读；
* 事务在更新某数据的瞬间（就是发生更新的瞬间），必须先对其加行级排他锁（Record，GAP，Next-Key），直到事务结束才释放。
* 通过间隙锁，在这个级别MySQL就解决了幻读的问题。
* 通过快照，在这个级别MySQL就解决了不可重复读的问题。

**可串行化：**

* 事务在读取数据时，必须先对其加**表级共享锁** ，直到事务结束才释放，都是当前读；
* 事务在更新数据时，必须先对其加**表级排他锁** ，直到事务结束才释放。

**为什么默认隔离模式是可重复读？**

历史原因,早阶段Mysql(5.1版本之前)的Binlog类型Statement是默认格式，即依次记录系统接受的SQL请求；5.1及以后，MySQL提供了Row,Mixed,statement 3种Binlog格式, 当binlog为statement格式,使用读已提交隔离级别时,会出现BUG因此Mysql将可重复读作为默认的隔离级别！

* 在可重复读的隔离模式下，由于数据库的修改会增加间隙锁，但是在读已提交隔离模式下，如果有某些连续修改的情况有可能会导致数据不一致的问题，而且在重新恢复的时候，可能有事务提交的顺序问题，以及索引选择不同问题，在一些语句执行上会出现歧义。
* 在数据恢复方面使用row模式也会更友好，针对delete语句，insert语句，update语句，使用row格式有时只需做简单的修改就能恢复成原来的样子。

## 隔离模式的实现原理（MVCC实现原理在下面）

读未提交（RU：read-uncommitted）：在RU级别中，事务读到的所有数据都是最新的数据，可能是事务提交后的数据，也可能是事务执行中的数据（可能会被回滚）。当隔离级别为读未提交时：（1）所有的读不加锁，读到的数据都是最新的数据，性能最好。（2）所有的写加行级锁，写完释放。

读已提交（RC：read-committed）：使用MVCC，在每一行加入隐藏的字段（DB\_TRX\_ID：修改该行的最后一个事务的id，DB\_ROLL\_PTR：指向当前行的undo log日志，DB\_ROW\_ID：行标识，DELETE\_BIT：删除标志），它实现了不加锁的读操作。当隔离级别为RC时：（1）写操作：加行级锁。事务开始后，会在UNDO日志中写入修改记录，数据行中的隐藏列POLL\_PTR存储指向该行的undo log记录的指针。（2）读操作：不加锁。在读取时，如果该行被其它事务锁定，则顺着隐藏列POLL\_PTR指针，找到上一个有效的历史记录/\*（有效的记录：该记录对当前事务可见，且DELETE\_BIT=0）\*/。

可重复读（RR：repeatable-read）：使用MVCC技术来实现不加锁的读操作。当隔离级别为RR时：（1）写操作：加行级锁。事务开始后，会在UNDO日志中写入修改记录，数据行中的隐藏列POLL\_PTR存储指向该行的undo log记录的指针。（2）读操作：不加锁。在读取时，如果该行被其它事务锁定，则顺着隐藏列POLL\_PTR指针，找到上一个有效的历史记录（有效的记录：该记录对当前事务可见，且DELETE\_BIT=0）。

串行化读的实现：读写的过程都会加锁。

**（MVCC在读已提交和可重复定读中的区别）**

RC和RR级别的操作基本相同，而不同之处在于：行记录对于当前事务的可见性（可见性：即哪个版本的行记录对这个事务是可见的）。RC级别对数据的可见性是该数据的最新记录，RR基本对数据的可见性是事务开始时，该数据的记录。

**行记录可见性的实现（MVCC实现原理）：**

在innodb中，创建一个事务的时候，会将当前系统中的活跃事务列表创建一个快照（read\_view），里面存储着的都是在当前事务开始时，还没commit的事务，这些事务里的值对当前事务不可见。read\_view中有两个关键值 up\_limit\_id（当前未提交事务的最小版本号-1，在up\_limit\_id之前的事务都已经提交，在up\_limit\_id之后的事务可能提交，可能还没提交） 和 low\_limit\_id（当前系统尚未分配的下一个事务id，也就是目前已出现过的事务id的最大值+1。///\*注意：low\_limit\_id不是最大的活跃事务的id。\*///）

读已提交和可重复读判断行可见性的逻辑是一样的。当该事务要读取undo中的行记录时，会将行记录的版本号（DB\_TRX\_ID）与read\_view（快照）进行比较：

* 如果DB\_TRX\_ID小于up\_limit\_id，表示该行记录在当前事务开始前就已经提交了，并且DELETE\_BIT=0，则该行记录对当前事务是可见的。
* 如果DB\_TRX\_ID大于low\_limit\_id，表示该行记录在所在的事务在本次事务创建后才启动的，所以该行记录的当前值不可见。
* 如果up\_limit\_id< = DB\_TRX\_ID <= low\_limit\_id，判断DB\_TRX\_ID是否在活跃事务链中，如果在就是不可见，如果不在就是可见的。
* 如果上面判断都是不可见的，则读取undo中该行记录的上一条行记录，继续进行判断。

读已提交和可重复度之间的区别是由于read\_view（快照）的生成时机不一样导致的，读已提交会在执行语句的时关闭之前的快照重新生成新的快照，而可重复读的快照只在事务开始时创建。

## MySQL事务的特性（ACID特性）

原子性：利用回滚日志（undo log）来保持，回滚日志可以撤销所有已成功的sql执行信息，它需要记录当前需要回滚的日志信息，当事务执行失败后，会利用回滚日志返回到修改之前的样子。

一致性：数据库的一致性其实包含两个层面。从数据库层面理解，“一致性”实际上是一组约束、规则，表示事务执行完毕后数据库的约束、规则还是一致的，通过其他三个特性来保证一致性。从业务层面讲，“一致性”实际上是指事务把数据库从一个有效的状态转移成另一个有效状态，通过代码判断数据库数据是否有效，然后决定回滚还是执行。

隔离性：通过隔离模式（读未提交、读已提交、可重复读、可串行化）实现不同级别下的隔离，利用的锁和MVCC机制实现隔离模式下的读写并行。MVCC即多版本并发控制，指的是一个读已提交、可重复读的隔离级别的事务执行select时访问记录的版本链的过程，一个行记录多个版本的快照数据，这些快照数据在回滚日志中。

持久性：通过redo log重做日志来保持，当我们进行数据修改的时候不仅会在内存上操作，还会在重做日志上记录，当事务提交的时候会将重做日志进行刷盘（一部分在内存上，一部分在磁盘上），当数据库宕机重启的时候会将日志中的内容恢复到数据库上，然后在根据redo log和bin log的内容选择回滚还是提交。

## 数据库一致性和cap一致性有什么区别

**数据库 ACID 的一致性**也称之为**内部一致性**，可以说是是事务开始前和结束后，数据库的完整性约束没有被破坏。比如 A 向 B 转账，不可能 A 扣了钱，B 却没收到。

**分布式 CAP 的一致性**也称之为**外部一致性**，可以说是为在分布式系统中，写操作后再读，就必须返回写入的值。比如分布式数据库A、B、C，A 中写入数据x = 1，写完马上读 B 和 C，就一定要读出x = 1，读出来我们就称之为符合一致性。

**两者区别**，内部一致性注重于事务前后数据的完整性，而外部一致性则注重于读写数据的一致性。

## MySQL三个范式

1. 第一范式：是关系数据库最基本的要求，表示数据库中的每一项都是不可分割的数据项，表示了原子性即字段不可再分。
2. 第二范式：第二范式在第一范式的基础上，显示了唯一性，有主键，非主键字段依赖主键
3. 第三范式：在第二范式的基础上，每列都与主键有直接关系，不存在传递依赖

## 分布式事务种类

分布式事务就是要要在分布式系统中实现事务，它其实是由多个本地事务组合而成。

1. **二阶段提交**

是一种强一致性设计，2PC 引入一个事务协调者的角色来协调管理各参与者（也可称之为各本地资源）的提交和回滚，二阶段分别指的是准备（投票）和提交两个阶段。

**准备阶段**协调者会给各参与者发送准备命令，同步等待所有资源的响应之后就进入第二阶段即提交阶段（注意提交阶段不一定是提交事务，也可能是回滚事务）。

假如在第一阶段所有参与者都返回准备成功，那么协调者则向所有参与者发送**提交事务命令**，然后等待所有事务都提交成功之后，返回事务执行成功。

1. **三段式提交**

三段式表示：准备阶段、预提交阶段和提交阶段，首先会在准备阶段去访问参与者自身情况是否能够执行事务，如果全部正常工作则进入预提交阶段，然后进入提交阶段进行提交。

三段式提交针对二段式进行了一定的改进：引入了参与者超时机制，并且增加了预提交阶段使得故障恢复之后协调者的决策复杂度降低，但整体的交互过程更长了，性能有所下降，并且还是会存在数据不一致问题。

1. **TCC**

TCC中表示Try 指的是预留，即资源的预留和锁定，注意是预留；Confirm 指的是确认操作，这一步其实就是真正的执行了；Cancel 指的是撤销操作，可以理解为把预留阶段的动作撤销了。

两段式和三段式都是数据库层面的，而 TCC 是业务层面的分布式事务，所以TCC**可以跨数据库、跨不同的业务系统来实现事务。**

1. **本地消息表**

本地消息表其实就是利用了各系统本地的事务来实现分布式事务。

本地消息表就是会有一张存放本地消息的表，一般都是放在数据库中，然后在执行业务的时候将执行的业务和消息放入消息表中的操作放在同一个事务中，这样就能保证消息放入本地表中业务肯定是执行成功的。

然后再去调用下一个操作，如果下一个操作调用成功，消息表的消息状态可以直接改成已成功。如果调用失败则会有后台任务定时去读取本地消息表，筛选出还未成功的消息再调用对应的服务。

1. **消息事务（RocketMQ能够很好支持）**

先给 Broker 发送事务消息即半消息，半消息不是说一半消息，而是这个消息对消费者来说不可见，然后发送成功后发送方再执行本地事务。

再根据本地事务的结果向 Broker 发送 Commit 或者 RollBack 命令。

1. **最大努力通知**

就本地消息表来说会有后台任务定时去查看未完成的消息，然后去调用对应的服务，当一个消息多次调用都失败的时候可以记录下然后引入人工，或者直接舍弃。这也算是尽最大努力了。

## MySQL的两段式提交及崩溃恢复（故障恢复）

MySQL进行事务提交的时候分为三步：写入redo log处于prepare状态；写入binlog并落盘；修改redo log为commit状态。

（1）使用两段式提交就是为了保证数据库逻辑的一致性。

当redo log处于prepare状态数据库宕机后，redo log可以进行异常恢复，此时redo log中有修改而binlog中没有对应修改，数据库重现的时候就会导致数据不一致。所以此时应该将事务回滚。

1. 当没有redo log时，如果只有binlog存在：

* 先写binlog在进行数据修改，如果在第一步完成后服务器异常关闭，则数据库会比binlog中缺少修改，进行备库的时候会导致主从不一致。
* 先进行数据修改在进写binlog，如果在第一步完成后服务器异常关闭，则会导致binlog中缺少了最后的修改数据。

**当需要崩溃恢复时**，如果redo log中事务是完整的并且有了commit标识，则直接提交；而当redo log中只有prepare标识时，需要判断binlog中事务是否完整，如果是则提交事务，否则回滚事务。

## MySQL刷脏页、刷盘的时机

1. Redo Log写满了, 需要将 checkpoint 向前推进, 以便继续写入日志，checkpoint 向前推进时, 需要将推进区间涉及的所有脏页刷新到磁盘。
2. 内存不足, 需要淘汰一些内存页(最久未使用的)给别的数据页使用，此时如果是干净页, 则直接拿来复用，如果是脏页, 则需要先刷新到磁盘(直接写入磁盘, 不用管Redo Log, 后续Redo Log刷脏页时会判断对应数据页是否已刷新到磁盘), 使之成为干净页再拿来使用.
3. 数据库系统空闲时会刷脏页.
4. 数据库正常关闭时需要将所有脏页刷新到磁盘.

## MySQL数据删除

当Innodb删除某个数据时，并不会立马删掉，而是在b+树中对应的位置设置为“可复用”状态，等插入数据时，直接放在这个“可复用”位置即可。当数据页的全部记录被删除后，该数据页就是“可复用”的了，数据页的可复用和数据项是不一样的，数据页成为“可复用”后可以复用到b+树的任何位置上。

因此删除数据可能会出现“空洞”现象，而且插入数据也可能会出现该现象，当插入数据导致页面分裂后，页末尾就留下了空洞，此时文件大小并没有改变，通过**重建表**才会达到目的。

## MySQL5.6后的Online DDL操作

分为5步：获取MDL写锁、降级为MDL读锁、进行DDL、升级成MDL写锁、释放MDL锁，由于第三步占了绝大多数的时间，这个期间是可以正常读写数据的，所以称为online。

## MySQL的count(\*)实现

Count(\*)是在server层实现的，引擎层会把server层需要的内容发过去，然后由server层进行计数操作，在不同的引擎中，count(\*)有不同的实现方式：

* 在MyISAM中，它把表的总行数存在了磁盘上，因此执行count(\*)会直接返回这个数，效率很高（注意是没有where过滤的累计）。
* 在InnoDB中，需要一行一行地从引擎中读出来，然后累积计数，效果不佳。（由于MVCC的原因（InnoDB默认可重复读隔离模式），同一时刻读取多少行是无法确定的，而MyISAM不支持事务，所以就没这个问题）

## MySQL的MRR(提升读性能)

借助MRR（Multi-Range Read），其目的是尽量使用顺序读盘。对于SQLselect \* from t1 where a>=1 and a<=100，其执行流程是：（1）根据索引a，定位到满足条件的记录，将id值放入read\_rnd\_buffer中。（2）将read\_rnd\_buffer中的id进行递增排序。（3）排序后的id数组，依次到主键id索引中查记录，并作为结果返回。

使用MRR的原因是某个条件的查询结果其主键可能是随机的，随机访问磁盘的性能较差，所以如果按照主键的递增顺序查询的话，对磁盘的读比较接近顺序读，能够提升读性能。

## //MySQL buffer pool（MySQL缓存）

为了提升查询性能，innodb存储引擎设计缓冲池，当数据从磁盘中取出后，缓存内存中，下次查询同样的数据的时候，直接从内存中读取。

**读取数据和更新数据流程：**

当读取数据时，如果数据存在于 Buffer Pool 中，客户端就会直接读取 Buffer Pool 中的数据，否则再去磁盘中读取。

当修改数据时，首先是修改 Buffer Pool 中数据所在的页，然后将其页设置为脏页，最后由后台线程将脏页写入到磁盘。

**Buffer Pool的大小**：Buffer Pool 是在 MySQL 启动的时候，向操作系统申请的一片连续的内存空间，默认配置下Buffer Pool只有128MB。可以通过调整 innodb\_buffer\_pool\_size 参数来设置 Buffer Pool 的大小，一般建议设置成可用物理内存的 60%~80%。

InnoDB 会把存储的数据划分为若干个「页」，以页作为磁盘和内存交互的基本单位，一个页的默认大小为 16KB。因此，Buffer Pool 同样需要按页来划分。在 MySQL 启动的时候，InnoDB 会为 Buffer Pool 申请一片连续的内存空间，然后按照默认的16KB的大小划分出一个个的页， Buffer Pool 中的页就叫做缓存页。此时这些缓存页都是空闲的，之后随着程序的运行，才会有磁盘上的页被缓存到 Buffer Pool 中。所以，MySQL 刚启动的时候，你会观察到使用的虚拟内存空间很大，而使用到的物理内存空间却很小，这是因为只有这些虚拟内存被访问后，操作系统才会触发缺页中断，接着将虚拟地址和物理地址建立映射关系。Buffer Pool 除了缓存索引页和「数据页」，还包括了undo页，插入缓存、自适应哈希索引、锁信息等等。为了更好的管理这些在 Buffer Pool 中的缓存页，InnoDB 为每一个缓存页都创建了一个控制块，控制块信息包括「缓存页的表空间、页号、缓存页地址、链表节点」。由于控制快会占用空间，它是放在Buffer Pool的最前面，其次后面才是缓存页，所以会导致可能控制块与缓存页不能一一映射，因为每一个控制快都会记录缓存页的位置，这时候会导致碎片空间，所以设置Buffer Pool的大小如果设置得当就不会产生碎片问题。

## Change buffer和redo log

Redo log是在两段式提交中先写日志，后写磁盘保证数据库的完整性，性能上减少随机写磁盘的消耗，change buffer表示修改数据时并没有立即将对应的数据读入内存进行修改，而是存入change buffer等下次读入数据页在进行merge。

因此change buffer是减少了随机读的操作，redo log是减少了随机读写的操作；**redo log中存放的是change buffer的数据修改，change buffer里面存的是数据的修改，通过系统其他线程定期purge将更新数据刷到磁盘中。**

## MySQL的redo log和binlog

### Redo log和Bin log的解释

· redo log是InnoDB引擎特有的；binlog是MySQL的**Server层**实现的，**所有引擎都可以使用**。

· redo log是**物理日志**，记录的是“在某个数据页上做了什么修改”；binlog是**逻辑日志**，记录的是这个语句的原始逻辑，比如“给ID=2这一行的c字段加1 ”。

· redo log是**循环写的，空间固定会用完**；binlog是可以**追加写入的**，“追加写”是指binlog文件写到一定大小后会**切换到下一个**，并不会覆盖以前的日志。

· redo log用于保证**crash-safe能力，进行异常恢复**，binlog主要用于**归档和复制**。

### Bin log的三种格式

Binlog有三种格式，分别是statement、row和mixed。当binlog\_format=statement时，binlog里面记录的就是SQL语句的原文。

在statement的条件下，delete 带limit，很可能会出现主备数据不一致的情况，即主库和备库选择不一样的索引进行删除。

row格式的binlog里没有了SQL语句的原文，而是替换成了两个event，指定了删除的表和删除的行为。相当于binlog里面记录了真实删除行的主键id，也就不会存在主备不一致的情况。

有些statement格式的binlog可能会导致主备不一致，所以要使用row格式。但row格式的缺点是，很占空间，例如一条SQL语句删除了十万行数据，但是对row来说就需要记录着十万行数据的行为。这个时候的折中方案就是mixed格式的binlog。mixed格式的意思是，MySQL自己会判断这条SQL语句是否可能引起主备不一致，如果有可能，就用row格式，否则就用statement格式。

### Redo log的写入机制（原理）

InnoDB首先会将其存在redo log buffer中，物理上也就是在MySQL进程内存中；然后写到磁盘(write)，但是没有持久化（fsync)，物理上是在文件系统的page cache里面；最后持久化fsync到磁盘。

InnoDB有一个后台线程，每隔1秒，就会把redo log buffer中的日志，调用write写到文件系统的page cache，然后调用fsync持久化到磁盘。

实际上，除了后台线程每秒一次的轮询操作外，还有两种场景会让一个没有提交的事务的redo log写入到磁盘中。

* 一种是，redo log buffer占用的空间即将达到 innodb\_log\_buffer\_size一半的时候，后台线程会主动写盘，此时由于这个事务并没有提交，所以这个写盘动作只是write，而没有调用fsync，也就是只留在了文件系统的page cache。
* 另一种是，并行的事务提交的时候，顺带将这个事务的redo log buffer持久化到磁盘。假设一个事务A执行到一半，已经写了一些redo log到buffer中，这时候有另外一个线程的事务B提交，如果innodb\_flush\_log\_at\_trx\_commit设置的是1，那么按照这个参数的逻辑，事务B要把redo log buffer里的日志全部持久化到磁盘。这时候，就会带上事务A在redo log buffer里的日志一起持久化到磁盘。

### Binlog的写入机制（原理）

Binlog的写入逻辑：事务执行的过程中，先把日志写到**binlog cache**，事务提交的时候，再把**binlog cache写入到磁盘**中。

事务commit-->>binlog write到文件系统的page cache-->>fsync数据持久化到磁盘上。

write 和fsync的时机，是由参数sync\_binlog控制的：

* sync\_binlog=0的时候，表示每次提交事务都只write，不fsync；
* sync\_binlog=1的时候，表示每次提交事务都会执行fsync；
* sync\_binlog=N(N>1)的时候，表示每次提交事务都write，但累积N个事务后才fsync。

### MySQL是如何知道binlog是否完整？

* statement格式的binlog，最后会有COMMIT；
* row格式的binlog，最后会有一个XID event。
* 在MySQL5.6.5版本后，还引入了binlog-checksum参数来验证binlog内容的正确性。

### redo log和binlog是如何关联起来的？

它们都有一个共同的字段叫XID，当发生崩溃的时候，会按顺序扫描redo log：

* 如果既有prepare，又有commit的redo log，就直接提交；
* 如果既有prepare，又没有commit的redo log，就拿上XID去binlog中查找binlog是否完整。

### 使用InnoDB引擎的话，能否只用redo log不用binlog

* 从崩溃恢复的角度来看是可以的，此时就没有了两段式提交，但系统依然是crash-safe的。
* 但是redo log的特点是循环写，binlog是追加写，如果想要回滚到以前某一时刻的话，redo log没有对日志进行保留，redo log没办法进行归档。
* MySQL的高可用依赖于binlog进行主从复制，以及其他一些异构系统消费binlog，如果没有binlog就没办法工作了。

### Binlog的格式为什么建议设置为row格式

* 在可重复读的隔离模式下，由于数据库的修改会增加间隙锁，但是在读已提交隔离模式下，如果有某些连续修改的情况有可能会导致数据不一致的问题，而且在重新恢复的时候，可能有事务提交的顺序问题，以及索引选择不同问题，在一些语句执行上会出现歧义。
* 在数据恢复方面使用row模式也会更友好，针对delete语句，insert语句，update语句，使用row格式有时只需做简单的修改就能恢复成原来的样子。

### 为什么会有mixed格式的binlog？

* 在主从复制的过程中，因为有些statement格式的binlog可能会导致主备不一致，所以要使用row格式。
* 但row格式的缺点是，很占空间。比如你用一个delete语句删掉10万行数据，用statement的话就是一个SQL语句被记录到binlog中，占用几十个字节的空间。但如果用row格式的binlog，就要把这10万条记录都写到binlog中。这样做，不仅会占用更大的空间，同时写binlog也要耗费IO资源，影响执行速度。
* 所以，MySQL就取了个折中方案，也就是有了mixed格式的binlog。mixed格式的意思是，MySQL自己会判断这条SQL语句是否可能引起主备不一致，如果有可能，就用row格式，否则就用statement格式。

### WAL机制的特点

**MySQL为什么需要WAL？**

为了持久化数据，MySQL最终需要将数据刷到磁盘，最直观的想法就是事务提交的时候直接把数据写入磁盘，但是这样做通常会导致两个问题：

（1）一个是浪费资源，就算是只修改一个字节，也需要将整个数据页刷入磁盘（一个页面16kb，每次改动都需要将16kb的内容刷入磁盘）。

（2）第二个问题是速度慢，每个事务里可能涉及到多个数据页的修改，而这些数据可能是不相邻的，这就涉及到了大量的随机IO操作。

所以这个时候的一个改进就是使用redo log实现WAL，当做数据修改的时候，不仅在内存中操作，还会在redo log中记录这次操作。当事务提交的时候，会将redo log日志进行刷盘(redo log一部分在内存中，一部分在磁盘上)，使用redo log进行WAL有如下的好处：

（1）redo log体积小，只记录了哪一页修改的内容，因此体积小，刷盘快

（2）redo log是一直往末尾进行追加，属于顺序IO。效率显然比随机IO来的快

* redo log 和 binlog都是顺序写，磁盘的顺序写比随机写速度要快；
* 组提交机制（获取当前最大的LSN将之前未写入的redo log一并写入），可以大幅度降低磁盘的IOPS消耗。

## ------------------------------锁的部分-----------------------------

## MySQL锁的种类

1. **全局锁（类比InnoDB的可重复读）**

当进行**全库逻辑备份**的时候，当需要把全库内容select出来做成文本的时候，就需要对数据库进行加锁，让数据库无法进行更新、添加操作，MySQL提供了一种全局加锁的方法命令是FTWRL，添加后数据库为只读状态。

由于在备份过程中，有可能备份某个表后，其他表发生的相应的变化，如果同意更新后就会造成**数据不一致**的问题。

1. **表锁（MyISAM的默认锁）**

* 读锁（read lock），也叫共享锁（shared lock），针对同一份数据，多个读操作可以同时进行而不会互相影响（select）。
* 写锁（write lock），也叫排他锁（exclusive lock），当前操作没完成之前，会阻塞其它读和写操作（update、insert、delete）。

特点：

1. 对整张表加锁 2. 开销小 3. 加锁快 4. 无死锁 5. 锁粒度大，发生锁冲突概率大，并发性低

**3. 行锁（InnoDB的默认锁）**

* 读锁（read lock），也叫共享锁（shared lock），允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。
* 写锁（write lock），也叫排他锁（exclusive lock），允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享锁和排他锁。
* 意向共享锁（IS），一个事务给一个数据行加共享锁时，必须先获得表的IS锁。
* 意向排它锁（IX），一个事务给一个数据行加排他锁时，必须先获得该表的IX锁。

特点：

1. 对一行数据加锁 2. 开销大 3. 加锁慢 4. 会出现死锁 5. 锁粒度小，发生锁冲突概率最低，并发性高

## 悲观锁和乐观锁

悲观锁：基于一种悲观的态度来防止一切数据冲突，每次数据修改之前会把数据锁住，然后对数据进行读写。

乐观锁：对数据冲突保持一种乐观的态度，操作数据时不会对操作的数据进行加锁，多个任务可以并行的对数据进行操作，只有数据提交的时候才通过一种机制来验证数据是否存在冲突（一般实现方式是通过加版本号，进行版本号的对比）

## 公平锁和非公平锁

公平锁：多个线程按照申请锁的顺序去获得锁，线程会直接进入队列去排队，永远都是队列的第一位才能得到锁。

优点：所有的线程都能得到资源，不会饿死在队列中。

缺点：吞吐量会下降很多，队列里面除了第一个线程，其他的线程都会阻塞，cpu唤醒阻塞线程的开销会很大。

非公平锁：多个线程去获取锁的时候，会直接去尝试获取，获取不到，再去进入等待队列，如果能获取到，就直接获取到锁。

优点：可以减少CPU唤醒线程的开销，整体的吞吐效率会高点，CPU也不必取唤醒所有线程，会减少唤起线程的数量。

缺点：这样可能导致队列中间的线程一直获取不到锁或者长时间获取不到锁，导致饿死。

## InnoDB行锁的实现方式

MySQL InnoDB 行锁是通过给索引上的索引项加锁来实现的，因此只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁。

## MySQL幻读现象（间隙锁）

幻读指的是一个事务在前后两次查询同一个范围的时候，后一次查询看到了前一次查询没有看到的行，幻读专指“新插入的行”。对MySQl来说，可重复读在更新或者插入的时候存在“当前读”的行为，可能出现读取某个key不存在，但是插入失败的情况，这就是幻读导致的。

出现幻读的原因是因为，行锁只能锁住行，对不存在的行数据是没办法上锁的，所以行锁没办法阻止其他线程的插入数据行为。为了解决幻读，innoDB引入了间隙锁（Gap Lock），就是锁住两个值之间的间隙。间隙锁和行锁合称next-key lock，每个next-key lock是前开后闭区间。

间隙锁之间不存在冲突问题，间隙锁可以解决幻读问题，但是也带来了一些困扰，间隙锁锁住了更大的范围，这是会影响并发度的，甚至可能造成死锁。例如两个事务，同时对某个间隙上锁，然后插入数据，并发情况下可能会导致相互等待死锁。

**Warning：**

RR虽然避免了幻读问题，但是毕竟不能保证完全的隔离。

如果在事务中第一次读取采用快照读，第二次读取采用当前读，则如果在两次读取之间数据发生了变化，两次读取到的结果不一样，因为第一次的快照读并没有加锁。

## MySQL的Next-Key Lock

InnoDB支持三种行锁定方式：

* 行锁（Record Lock）：锁直接加在索引记录上面（无索引项时演变成表锁）。
* 间隙锁（Gap Lock）：锁定索引记录间隙，确保索引记录的间隙不变。间隙锁是针对事务隔离级别为可重复读或以上级别的。
* Next-Key Lock ：行锁和间隙锁组合起来就是 Next-Key Lock。

innoDB默认的隔离级别是可重复读(Repeatable Read)，并且会以Next-Key Lock的方式对数据行进行加锁。Next-Key Lock是行锁和间隙锁的组合，当InnoDB扫描索引记录的时候，会首先对索引记录加上行锁（Record Lock），再对索引记录两边的间隙加上间隙锁（Gap Lock）。加上间隙锁之后，其他事务就不能在这个间隙修改或者插入记录。

当查询的索引含有唯一属性（唯一索引，主键索引）时，Innodb存储引擎会对next-key lock进行优化，将其降为record lock,即仅锁住索引本身，而不是范围。

## InnoDB何时用行锁，何时用间隙锁

* 只使用唯一索引查询，并且只锁定一条记录时，innoDB会使用行锁。
* 只使用唯一索引查询，但是检索条件是范围检索，或者是唯一检索然而检索结果不存在（试图锁住不存在的数据）时，会产生 Next-Key Lock。
* 使用普通索引检索时，不管是何种查询，只要加锁，都会产生间隙锁。
* 同时使用唯一索引和普通索引时，由于数据行是优先根据普通索引排序，再根据唯一索引排序，所以也会产生间隙锁。

## MySQL发生死锁怎么办

MySQL发生死锁时候，应对有两种策略，一种策略是，直接进入等待，直到超时，这个超时时间可以通过参数innodb\_lock\_wait\_timeout来设置。另一种策略是，发起死锁检测，发现死锁后，主动回滚死锁链条中的某一个事务，让其他事务得以继续执行。这种方式是将innodb的死锁检测参数innodb\_deadlock\_detect打开，设置成on。

## 高并发条件下如何尽量避免死锁

（1）合理的设计索引，区分度高的列放到组合索引前面，使业务 SQL 尽可能通过索引定位更少的行，减少锁竞争。

（2）调整业务逻辑 SQL 执行顺序， 避免 update/delete 长时间持有锁的 SQL 在事务前面。

（3）避免大事务，尽量将大事务拆成多个小事务来处理，小事务发生锁冲突的几率也更小。

（4）以固定的顺序访问表和行。比如两个更新数据的事务，事务 A 更新数据的顺序为 1，2;事务 B 更新数据的顺序为 2，1。这样更可能会造成死锁。

（5）在并发比较高的系统中，不要显式加锁，特别是是在事务里显式加锁。如 select … for update 语句，如果是在事务里（运行了 start transaction 或设置了autocommit 等于0）,那么就会锁定所查找到的记录。

（6）尽量按主键/索引去查找记录，范围查找增加了锁冲突的可能性，也不要利用数据库做一些额外额度计算工作。比如有的程序会用到 “select … where … order by rand();”这样的语句，由于类似这样的语句用不到索引，因此将导致整个表的数据都被锁住。

（7）优化 SQL 和表设计，减少同时占用太多资源的情况。比如说，减少连接的表，将复杂 SQL 分解为多个简单的 SQL。

## --------------------------主从部分及切分--------------------------

## MySQL的主从复制过程

备库B跟主库A之间维持了一个长连接。主库A内部有一个线程，专门用于服务备库B的这个长连接。一个事务日志同步的完整过程是这样的：

* 在备库B上通过change master命令，设置主库A的IP、端口、用户名、密码，以及要从哪个位置开始请求binlog，这个位置包含文件名和日志偏移量。
* 在备库B上执行start slave命令，这时候备库会启动两个线程，就是io\_thread和sql\_thread。其中io\_thread负责与主库建立连接。
* 主库A校验完用户名、密码后，开始按照备库B传过来的位置，从本地读取binlog，发给B。
* 备库B拿到binlog后，写到本地文件，称为中转日志（relay log）。
* sql\_thread读取中转日志，解析出日志里的命令，并执行。

## MySQL强同步、异步同步、半同步（主从复制方式）

强同步复制：当主库执行完一个事务，所有的**从库都执行了该事务**才返回给客户端。因为需要等待所有从库执行完该事务才能返回，所以全同步复制的性能必然会收到严重的影响。

**除非是对所有从库数据一致性要求非常高的场景，否则一般不采用这种策略。**

异步同步复制：MySQL默认的复制即是异步的，主库在执行完客户端提交的事务后会立即将结果返给给客户端，并不关心从库是否已经接收并处理，这样就会有一个问题，主如果crash掉了，此时主上已经提交的事务可能并没有传到从库上，如果此时，强行将从提升为主，可能导致新主上的数据不完整。

**异步同步性能最好，适用于高性能且数据一致性要求不是很强的场景。**

半同步复制：是介于强同步复制与异步复制之间的一种，主库只需要等待至少一个从库节点收到**并且 Flush Binlog 到 Relay Log 文件即可**，主库不需要等待所有从库给主库反馈。同时，这里只是一个收到的反馈，而不是已经完全完成并且提交的反馈，如此，节省了很多时间。

**对于数据一致性要求高的场景，可以采用半同步复制的同步策略，比如主库挂掉时，准备接管的那一个从库，对数据的一致性要求比较高。**

增强的半同步复制：在半同步复制情况下如果有这样一个情况:当主库提交完事务之后，就开始等待从库的relay写入返回，如果从库这个时候发生了意外，导致主库等待超时，这个时候主库就会直接结束等待了，因为事务已经提交了，用户肯定是可以看到没有被同步的数据，为了解决这个问题，mysql5.7引入了增强半同步复制: 只有当收到至少一个从库返回的relay log写入确认后，才提交事务，也就是说提交事务是在收到确认之后进行的。

## MySQL基于GTID的主备复制方式

GTID表示**全局事务ID**，在主库上生成与事务绑定的唯一标识，其在MySQL集群中是唯一的。

GTID是在MySQL 5.6.5 中新增的一种主从复制方式。通过 GTID 保证了每个在主库上提交的事务在集群中有一个唯一的ID。这种方式主要是强化了数据库的主备一致性，故障恢复以及容错能力。

优点：

* 由于在主从复制中，由于Master 的dump进程一边要发送binlog给Slave，一边要等待Slave的ACK消息，这个过程是串行的，就是说前一个事物的ACK没有收到消息，那么后一个事物只能排队候着； 这样将会极大地影响性能；有了GTID后，SLAVE就直接可以通过数据流获得GTID信息，无需每次等待ack回应。
* 另外，主从故障切换中，如果一台MASTER down，需要提取拥有最新日志的SLAVE做MASTER，这个是很好判断，而有了GTID，就只要以GTID为准即可方便判断；而有了GTID后，SLAVE就不需要一直保存这bin-log 的文件名和Position（在change mater部分包含的）了。

缺点：

* 主从库的表存储引擎必须是一致的，如果主从库的表存储引擎不一致，就会导致数据不一致，例如一个是事务存储引擎，一个是非事务存储引擎，则会导致事务和 GTID 之间一对一的关系被破坏，结果就会导致基于 GTID 的复制不能正确运行；
* 不允许一个SQL同时更新一个事务引擎和非事务引擎的表；
* 对于create temporary table 和 drop temporary table语句不支持;
* 在一个复制组中，必须要求统一开启GTID或是关闭GTID;

## MySQL的主备延迟

有时候备库的性能没有主库的好，或者有时候多个主库的备库放到了一台机器上，所以当备库机器上竞争资源时，就可能会导致主备延迟了。（因此一般是对称部署的）

备库的压力太大，如果备库上收到了太多的读请求，导致其查询耗费了大量的CPU资源影响了同步速度，造成主备延迟。（解决办法一般是用一主多从的结构，可以采用多个备库分担查询压力）

还要一种情况是大事务问题，如果一个主库上的语句执行10分钟，那么这个事务可能导致从库备份延迟10分钟，例如我们一般不能delete太多数据，需要分成多次删除。

## MySQL主备延迟太长的解决方式（备库并行复制）

如果sql\_thread按照单线程的方式来同步relay log，那么可能它的sql执行速度可能并没有io\_thread复制快，所以有可能会导致延迟时间太长，因此从库一般会使用多个worker并行复制加速relay log的执行速度，但是并行复制有两个限制：**更新同一行的两个事务，必须分发到一个worker中；同一个事务不能被拆开，必须放到同一个事务中。**

1. MySQL5.5版本不支持并行复制，但是业界提出了两种策略：

* **按表分发策略（按照库名+表名构建哈希表，相同的在一个worker中）：**
* 如果事务涉及的表和所有worker里的表都不冲突，分配给最闲的worker。
* 如果和多于一个workder冲突，进入等待。
* 如果只和一个worker冲突，分配给存在冲突的worker。
* **按行分发策略（按照库名+表名+主键id构建哈希表，相同的在一个worker中）：**
* 如果两个事务没有更新相同的行，在备库上可以并行执行。
* 要求binlog格式必须是row。
* 相比按表分发，要消耗更多的计算资源。

1. MySQL5.6版本的按库并行：

* 对于可以按表分发的场景，可以通过将表迁到不同的库，来应用此策略，有可操作性。
* 速度更快，因为hash key就一个库名。

1. MariaDB：

* MariaDB采用了redo log的组提交优化来模拟主库的并行模式，每一组相同的事务有一个相同的commit\_id，commit\_id直接写到binlog里面，备库中相同commit\_id的事务分发到多个worker执行，全部执行完再取下一批。
* 但是这样并没有很高效地进行并发，因为在第一个事务的commit阶段和第二个running阶段是可以并行的，所以MySQL5.7针对这点进行了优化。

1. MySQL5.7版本优化：

* 同时处于prepare状态的事务，在备库执行时是可以并行的；
* 处于prepare状态的事务，与处于commit状态的事务之间，在备库执行时也是可以并行的。

## MySQL主备切换的策略

1. **可靠性优先策略：**

* 判断备库B现在的seconds\_behind\_master（主备延迟时间），如果小于某个值继续下一步，否则持续重试这一步；
* 把主库A改成只读状态，即把readonly设置为true；
* 判断备库B的seconds\_behind\_master的值，直到这个值变成0为止；
* 把备库B改成可读写状态，也就是把readonly 设置为false；
* 把业务请求切到备库B。

1. **可用性优先策略：**

* 如果我们先将备库B改为可读写状态然后将业务请求切换过去，这样做就是保证了可用性优先，但是这样可能会数据不一致的问题。

因此我们一般是会使用可靠性优先策略。

## 数据库的水平切分和垂直切分

**垂直切分**：垂直切分是根据业务来拆分数据库，同一类业务的数据表拆分到一个独立的数据库，另一类的数据表拆分到其他数据库。

垂直切分可以降低单节点数据库的负载。原来所有数据表都放在一个数据库节点上，因此所有的读写请求也都发到这个MySQL上面，所以数据库的负载太高。如果把一个节点的数据库拆分成多个MySQL数据库，这样就可以有效的降低每个MySQL数据库的负载。

但是垂直切分无法解决表记录太多的情况，因此使用**水平切分**进行“缩表”。

**水平切分**：水平切分是按照某个字段的某种规则，把数据切分到多张数据表，进行一个数据分片，把一张大的数据表拆分成多张小的数据表，这样就可以起到缩表的效果了。（注意：其实MySQL水平拆分出来的数据表也可以放在同一个MySQL结点上，因为MySQL自带一种数据分区的技术，可以把一张表按照特殊的规则切分存储在不同的目录下，因此我们可以利用MySQL分区技术将一张表的数据切分存储在不同的磁盘上，这样就能有效提高数据库的效率）。

## 为什么先做水平切分在做垂直切分？

随着数据的增加，最先做的优化应该是**数据分片**，利用多块硬盘加大数据IO能力和存储空间，这样的成本是比较低的，只需要将多个分片挂在一个MySQL结点上即可。但是随着数据的持续增大，这时候我们要把数据切分到多个MySQL结点上，同时业务方面也需要加上负载均衡、分布式架构等。

接着往后走的话，其实数据量已经不是更改架构的主要目的了，而是这个业务系统承受不住了，因此可以针对模块对业务系统进行拆分，若干个子系统之间数据相对独立，此时做的就是垂直拆分操作。

因此整个过程水平切分实现起来比较简单且高效，垂直拆分涉及到业务系统拆分，工作量比较大，因此要先从水平切分做起。

## 分库分表情况下的id设置

采用uuid算法：UUID代表全局唯一标识 ID，由于其全局唯一性，所以能够把它用来作为数据库分库分表的主键。

UUID = 12位时间+10位的IP地址+6位生成序列号+最后4位的分库和分表id。

在存储时间时，UUID 是根据时间位逆序存储，也就是低时间低位存放在最前面，高时间位在最后，UUID 的前 4 个字节会随着时间的变化而不断“随机”变化，并非单调递增。在插入时会产生离散 IO，从而产生性能瓶颈。为了解决这个问题，MySQL8.0推出了UUID\_TO\_BIN函数：

* **能够把UUID字符串的时间高位放在前面，解决了UUID插入乱序的问题；**
* **去掉了无用的字符串"-"，精简存储空间；**

**将字符串其转换为二进制值存储，空间最终从之前的 36 个字节缩短为了 16 字节。**

## 分库分表下如何做数据库查询

一般数据量大不建议联表，尤其是交易订单这种。可以在每个库分别查询，然后再进行综合处理。

如果业务上有需要的话，可以在分表的时候将字段冗余一下，把有可能需要的信息也存下来；如果没办法处理的话也可以用服务组合需要关联的数据，监视数据变化，将数据拿到进行组合放入 nosql 内存数据库，查询时直接从nosql 内存数据库查询。

所以当进行了分库分表后我们一般不会直接进行联表查询。

## -------------------------MySQL调优、设计------------------------

## 数据库的RBO（基于规则优化）和CBO（基于代价优化）

**基于规则的优化**就是当数据库执行一条query语句的时候必须遵循预先定义好的一系列规则来确定执行过程，它不关心访问表的数据分布情况，而是凭借规则经验来确定，所以说是一种比较粗放的优化策略。

**基于代价的优化**的产生就是为了解决上面RBO的弊端，让执行引擎依据预先存储到数据库中表的一些实时更新的统计信息来选择出最优代价最小的**执行计划**来执行query语句，CBO会根据统计信息来生成一组可能被使用到的执行计划，进而估算出每个计划的代价，从而选择出代价最小的交给执行器去执行，其中表的统计信息一般会有表大小，行数，单行长度，单列数据分布情况，索引情况等等。

## MySQL慢查询解析

1. SQL 没加索引，通常，慢查询都是因为没有加索引。如果没有加索引的话，会导致全表扫描的。因此，应考虑在 where 的条件列，建立索引，尽量避免全表扫描。
2. 长事务太多，导致查询的undo log回滚长度太长，要尽量减少长事务的个数。
3. SQL 索引不生效，比如优化器选错啦索引或者查询条件包含 “or”，可能导致索引失效。
4. join 或者子查询过多。
5. 数据库在刷脏页，例如redo log 写满了，要刷脏页，这时候会导致系统所有的更新堵住，写性能都跌为 0 了，一般要杜绝出现这个情况。

## MySQL执行计划

执行计划是指一条 SQL 语句在经过 MySQL 查询优化器的优化会后，具体的执行方式。MySQL 提供了 EXPLAIN 语句，来获取执行计划的相关信息，这里的EXPLAIN 语句并不会真的去执行相关的语句，而是通过查询优化器对语句进行分析，找出最优的查询方案，并显示对应的信息，例如type（一般到ref就比较好了）、key、possible\_key等。



## MySQL数据库性能优化过程

* 使用Explain进行分析，查看其使用的索引、扫描行数以及查询类型等。
* 优化数据访问，要减少数据的访问量，限制输出必要的行（limit语句）和列，针对重复查询的数据，可以对其增加缓存。
* 使用索引优化查询，如增加覆盖索引或建立对应的二级索引减少回表次数防止全表扫描。
* 重构查询方式：一个大查询如果要一次性执行的话可能会锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但重要的查询。
* 针对多表查询可以分解为多个单表查询然后在应用程序中进行关联，这样做的好处是：
  + 让缓存更高效如果连接的其中一个表发生变化则整个缓存查询就无法使用，进行分解后即使其中某个表发生了变化，对其他表的查询缓存依然有效。
  + 减少单表的锁竞争。

## 常见的系统性能调优策略

1. 代码：代码上要去检查是否使用了合适的数据结构，使用火焰图等工具查看其性能瓶颈，保证代码的性能后再去处理其他问题。
2. 数据库调优：

SQL调优：可以使用慢查询日志来定位到出问题的SQL，通过explain、profile等工具进行调优。

架构层面调优：读写分离、主从库负载均衡、水平和垂直分库分表等。

连接池调优：根据具体的连接池监控数据以及当前的业务量做出对应的判断进行调优。

1. 缓存：

如果数据量小，并且不会频繁地增长清空，可以考虑本地缓存，否则可以考虑redis等缓存服务。

考虑缓存穿透、缓存击穿、缓存雪崩等问题处理。

1. 异步：

针对客户端的某些请求，在服务端可能要针对这些请求做一些附属的事情，而并不关心得到的结果，可以选择异步的方式去进行处理这些事情。

常见做法：额外**开辟新的线程或者采用线程池**的策略让处理请求的线程先返回结果，其他线程异步地进行处理。或者可以采用**消息队列**的思想，一些额外任务可能不需要这个系统来处理，可以将其封装成一个消息丢到消息队列里面。

## 数据库设计建表需要注意什么

（1）面向对象原则，要保证表中的每列与主键相关。

（2）表职能单一性原则，如果一张表负责了两个或两个以上的职责，那么该表应进行拆分。

（3）表字段与表直接关联原则，如果一个字段与当前表是间接关联的，那么就该创建一张新的表来保存该字段。

（4）字段最小原子化原则，一个字段如果包含了多个信息或含义，则该字段就应该拆成多个字段。

（5）字段名唯一原则，所有字段不允许单个单词形式存在，例：机构信息表code改为org\_code，name改为org\_name。好处：避免一些容易出现的SQL错误；方便代码检索；减少部分SQL中输出字段的别名设置；看到字段名或属性名就能知道其含义。

（6）不创建外键关联，只是外键关系，设计中的外键仅是为了让大家知道其对应的表才写进去的，实际并不需要在数据库中创建外键。目的是为了确保表之间的独立性。

（7）尽量少使用存储过程，除非某个功能对性能要求非常高，且没办法通过存储过程以外的方式来提升性能时，才考虑使用存储过程。

（8）应针对所有表的主键和外键建立索引，有针对性的建立组合属性的索引，提高检索效率。

（9）必须建立业务主键，除自增id之外，需要使用业务主键，常使用”业务前缀+时间戳+随机数”组成，优点：实现简单，与数据库无关，移植性较好。

（10）所有字段不允许空值，所有字段不允许空值，varchar类型默认统一设置为”Empty String”，此处不使用NULL。区别：NULL，默认存为(Null)，指的是对象；Empty String默认存为空串””，在java中进行非空判断存在区别userId==null/userId.equals(“”)；数值类型默认为0

（11）表中字段根据数据类型统一，数值类：bigint，金额：decimal；状态：tinyint （从数据库底层来说是以二进制方式传输，此处存字符运行效率较高varchar(2)—了解），时间：datetime

（12）每张表需要建立索引，建立索引方式：选取使用频率最高的字段作为索引字段。

## -------------------------其他问题补充问题-------------------------

## SQL注入攻击

SQL 注入就是在用户输入的字符串中加入 SQL 语句，如果在设计不良的程序中忽略了检查，那么这些注入进去的 SQL 语句就会被数据库服务器误认为是正常的 SQL 语句而运行，攻击者就可以执行计划外的命令或访问未被授权的数据。因此有必要从开发、测试、上线等各个环节对其进行防范。

SQL的注入原理：

1. 恶意拼接查询

SELECT \* FROM users WHERE user\_id = $user\_id,如果传入参数是“1234; DELETE FROM users”得到SQL语句就会变成“SELECT \* FROM users WHERE user\_id = 1234; DELETE FROM users”.

1. 利用注释执行非法命令，比如在传入的参数包含SLEEP(500)将导致SQL语句一直运行如果其中添加了修改、删除数据的恶意指令，那么将会造成更大的破坏。
2. 传入非法参数。SQL 语句中传入的字符串参数是用单引号引起来的，如果字符串本身包含单引号而没有被处理，那么可能会篡改原本 SQL 语句的作用。
3. 添加额外条件，在 SQL 语句中添加一些额外条件，以此来改变执行行为。条件一般为真值表达式。如在传入的参数“1234 OR TRUE”，这样在执行更新的时候这个条件始终是真值，于是满足条件的记录如果执行删除或者更新等操作就会出现问题。

避免SQL注入

1. 过滤输入内容，校验字符串。过滤输入内容就是在数据提交到数据库之前，就把用户输入中的不合法字符剔除掉。如果值属于特定的类型或有具体的格式，那么在拼接 SQL 语句之前就要进行校验，验证其有效性。比如对于某个传入的值，如果可以确定是整型，则要判断它是否为整型，在浏览器端（客户端）和服务器端都需要进行验证。
2. 参数化查询。参数化查询目前被视作是预防 SQL 注入攻击最有效的方法。参数化查询是指在设计与数据库连接并访问数据时，在需要填入数值或数据的地方，使用参数来给值。MySQL 的参数格式是以“?”字符加上参数名称而成，UPDATE myTable SET c1 = ?c1, c2 = ?c2, c3 = ?c3 WHERE c4 = ?c4。在使用参数化查询的情况下，数据库服务器不会将参数的内容视为 SQL 语句的一部分来进行处理，而是在数据库完成 SQL 语句的编译之后，才套用参数运行。因此就算参数中含有破坏性的指令，也不会被数据库所运行。
3. 除了开发规范，还需要合适的工具来确保代码的安全。我们应该在开发过程中应对代码进行审查，在测试环节使用工具进行扫描，上线后定期扫描安全漏洞。通过多个环节的检查，一般是可以避免 SQL 注入的。

常见开发过程中避免SQL注入的方法

1. 避免使用动态SQL
2. 不要将敏感数据保留在纯文本中，加密存储在数据库中的私有/机密数据，这样可以提供了另一级保护，以防攻击者成功地排出敏感数据。
3. 避免直接向用户显示数据库错误。攻击者可以使用这些错误消息来获取有关数据库的信息

## 数据库join的过程

JOIN（联结）的本质就是把各个联结表中的记录取出来依次匹配的组合放入结果集，在MySQL 8.0版本之前这个JOIN的实现过程实际上是一个笛卡尔的过程，也就是一个嵌套循环的过程，即驱动表会逐行去扫描被驱动表中符合条件的结果集，直到拿出所有的答案。这个过程中，驱动表只需要访问一次，被驱动表可能会被多次访问，MySQL 8.0版本之后采用哈希原理实现联结，即Hash Join。

根据IO的开销，INNODB对嵌套循环的过程进行了优化，根据不同的优化手段Join有三种方式：Basic Nested-Loop Join（基础嵌套循环Join）、Block Nested-Loop Join（基于块的嵌套循环Join）以及Index Nested-Loop Join（基于索引的嵌套循环Join）

表之间的普通嵌套循环查询的过程就是Basic Nested-Loop Join，这个过程中驱动表的一条数据的一次比较过程需要把被驱动表中满足条件的数据都读入内存一次再进行比对，进行了n方的查找，但是当表特别大的时候是不行的，IO开销太大，这个时候可以加一个join buffer，缓存多条驱动表数据，一次进行多条数据的匹配，这个过程其实就是优化了对驱动表读取的**IO次数**，这个算法依赖buffer的大小，如果足够大，可以一下子缓存驱动表中的数据，这个时候就只需要读取一次被驱动表。

除了优化被驱动表的扫描次数之外，还可以对表的读取做出优化，查询被驱动表的过程实际上就是一次单表扫描，所以借助索引可以加快查询的过程，通过驱动表的匹配条件直接与被驱动表索引进行匹配，减少了对被驱动表的IO匹配次数。原来的匹配次数为驱动表行数×被驱动表行数，借助索引之后匹配次数为驱动表的行数×被驱动表索引树的高度。

INNODB中的Hash Join？

以多表等值查询举例。Hash Join利用Hash原理，加速了联结查询，匹配过程的时间复杂度又O(n^2)优化到了接近O(1)。实现过程如下：

（1）首先有一个build的过程，该过程就是建立散列表，对于两个表来说优化器会使用两个表中较小的表作为驱动表，在内存中建立hash表。

（2）第二个阶段就是probe过程，在这个阶段扫描被驱动表，每读出一行数据，就从散列表中做一个对应查询，但是这个时候并不确定这两行就是满足条件的数据，需要在再进行一个条件过滤。

（Hash Join面临的问题）散列表实际上也是放在Join buffer中的（嵌套循环Join中用来缓存多条驱动表数据的缓存区域），但是如果散列表太大，这个缓冲区装不下怎么办？

Hash join针对这个问题，采用了分治的思想，借助磁盘来进行过程优化，对两个表按照一个特定的哈希函数进行分片，写入对应相同的磁盘空间上，然后对同一磁盘空间中的数据进行hash join。

## 数据库连接池

数据库连接是一种关键的有限的昂贵的资源，一个数据库连接对象均对应一个物理数据库连接，每次操作都打开一个物理连接，使用完都关闭连接，这样造成系统的性能浪费。

[数据库连接池](https://so.csdn.net/so/search?q=%E6%95%B0%E6%8D%AE%E5%BA%93%E8%BF%9E%E6%8E%A5%E6%B1%A0&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Sundy_sc/article/details/_blank)的解决方案是在应用程序启动时建立足够的数据库连接，并将这些连接组成一个连接池，由应用程序动态地对池中的连接进行申请、使用和释放，它允许应用程序重复使用一个现有的数据库连接，而不是再重新建立一个。

优点：

较少了网络开销  
 系统的性能会有一个实质的提升  
 没了麻烦的TIME\_WAIT状态

常见的连接池：**DBCP连接池（JAVA系）**

## 缓存一致性问题

强一致性保证写入后立即可以读取，弱一致性则不保证立即可以读取写入后的值，而是尽可能的保证在经过一定时间后可以读取到，在弱一致性中应用最为广泛的模型则是最终一致性模型，即保证在一定时间之后写入和读取达到一致的状态。对于应用缓存的大部分场景来说，追求的则是最终一致性，少部分对数据一致性要求极高的场景则会追求强一致性。

针对缓存和数据库一致性的问题一般有四种策略：

1. 先更新数据库，再更新缓存：

这种方式一般适合更新缓存的代价很小的数据，再多线程并发的情况下容易引发线程不安全问题，例如两个线程要进行更新操作：

线程A更新了数据库；线程B更新了数据库；线程B更新了缓存；线程A更新了缓存。

这就出现请求A更新缓存应该比请求B更新缓存早才对，但是因为网络等原因，B却比A更早更新了缓存，这就导致了脏数据。

1. 先更新缓存，再更新数据库：

这种情况和第一种一样会存在线程安全问题。

1. 先删除缓存，再更新数据库：

这种方式可能不会因更新数据库出现线程不安全，但是也可能会导致脏读问题，例如：线程A进行写操作，删除缓存，线程B查询发现缓存不存在，线程B去数据库查询得到旧值，线程B将旧值写入缓存，线程A将新值写入数据库。这时候就出现了脏读问题。

1. 先更新数据库，再删除缓存

这种情况出现脏读的情况要更低，因为这个条件需要发生在读缓存时缓存失效，而且并发着有一个写操作。而实际上数据库的写操作会比读操作慢得多，而且还要锁表，而读操作必需在写操作前进入数据库操作，而又要晚于写操作删除缓存，所有的这些条件都具备的概率基本并不大，但是还是会有出现的概率。

**一般我们会使用延时双删策略：**

先删除(淘汰)缓存；再写数据库（这两步和原来一样）；休眠1秒，再次删除(淘汰)缓存。并且为了性能更快，可以把第二次删除缓存可以做成**异步**的，这样不会阻塞请求了，而且为了防止第二次删除缓存失败，这个异步删除缓存可以加上重试机制，失败一直重试，直到成功（将第二次删除的key放入消息队列）。

**为什么要两次删除？可以将前面第一次删除造成的脏数据进行再次删除，保证数据一致性。**

**这个1秒应该看具体的的业务场景，应该去评估下项目的读数据业务逻辑的耗时，然后写数据的休眠时间则在读数据业务逻辑的耗时基础上，加几百ms即可，这么做确保读请求结束，写请求可以删除读请求造成的缓存脏数据。**

# Redis

## Redis的数据类型及应用

1. **String(字符串、整数或浮点数)**：做简单的键值对缓存。采用SDS类型进行存储，和C 语言字符串比较相近，但是又在字符串长度、内存重分配次数方面做了对应的优化。

**SDS的内容？**buf柔性数组，真正存储字符串的数据空间；len表示buf中已占用字节数；alloc表示buf中已分配字节数，记录的是为buf分配的总长度；flags标识当前结构体的类型，低3位用作标识位，高5位预留。

**何时发生扩容？**SDS在涉及字符串修改时会调用sdsMakeroomFor函数进行检查，会根据空闲长度和新增内容的长度进行比较判断，然后根据不同情况动态扩容。

String如何保证二进制安全？SDS对象中的buf是一个柔性数组，上层调用时，SDS直接返回了buf。由于buf是直接指向内容的指针，所以兼容C语言函数。而当真正读取内容时，SDS会通过len来限制读取长度，而非“\0”，所以保证了二进制安全。

1. **List(列表)**：存储一些表型数据结构，例如商品的评论列表等。采用双向链表的形式进行存储。

**当列表对象所有的字符串元素长度都小于64字节并且元素数量小于512时会使用ziplist（原理在下面）；当不满足上述条件会转换为双向链表。**

1. **Set(无序集合)**：用于存储、添加或者检查某个元素是否在对应的集合中，也可以做并、交、差集操作。

**当列表对象所有的元素都是整数并且元素数量小于512时会使用ziplist（原理在下面）；当不满足上述条件会转换为哈希表（类似unordered\_map）。**

1. **Hash(包含键值对的无序散列表)**：添加、获取单个键值对，类似结构化数据，比如对象。

**当列表对象所有的元素长度都小于64字节并且元素数量小于512时会使用ziplist（原理在下面）；当不满足上述条件会转换为哈希表（类似unordered\_map）。**

1. **Zset(有序集合)**：根据范围来获取元素，计算值的排名，进行去重及排序操作。跳表代替平衡树，实现起来较方便一些。

跳表：为了存储多个不同行的数据，我们可以用链表将他们串起来，这样查询数据的时间复杂度是O（n），于是我们可以将链表中的部分结点拿出来构建出一个新的链表放在其上层，根据这种思路依次以二分的形式构建上层链表，查询的时候可以根据一层一层的查询实现类似二分的查找，能够有效提高检索速度，跳表通过牺牲空间换取时间的方式提升查询性能，查询的时间复杂度变为log（n）。跳表的插入是当插入最底层的链表后，是否需要在上面几层插入索引靠随机函数，理论上为了达到二分的效果，每一层的结点数是下一层的一半，因此也是以二分之一的概率依次向下进行插入，因此每次插入数据后会以随机函数的方式随机其插入的索引层数，当数据足够多的时候，数据的分布也符合了我们的“二分”要求。

**当列表对象所有的元素长度都小于64字节并且元素数量小于128时会使用ziplist（原理在下面）；当不满足上述条件会转换为哈希表（类似unordered\_map）。**

zipList（压缩列表）原理：是为了节约内存而开发的。和各种语言的数组类似，它是由连续的内存块组成的，这样一来，由于内存是连续的，就减少了很多内存碎片和指针的内存占用，进而节约了内存。

哈希表（类似c++的unordered\_map）

## Redis高可用策略

1. 持久化：

主要作用是将数据存储到硬盘，保证数据不会因为进程退出而丢失。Redis持久化主要有两种方式：RDB和AOF；RDB是Redis的默认存储方式，在指定的时间内将内存中的数据集快照写入磁盘，下次将磁盘中的数据读入内存即可。Redis 会单独创建（fork）一个子进程来进行持久化，首先会将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何 IO 操作的，这就确保了极高的性能。AOF是以日志的形式来记录每个写操作（增量保存），将Redis执行过的所有写指令记录下来(读操作不记录)，只许追加文件但不可以改写文件，redis启动之初会读取该文件重新构建数据。

1. 主从复制（读写分离）：

哨兵和集群都是在复制基础上实现高可用的。复制主要实现了数据的多机备份，以及读操作的负载均衡和简单的故障恢复。在redis主从架构中，Master（主）节点负责处理写请求，Slave（从）节点只处理读请求。对于写请求少，读请求多的场景，通过这种读写分离的操作可以大幅提高并发量。主从之间的同步有增量同步、快照同步以及无盘复制的优化。

1. 哨兵机制：

哨兵是一个独立的进程，它会在后台独立运行，其原理是哨兵通过发送命令，等待Redis服务器的响应，从而对多个Redis进行监控。当哨兵监测到master宕机，会自动将slave切换成master，然后通过订阅模式通知其他服务器，修改配置文件，让他们切换主机。

然而当只有一个哨兵对Redis进行监控的时候，可能会出现问题（进程挂了怎么办），所以可以采用多个哨兵进行监控，并且各个哨兵之间还会进行监控，形成多哨兵模式。

1. 分布式集群：

集群解决了Redis写操作无法负载均衡，以及存储能力受到单机限制的问题，实现了较为完善的高可用方案。Redis cluster中有16384个槽（序号从0~16383），集群中的每个主节点控制一定数量的槽（一般是平均分配）。以集群中有三台主节点为例：第一台控制前三分之一，第二台控制三分之一、第三台控制剩下的三分之一，每个节点维护部份槽及槽所映射的键值数据。当我们向Redis服务发送请求时，内部会经过哈希算法和取模运算将key固定的映射到某个槽中，而这个槽由某台主节点负责，这样就能从对应的Redis服务中读写数据了。

## Redis为何这么快

（1）完全基于内存，数据存在内存中，绝大部分请求是纯粹的内存操作，非常快速，跟传统的磁盘文件数据存储相比，避免了通过磁盘IO读取到内存这部分的开销。

（2）数据结构简单，对数据操作也简单。Redis中的数据结构是专门进行设计的，每种数据结构都有一种或多种数据结构来支持。Redis正是依赖这些灵活的数据结构，来提升读取和写入的性能。

（3）采用单线程，省去了很多上下文切换的时间以及CPU消耗，不存在竞争条件，不用去考虑各种锁的问题，不存在加锁释放锁操作，也不会出现死锁而导致的性能消耗。

（4）使用基于IO多路复用机制的线程模型，可以处理并发的链接。

## Redis6.0的多线程引入是否有线程不安全问题？

因为Redis的瓶颈不在内存，而是在网络I/O模块带来CPU的耗时，所以Redis6.0的多线程是用来处理网络I/O这部分，充分利用CPU资源，减少网络I/O阻塞带来的性能损耗。

在redis的多线程模式下，获取、解析命令，以及输出结果着两个过程，可以配置成多线程执行的，因为它毕竟是我们定位到的主要耗时点，但是命令的执行，也就是内存操作，依然是单线程运行的。所以，Redis 的多线程部分只是用来处理网络数据的读写和协议解析，执行命令仍然是单线程顺序执行，也就不存在并发安全问题。

## Redis为什么使用跳表而不使用B+树

Redis是纯内存的数据库，进行读写数据都是操作内存，跟磁盘没啥关系，因此也不存在磁盘IO了，所以层高就不再是跳表的劣势了。

而且B+树插入数据时是有一些数据页合并拆分操作的，换成红黑树或者其他AVL树的话也是各种旋转。

而跳表插入数据时，只需要随机一下，就知道自己要不要往上加索引，根本不用考虑前后结点的感受，也就少了旋转平衡的开销。

## Redis的大Key和热Key问题及其处理

1. **大Key和热Key问题是什么？**

**大Key**通常以Key的大小和Key中成员的数量来综合判定，**一般会导致客户端执行命令的时长变慢**，例如：

* Key本身的数据量 过大：一个String类型的Key，它的值为5 MB。
* Key中的成员数过多：一个ZSET类型的Key，它的成员数量为10,000个。
* Key中成员的数据量过大：一个Hash类型的Key，它的成员数量虽然只有1,000个但这些成员的Value（值）总大小为100 MB。

**热Key**通常以其接收到的Key被请求频率来判定，一般**占用大量的CPU资源，影响其他请求并导致整体性能降低，**例如：

* QPS集中在特定的Key：Redis其中一个Key的QPS达到总QPS的百分之60。
* 带宽使用率集中在特定的Key：对一个拥有很多成员的HASH Key每秒发送大量的HGETALL（返回所有的哈希表字段和值）操作请求。
* CPU使用时间占比集中在特定的Key：对一个拥有很多成员的Key（ZSET类型）每秒发送大量的ZRANGE（返回区间成员）操作请求。

1. **大Key问题的查找办法**

* Redis4.0之前可以采用redis-rdb-tools工具。相当于在redis实例上执行bgsave，然后对dump出来的rdb文件进行分析，找到其中的大KEY；或者自定义扫描脚本去找。
* Redis4.0之后memory usage命令和lazyfree机制：memory usage默认抽样5个field来循环累加计算整个key的内存大小，样本的数量决定了key的内存大小的准确性和计算成本，样本越大，循环次数越多，计算结果更精确，性能消耗也越多；而Redis中的lazyfree可以将删除键或数据库的操作放在后台线程里执行， 从而尽可能地避免服务器阻塞，lazyfree的原理其实就是在删除对象时只是进行逻 辑删除，然后把对象丢给后台，让后台线程去执行真正的destruct，避免由于对象体积过大而造成阻塞。

1. **大Key和热Key的解决**

* 对大Key进行合理的拆分，可以将很多成员的一个Hash Key拆分为多个Hash Key，在Redis集群架构中拆分大的Key能对数据分片间的内存平衡起到显著作用。
* 对大Key进行清理，将不适合放入Redis中的大Key数据放到其他存储中。
* 对过期数据进行定期清理，在Redis的set、Hash中如果有大量的过期数据，也会造成大Key问题。
* 对热Key问题可以采用读写分离的架构。

## Redis查询速度慢怎么定位问题

慢查询只记录命令的执行时间,并不包括命令排队和网络传输时间.因此客户端执行命令的时间会大于命令的实际执行时间.因为命令执行排队机制,慢查询会导致其他命令级联阻塞,因此客户端出现请求超时时,需要检查该时间点是否有对应的慢查询,从而分析是否为慢查询导致的命令级联阻塞。

Redis 执行命令分为四个步骤：发送命令、命令排队、执行命令、返回结果。需要注意的是，慢查询只统计执行命令 的时间,所以没有慢查询并不代表客户端没有超时问题。

Redis 慢查询有两个参数需要配置：slowlog-log-slower-than：设置慢查询预设的超时阈值，单位是微秒，slowlog-max-len：表示慢查询日志存储的条数。

slowlog-log-slower-than 表示的是慢查询预设的超时阈值。它所阐述的意思是如果某条命令（如 key \*） 执行”很慢“，执行时间超过了设置的阈值，那么这条命令将会被记录到慢查询日志中。

若设置 slowlog-log-slower-than = 0，则会记录所有命令，

若设置 slowlog-log-slower-than < 0，则不会记录任何命令

slowlog-log-slower-than默认是10毫秒，我们需要根据 Redis 的并发量调整该值。若我们采用默认，那么命令的执行时间要超过 10 毫米才会记录，如果命令的执行时间超过 10 毫秒，那么 Redis 的 QPS 连 100 都不到，因此对于高并发场景，建议将该值设置较小。

此外Redis 会使用一个列表来存储慢查询日志，slowlog-max-len 就是该列表的最大长度。一个命令如果满足慢查询阈值条件则会加入到该列表来，但是如果该列表已经处于最大长度时，那么会删除最开始的一条记录，然后将最新的命令插入到末尾，所以慢查询日志列表是一个有限的先进先出列表。

slowlog-max-len: 线上建议调大慢查询列表,记录慢查询时Redis会对长命令做阶段操作,并不会占用大量内存.增大慢查询列表可以减缓慢查询被剔除的可能,例如线上可设置为1000以上。

获取慢查询日志使用slowlog get [n]，n表示展示的条数，慢查询日志由4个属性组成（1）日志的标识id（2）发生的时间戳（3）命令的耗时（4）执行的命令和参数。获取慢查询列表长度slowlog len，清空日志列表slowlog reset。

## Redis过期策略和淘汰机制

Redis采用了定期删除+惰性删除的方式。

**为何不用定时删除？**使用定时删除的话需要用一个定时器来负责监视key，过期则自动删除，这样虽然释放的很及时，但是会很消耗CPU资源，而CPU应该用在处理请求上，而不是删除检查上。

**定期删除的思想：**Redis默认经过100ms进行检查，是否有过期的key还未被删除，如果有的话进行删除。Redis并不是100ms对所有的Key进行一次检查而是**抽取检查**。

**惰性删除的思想：**当在获取某个key的时候，Redis会对这个key进行检查其是否过期，如果过期则删除。

因此有时候会出现已经到了过期时间，而Redis内存占用率还是很高的情况，就是未及时删除导致的。

但是采用定期删除+过期删除的思想并不能保证没用到的数据得到了及时的删除，因此Redis还采用了**内存淘汰机制（一般是LRU）。**

**内存淘汰机制（内存页面置换算法）：**

* volatile-lru，针对设置了过期时间的key，使用lru算法进行淘汰。
* allkeys-lru，针对所有key使用lru算法进行淘汰。
* volatile-lfu，针对设置了过期时间的key，使用lfu算法进行淘汰。
* allkeys-lfu，针对所有key使用lfu算法进行淘汰。
* volatile-random，从所有设置了过期时间的key中使用随机淘汰的方式进行淘汰。
* allkeys-random，针对所有的key使用随机淘汰机制进行淘汰。
* volatile-ttl，针对设置了过期时间的key，越早过期的越先被淘汰。
* noeviction，不会淘汰任何数据，当使用的内存空间超过 maxmemory 值时，再有写请求来时返回错误。

## Redis的渐进式哈希

Redis和unordered\_map相同，采用了哈希进行保存，因此哈希表超过最大装载因子后会发生Rehash的过程，这是一个渐进式哈希的过程，目的就是将原哈希表中的桶依次放入新桶中（Redis维护了两个哈希表，每次移动时为另一个哈希表分配空间，在依次转移即可）。

## Redis事务的特性

Redis中事务是**以队列形式序列化、按顺序地执行**，分为两个阶段分别是**Multi组队阶段、Exec执行阶段**。组队阶段中如果有命令错误会导致队伍取消，执行阶段如果有命令错误会跳过该命令继续执行。

Redis事务的三个特性：

1. 单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
2. 没有隔离级别的概念：队列中的命令没有提交之前都不会实际被执行，因为事务提交前任何指令都不会被实际执行。
3. 不保证原子性：事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚 。

## Redis锁机制

为了避免秒杀场景的多余扣费问题，Redis采用了锁机制（乐观锁）

1. 悲观锁：每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到锁被释放。 传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。
2. 乐观锁：就是对操作持乐观态度，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量。

## Redis持久化

### RDB

RDB是Redis的默认存储方式，在指定的时间内将内存中的数据集快照写入磁盘，下次将磁盘中的数据读入内存即可。RDB文件时一个经过压缩的**二进制文件**，由多个部分组成；对于不同类型的键值对，RDB文件会使用不同方式保存。

1. **RDB的执行过程：**

Redis 会单独创建（fork）一个子进程来进行持久化，首先会将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何 IO 操作的，这就确保了极高的性能，因为Redis是使用操作系统的多进程写时复制机制来实现快照的持久化，在持久化过程中调用glibc(Linux下的C函数库)的函数fork()产生一个子进程，快照持久化完全交给子进程来处理，父进程继续处理客户端的读写请求。子进程刚刚产生时，和父进程共享内存里面的代码段和数据段，此时父子进程的虚拟空间不同，但其对应的物理空间（内存区）是同一个。这是Linux操作系统的机制，为了节约内存资源，所以尽可能让父子进程共享内存，这样在进程分离的一瞬间，内存的增长几乎没有明显变化。

1. **RDB的触发条件：**

* 符合自定义配置的快照规则；
* 执行save或者bgsave命令（save直观保存，全部阻塞；bgsave仅阻塞主进程fork子进程的时候）；
* 执行flush all命令（执行 flush all 命令，也会产生 dump.rdb 文件，但里面是空的，无意义）；
* 执行主从复制操作 (第一次)。

1. **RDB的优缺点**

优点：

（1）适合大规模的数据恢复

（2）对数据完整性和一致性要求不高更适合使用（快照形式，不保证后续更新跟进）。

（3）节省磁盘空间（RDB二进制压缩文件，占用空间小）

（4）恢复速度快

缺点：

（1）父进程在fork子进程的时候如果主进程比较大会阻塞。

（2）子进程写操作的时候，会对主进程数据进行拷贝（相当于内存膨胀了），需要考虑内存。

（3）无法完全保证数据的一致性和完整性。

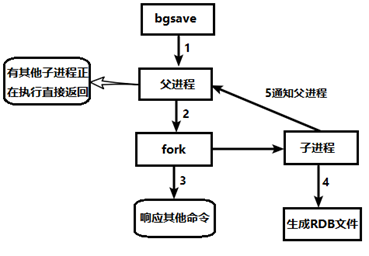
1. **Fork子进程**

Fork 的作用是复制一个与当前进程一样的进程。新进程的所有数据（变量、环境变量、程序计数器等）数值都和原进程一致，但是是一个全新的进程，并作为原进程的子进程。

1. **子进程的写时复制机制**

在 Linux 系统中，调用 fork 系统调用创建子进程时，并不会把父进程所有占用的内存页复制一份，而是与父进程共用相同的内存页，而当子进程或者父进程对内存页进行修改时才会进行复制，这就是写时复制机制。

1. **RDB执行过程如下图所示：**



### 2.AOF

以日志的形式来记录每个写操作（增量保存），将Redis执行过的所有写指令记录下来(读操作不记录)，只许追加文件但不可以改写文件，redis启动之初会读取该文件重新构建数据。AOF文件中的所有命令都是以**Redis命令请求协议的格式**保存的，命令请求会先保存到AOF缓冲区中，之后再定期写入并同步到AOF文件。

1. **AOF的持久化流程**

* 客户端的请求操作会append写入AOF的缓存中。
* AOF缓存区根据采用的持久化策略将操作sync同步到磁盘的AOF文件中。
* AOF的大小超过重写策略或者手动重写时，会对AOF文件进行重写，压缩AOF文件容量。
* Redis恢复的时候会加载AOF文件中的写操作达到数据恢复的目的。

1. **持久化策略的种类**

⦁ always：始终同步，每次 Redis 的写入都会立刻记入日志；性能较差但数据完整性比较好。

⦁ everysec：每秒同步，每秒记入日志一次，如果宕机，本秒的数据可能丢失。

⦁ no：redis 不主动进行同步，把同步时机交给操作系统（Redis 被关闭、AOF 功能被关闭 、 系统的写缓存被刷新（可能是缓存已经被写满，或者定期保存操作被执行））。

1. **AOF优缺点：**

优点：

⦁ 备份机制更稳健，丢失数据概率更低。

⦁ 可读的日志文本，通过操作AOF文件，可以处理误操作。

缺点：

⦁ 比起 RDB 占用更多的磁盘空间（没有进行二进制压缩）。

⦁ 恢复备份速度要慢（需要按照命令进行恢复）。

⦁ 每次读写都同步的话，有一定的性能压力

1. **AOF重写机制**

Redis可以在 AOF体积变得过大时，自动地在后台（Fork子进程）对 AOF进行重写。重写后的新 AOF文件包含了恢复当前数据集所需的最小命令集合。 AOF 重写并不需要对原有的AOF文件进行任何写入和读取，它针对的是Redis中键的当前值。

由于Redis将子进程放在后台执行，所以主进程可以继续处理用户的请求，同时子进程带有主进程的数据副本，使用子进程而不是线程，可以在避免锁的情况下，保证数据的安全性。

但是有个显著的问题：**子进程在AOF重写的时候，主进程会继续执行命令，因此可能会进行数据修改，这会让当前Redis数据和重写后的AOF文件中的数据不一致**。

为了解决这个问题，Redis增加了一个AOF重写缓存，这个缓存在fork出子进程之后开始启用，Redis主进程在接到新的写命令之后，除了会将这个写命令的协议内容追加到现有的AOF文件之外，还会追加到这个缓存中。

当子进程完成 AOF 重写之后，它会向父进程发送一个完成信号，父进程在接到完成信号之后，会调用一个信号处理函数，并完成以下工作：

（1）将AOF重写缓存中的内容全部写入到新AOF文件中；

（2）对新的AOF文件进行改名，覆盖原有的AOF文件；

这个信号处理函数执行完毕之后，主进程就可以继续像往常一样接受命令请求了。在整个AOF后台重写过程中，只有最后的写入缓存和改名操作会造成主进程阻塞，在其他时候，AOF后台重写都不会对主进程造成阻塞，这将AOF重写对性能造成的影响降到了最低。

1. **AOF重写流程**

（1）bgrewriteaof 触发重写，判断是否当前有 bgsave 或 bgrewriteaof 在运行，如果有，则等待该命令结束后再继续执行；

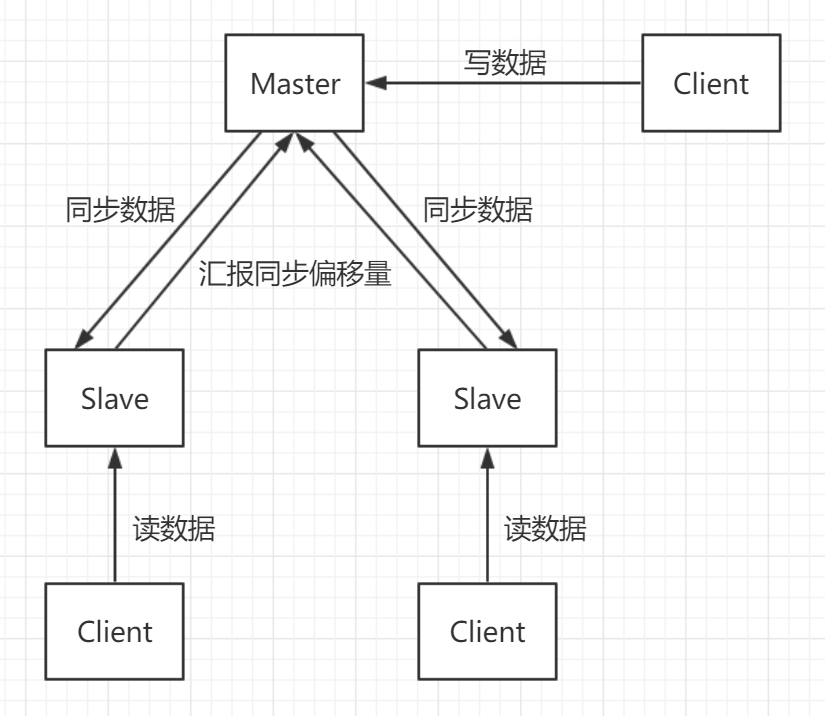
（2）主进程 fork 出子进程执行重写操作；

（3）子进程遍历 redis 内存中数据到临时文件，客户端的写请求同时写入 aof\_buf 缓冲区和 aof\_rewrite\_buf 重写缓冲区，保证原 AOF 文件完整以及新 AOF 文件生成期间的新的数据修改动作不会丢失；

（4）子进程写完新的 AOF 文件后，向主进程发信号，父进程更新统计信息。主进程把 aof\_rewrite\_buf 中的数据写入到新的 AOF 文件；

（5）使用新的 AOF 文件覆盖旧的 AOF 文件，完成 AOF 重写。

### 3.Redis主从架构



#### Redis主从架构原理：

**读写分离：**在redis主从架构中，**Master（主）节点负责处理写请求，Slave（从）节点只处理读请求**。对于写请求少，读请求多的场景，通过这种读写分离的操作可以大幅提高并发量。

**主从同步：**Master节点接收到写请求并处理后，需要告知Slave节点数据发生了改变，**保持主从节点数据一致的行为称为主从同步**，所有的Slave都和Master通信去同步数据也会加大Master节点的负担，实际上，除了主从同步，**redis也可以从从同步。**

1. **增量同步：**在增量同步的过程中，主结点会将那些对自己产生影响的指令存到本地内存的buffer中，然后通过**异步**的方式同步到从结点中。从结点一边会对过来的指令流进行同步，一边会对主节点反馈自己的同步进度。从结点同步的同时**不会影响自己的读服务**，它会用旧数据进行查询，当同步完成后，需要删除旧数据加载新数据，这个时候才会暂停对外服务。

由于**内存是有限的**，所以Redis主节点并不能将所有的指令都记录在内存中，而是采用环形结构进行保存，满了即覆盖。

1. 快照同步：如果增量同步的时候由于结点间通信不好，导致从结点同步的速度不如主节点接收新请求的速度，buffer中会丢失一部分指令，从结点和主结点之间就会发生快照同步，

快照同步是一个极其消耗资源的操作，它首先需要在主结点上进行一次bgsave将内存中的数据全部快照到RDB文件中，然后将快照文件发送到从结点中。从结点接收到RDB文件后立即进行一次全量加载，加载完毕后通知主结点继续进行增量同步。

但是在快照同步的过程中主结点的复制buffer还在不停的复制，如果快照同步时间过长或者增量buffer长度不够，又会导致再次快照复制。所以需要配置一个合适的buffer长度，避免陷入快照死循环。

1. 无盘复制：主节点在进行快照同步的时候，会进行大量的IO操作，这样会导致主结点的服务效率降低。

Redis2.8.18版本开始支持无盘复制。无盘复制表示主节点会一边遍历内存，一边将序列化的内容发送到从结点，而不是生成完整的RDB文件才进行发送，而从结点和原先一样，先将收到的内容保存到磁盘上，再进行一次性加载。

#### 故障恢复的过程

当一个master宕机后，后面的slave可以立刻升级为master，而哨兵模式能够监视主机是否有故障，如果发生故障则根据投票将从库转换为主库。

哨兵是一个独立的进程，它会在后台独立运行，其原理是哨兵通过发送命令，等待Redis服务器的响应，从而对多个Redis进行监控。当哨兵监测到master宕机，会自动将slave切换成master，然后通过订阅模式通知其他服务器，修改配置文件，让他们切换主机。

然而当只有一个哨兵对Redis进行监控的时候，可能会出现问题（进程挂了怎么办），所以可以采用多个哨兵进行监控，并且各个哨兵之间还会进行监控，形成多哨兵模式。

假设主服务器发生了宕机，哨兵1先检测到这个结果，但是不会立马进行master切换，仅仅是哨兵1认为发生了故障，该情况被称为主观下线。当其他哨兵也检测到主服务器故障后，就会进行投票选取一个哨兵进行故障切换，切换成功后就会发布订阅模式，这个过程称为客观下线。

#### 哨兵模式

当一个master宕机后，后面的slave可以立刻升级为master，而哨兵模式能够监视主机是否有故障，如果发生故障则根据投票将从库转换为主库。

哨兵是一个独立的进程，它会在后台独立运行，其原理是哨兵通过发送命令，等待Redis服务器的响应，从而对多个Redis进行监控。当哨兵监测到master宕机，会自动将slave切换成master，然后通过订阅模式通知其他服务器，修改配置文件，让他们切换主机。

然而当只有一个哨兵对Redis进行监控的时候，可能会出现问题（进程挂了怎么办），所以可以采用多个哨兵进行监控，并且各个哨兵之间还会进行监控，形成多哨兵模式。

#### 哨兵选举算法

1. 哨兵的三个定时任务：

* 每个哨兵每10秒会向主结点发送info命令获取最新的拓扑结构图（Redis的主从结构图），当有新的从结点加入后就可以立马感知。
* 每个哨兵结点每隔两秒会对Redis数据结点的指定频道上发送该哨兵对主结点的判断以及当前哨兵结点信息，同时每个哨兵也会订阅该频道，来了解其他哨兵信息及对主结点的判断。
* 每隔一秒会对主结点、从结点、其他哨兵发送ping命令进行心跳检测。

1. 主观下线及客观下线：
2. 选举Leader哨兵，来进行故障转移：

**哨兵选举采用的是Raft算法介绍：**

Raft算法是一个用户管理日志一致性的协议，它将分布式一致性问题分成多个子问题：Leader选举、日志复制、安全性、日志压缩等。

1. **raft角色介绍：**

Leader角色：接受客户端请求，并向Follower同步请求日志，当日志同步到大多数结点后告诉Follower进行提交日志。

Follower角色：接受并持久化Leader 同步的日志，在Leader告知可提交后进行日志提交。

Candidate：Leader选举过程中的临时角色。

1. **任期：**

在分布式系统中，各个结点的时间同步是一个很大的难题，为了识别过期时间，raft协议引入了term（任期）的概念。

Raft将时间划分为任意不同长度的任期，每个任期的开始都是一次选举，一个或者多个候选人会试图成为Leader，但是某些情况下选票被瓜分，没有选出领导人，那么会开始下一个任期，并且立即开始下一次选举。Raft保证在一次任期内最多只有一个Leader。

1. **Leader哨兵选举流程：**

⦁ 每个做主观下线的哨兵向其他哨兵结点发送命令，该哨兵会先看看自己有没有投过票，如果自己已经投过票给其他哨兵了，在2倍故障转移的超时时间（默认故障转移时间为3分钟）自己就不会成为Leader，否则要求将自己设置为领导者。

⦁ 如果该哨兵发现自己的票数已经超过一半并且超过了quorum（配置文件中的一个阈值）那么它成为Leader。

⦁ 如果在一个任期中没能选出Leader，那么等2倍故障转移超时时间后，Candidate增加epoch重新投票。

#### 主结点选取

* 优先选取slave-priority最高的结点（Redis.conf设置，默认为100）。
* 选择最高优先级中复制偏移量最大的从机。
* 如果还没选出来，则按ID进行排序选择一个最小的从结点。

## Redis分片集群

### Redis集群设计思想

**采用主从架构（读写分离，通过增加Slave可以提高并发读的能力）+哈希分片（将Redis的写操作分摊到了多个节点上，提高写的并发能力，扩容简单）的思想：**

**读写分离：**在redis主从架构中，**Master（主）节点负责处理写请求，Slave（从）节点只处理读请求**。对于写请求少，读请求多的场景，通过这种读写分离的操作可以大幅提高并发量。

**主从同步：**Master节点接收到写请求并处理后，需要告知Slave节点数据发生了改变，**保持主从节点数据一致的行为称为主从同步**，所有的Slave都和Master通信去同步数据也会加大Master节点的负担，实际上，除了主从同步，**redis也可以从从同步。**

Redis集群会采用数据分片来实现，一个Redis集群包含16384个哈希槽，数据库中每个键都属于16384中的一个，采用CRC16（key）%16384来得到key属于哪个槽。

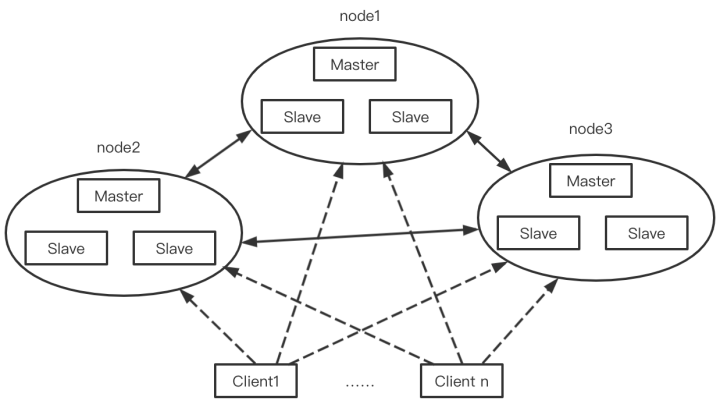
主从和哈希的设计正好是相互弥补的，先Hash分逻辑节点，然后每个逻辑节点内部是主从，想扩展并发读就添加Slave，想扩展并发写就添加Master，想扩容也就是添加Master，任何一个Slave或者几个Master挂了都不会是灾难性的故障。

### Redis集群介绍

Redis集群实现了对Redis的水平扩容，即启动n个结点，将整个数据库分布在这n个结点中；并且Redis采用了去中心化的思想，没有中心结点的说法，因此客户端可以连接任意一个结点进行操作。

Redis集群的特点：分治+分片。

Redis集群可以看成多个主从架构组合起来的，每一个主从架构可以看成一个节点（其中，只有master节点具有处理请求的能力，slave节点主要是用于节点的高可用）



### 数据如何进行分片

Redis集群采用数据分片来实现，一个Redis集群包含16384个哈希槽，数据库中每个键都属于16384中的一个，采用CRC16（key）%16384来得到key属于哪个槽。集群中的每个结点可以复制一部分的哈希槽，例如假设存在3个结点：

⦁ 节点 A 负责处理 0 号至 55xx 号哈希槽。

⦁ 节点 B 负责处理 55xx +1号至 11000 号哈希槽。

⦁ 节点 C 负责处理 11001 号至 16384 号哈希槽。

哈希槽数据分区算法具有以下几种特点：

⦁ 解耦数据和节点之间的关系，简化了扩容和收缩难度；

⦁ 节点自身维护槽的映射关系，不需要客户端代理服务维护槽分区元数据

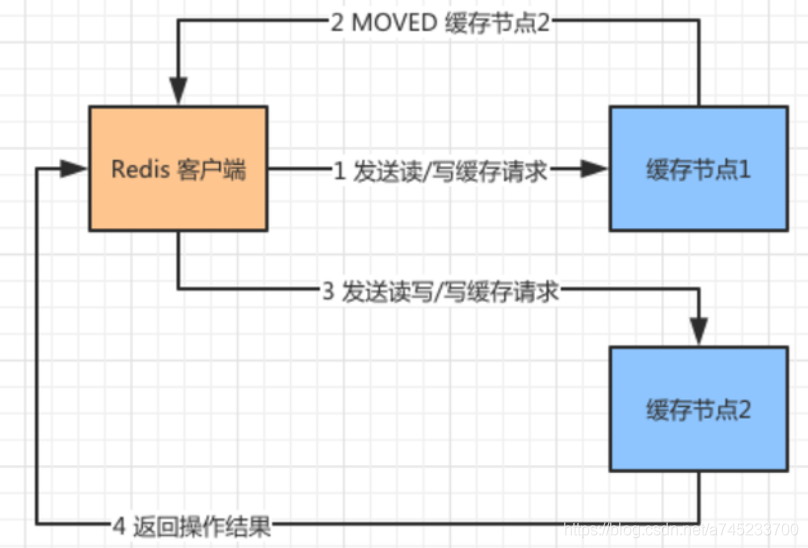
⦁ 支持节点、槽、键之间的映射查询，用于数据路由，在线伸缩等场景

默认情况下，Redis集群的读和写都是在master上执行的，不支持slave结点的读和写，跟Redis单机的主从复制是不一样的，因为Redis集群的核心理念，主要是用slave做数据的热备，以及master故障时的主备切换来实现高可用的。Redis单机的读写分离是为了横向扩展slave结点去支撑更大的读吞吐量。而Redis集群的架构下，本身master就是可以任意扩展的，如果想要支撑更大的读和写的吞吐量，都可以直接对master进行横向扩展。

### Redis集群的请求重定向：

1. **MOVED重定向请求**

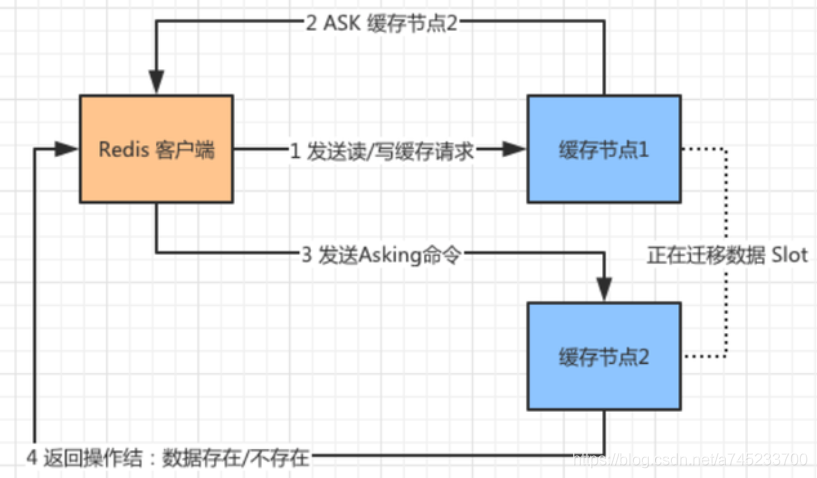
Redis集群在客户端层面没有采用代理，并且无论Redis的客户端访问集群中的哪个结点，都能重定位到相应的结点上。



当客户端通过CRC16（key）%16384计算出哈希槽的idx发现需要到“结点1”进行数据操作时，由于数据迁移的原因该槽被转移到“结点2”中，此时由于结点1中保存了所有集群中的结点信息，因此它知道这个槽在“结点2”中保存，因此向Redis客户端发送了一个MOVEN的重定向请求，这个请求告诉我们要访问“结点2”，Redis拿到这个地址后重新进行访问。

1. **ASK请求**

如果当结点1正在向结点2进行数据迁移的时候，此时客户端发送查询后，会如何处理呢？



其实就是一个ASK重定向以及后续的Asking命令，客户端就会收到数据是否存在的结果。然后再次进行查询即可。

1. **频繁重定向造成的网络开销：smart客户端**

由于客户端的频繁重定向会增加集群的网络负担和单次查询的耗时，因此大部分客户端在本地维护了一个映射表的缓存，大部分情况下走本地缓存就能查询到对应的结果。（数据映射关系更新的时候本地缓存需要同步更新，smart客户端从编程语言上支持）

### Redis集群中结点的通信机制：gossip协议

当集群的状态发生变化时，比如新结点的加入、结点宕机、slave提升为Master等，Redis集群的不同结点之间通过gossip协议进行通信，结点之间的通信是为了维护结点之间的元数据信息，最终达到数据的一致性。

Redis集群中的每个结点都维护一份自己视角下的集群信息，包括：

* 当前集群状态；
* 集群中各节点的master-slave状态；
* 集群中各节点的存活状态及怀疑Fail状态。

Goosip协议的原理就是不同的结点不停地通信交换信息，一段时间后，所有的结点就有了整个集群的完整信息，并且所有结点的状态都能达到一致。每个结点可能知道全部结点信息，也可能只知道部分结点信息，但是通过网络连通，最终他们的状态是一致的。通信过程如下所述：

1. 集群中每个结点都会单独开一个TCP通道，用于结点间的彼此通信。
2. 每个结点在固定的周期会通过特定的规则选择几个结点发送ping消息。
3. 接受到ping消息的结点用pong消息作为回应。

使用gossip协议的优点在于将元数据的更新分散在了不同的结点上（相当于逐渐的扩散），降低了压力；但是缺点就是元数据的更新有延时，可能导致集群中的一些操作有滞后。另外，gossip协议对服务器时间要求较高，时间戳不准确会影响结点判断消息的有效性。而且结点增多后网络的开销也会增加服务器的压力，同时达到最终一致性要求的时间也会加长，因此官方推荐的最大节点数为1000左右。

Redis cluster结构下每个Redis要开两个端口号，比如一个是6379，另一个就是加10000的16379.

* 6379就是Redis服务器的入口。
* 16379就是使用gossip协议（二进制协议）进行结点之间通信的端口号。

### Redis集群故障检测与故障恢复机制

Redis集群检测是基于gossip协议的，集群中的每个结点会定期向集群中的其他结点发送PING消息，交换各个结点之间的信息。

1. 主观下线：

当结点A检测到与结点B通讯时间超过了cluster-node-timeout的时候，就会更新本地结点状态，把结点B更新为主观下线。

注意：主观下线并不能代表某个节点真的下线了，有可能是节点A与节点B之间的网络断开了，但是其他的节点依旧可以和节点B进行通讯。

（2）客观下线：

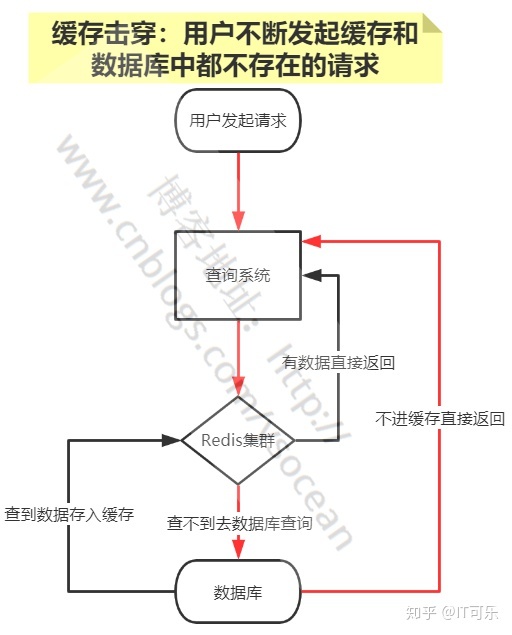
由于集群内的结点会不断进行通信，下线信息也会通过Gossip消息传遍其他结点，因此集群内的结点会不断收到下线报告。

当半数以上的结点标记了结点B是主观下线时，便会出发客观下线的流程。接着向集群广播一条主结点B的Fail消息，所有收到消息的结点都会标记B为客观下线。

## Redis实际应用问题（用作缓存）

### 缓存穿透

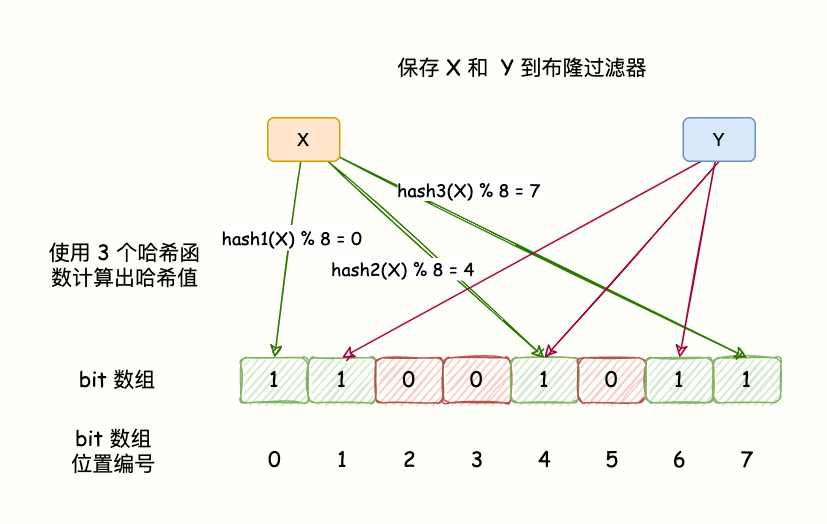
Redis缓存和数据库中都没有的数据，可用户还是源源不断的发起请求，导致每次查询都会发送到数据库，从而压垮数据库。



缓存穿透的时候，Redis在整个过程中保证平稳运行，但是数据库压力倍增可能会导致宕机。

解决方案：

1. 对空值设定短时间缓存，如果一个数据库查询返回的结果为空，我们仍然把这个空结果进行缓存，或者将其设置为默认值，设置空结果的过期时间一般会很短。
2. 业务逻辑前置校验，在业务请求入口处进行数据合法检验，检验请求参数是否合理，是否包含非法值、恶意请求等。
3. 采用布隆过滤器，它利用一个很长的二进制向量（位图）和一系列的随机映射函数，判断数据“一定不存在或者可能存在”，用户发送的请求会先被布隆过滤器拦截，一定不存在的数据就直接拦截返回，避免下一步对数据库的压力。
4. 进行实时监控，当发现Redis的命中率开始极速变低后，需要排查访问的对象和访问的数据，和运维人员配合，可以设置用户黑名单限制服务。



### 缓存击穿

Redis在一个热点key在失效的同时，大量的请求发过来，从而会全部打到数据库，压垮数据库。

解决方案：

1. 预先设置热门数据：在Redis高峰访问之前，把一些热门数据提前存入Redis里面，加大这些热门key的时长或者设置为不过期。
2. 实时调整：现场监控哪些数据热门，实时调整key的过期时长。
3. 使用互斥锁：多个请求在进行查询时，竞争一个锁，获得锁的请求从数据库中查询获取数据，并更新Redis缓存，若其他请求也在请求该key时，发现锁获取失败，则睡眠一段时间后再次重试即可。

### 缓存雪崩

Redis中缓存的数据大量失效或者Redis宕机时，导致大量请求直接到数据库，压垮数据库。

解决办法：

1. 设置有效期均匀分布：避免缓存设置相似的有效期，我们可以对有效期时间加上一个随机值，或者同一规划有效期，使得过期时间均匀分布。
2. 保证Redis的高可用：采用哨兵模式和集群模式，防止Redis集群单点故障。
3. 设置过期标志更新缓存：记录缓存数据是否过期，如果过期会触发另外的线程在后台更新实际的key缓存。

## Redis实现分布式锁

我们在系统中修改已有数据时，需要先读取，然后进行修改保存，此时很容易遇到并发问题。由于修改和保存不是原子操作，在并发场景下，部分对数据的操作可能会丢失。在单服务器系统我们常用本地锁来避免并发带来的问题，然而，当服务采用集群方式部署时，本地锁无法在多个服务器之间生效，这时候保证数据的一致性就需要分布式锁来实现。

Redis 锁主要利用 Redis 的 setnx 命令（原子性操作）。

* 加锁命令：SETNX key value，当键不存在时，对键进行设置操作并返回成功，否则返回失败。
* 解锁命令：DEL key，通过删除键值对释放锁，以便其他线程可以通过 SETNX 命令来获取锁。
* 锁超时：EXPIRE key timeout, 设置 key 的超时时间，以保证即使锁没有被显式释放，锁也可以在一定时间后自动释放，避免资源被永远锁住。

## Redis实现分布式锁遇到的问题

1. 死锁问题：

当某个进程使用SETNX申请锁后，可以DEL释放锁，但是假如他在执行过程中进程挂掉了，那这个锁就无法得到释放了，因此可以在加锁的时候加上过期时间防止死锁发生。

1. 由于占用锁时间较长，导致释放了其他进程的锁：

当进程A申请锁后，假如超过了过期时间它还没释放，则此时锁过期，进程B就申请锁成功，然后进程A释放锁，把进程B占用的这个锁给释放掉了。因此上锁和释放锁必须要原子执行，则此时必须采用Lua脚本，Redis操作Lua脚本时其他线程必须等待，直到这个 Lua 脚本处理完成，这样一来，GET + DEL 之间就不会插入其它命令了。

1. 过期时间评估问题：

过期时间太长则导致系统并发性下降，过期时间太短则导致有可能在使用共享变量的过程中就过期掉了，针对这个问题可以开一个守护进程，定时检测这个锁的失效时间，如果失效时间快到了但是进程还在使用共享资源则自动对锁进行“续期”。

1. Redis误删问题：

分布式锁的key设置为对应的方法名，value应该设置为一个指定的唯一ID，释放前获取这个值，判断是否是自己的锁避免误删。

假设业务逻辑的执行时间是7秒：（1）业务逻辑1没执行完，3秒后锁被自动释放。（2）业务逻辑2获取到锁，执行业务逻辑，3秒后锁被自动释放。（3）业务逻辑3获取到锁，执行业务逻辑。（4）业务逻辑1执行完成，开始调用del释放锁，这时释放的是业务逻辑3的锁，导致业务逻辑3的业务只执行1s就被别人释放。出现没有上锁的情况。

## Redis实现消息队列

1. **list：**使用lpush 和rpop，使用 List 实现消息队列的

* 优点：是消息可以被持久化，List 可以借助 Redis 本身的持久化功能，AOF 或者是 RDB 或混合持久化的方式，用于把数据保存至磁盘，这样当 Redis 重启之后，消息不会丢失。
* 缺点：消息不支持重复消费、没有按照主题订阅的功能、不支持消费消息确认等。

1. **zset：**用 zadd 和 zrangebyscore 来实现存入和读取消息的。和list差不多，但是由于ZSet 多了一个分值（score）属性，如用它来存储时间戳，以此来实现延迟消息队列等。
2. **发布订阅类型：**解决了支持多组消费者问题，

* 优点：支持发布/订阅，多组消费。
* 缺点：无法持久化保存消息，如果 Redis 服务器宕机或重启，那么所有的消息将会丢失；消费者下线后，再上线后，由于之前的消息没有存储，老消息都会丢失，只能接收新消息。消费者全下线后，没人消费，消息全都会丢失。

1. **Stream：**通过xadd和xreadgroup来实现消息的生产和消费，xack 和 xreadgroup 实现消息的应答与恢复。

**redis实现消息队列和消息队列中间件对比：**

1. **消息不丢：**

从生产者、消费者、中间件三个角度分析：

* 生产者丢失消息的情况，没成功发送就重发。发送出去来但是回报丢了，只能重试导致消息重复，因此消费者要考虑幂等性
* 消费者丢失消息：消费者宕机，取出没有回ack，因此redis的stream会通过xack返回确认消息。
* 中间丢失情况：redis的aof周期性刷盘，这个过程异步有丢失的风险；主从切换，从库没有同步完就被提成主库。

1. **消息积压：**

redis基于内存，存储有限，积压一定程度就会丢弃消息

1. **消息积压成本低。**

## 缓存一致性问题

强一致性保证写入后立即可以读取，弱一致性则不保证立即可以读取写入后的值，而是尽可能的保证在经过一定时间后可以读取到，在弱一致性中应用最为广泛的模型则是最终一致性模型，即保证在一定时间之后写入和读取达到一致的状态。对于应用缓存的大部分场景来说，追求的则是最终一致性，少部分对数据一致性要求极高的场景则会追求强一致性。

针对缓存和数据库一致性的问题一般有四种策略：

1. 先更新数据库，再更新缓存：

这种方式一般适合更新缓存的代价很小的数据，再多线程并发的情况下容易引发线程不安全问题，例如两个线程要进行更新操作：

线程A更新了数据库；线程B更新了数据库；线程B更新了缓存；线程A更新了缓存。

这就出现请求A更新缓存应该比请求B更新缓存早才对，但是因为网络等原因，B却比A更早更新了缓存，这就导致了脏数据。

1. 先更新缓存，再更新数据库：

这种情况和第一种一样会存在线程安全问题。

1. 先删除缓存，再更新数据库：

这种方式可能不会因更新数据库出现线程不安全，但是也可能会导致脏读问题，例如：线程A进行写操作，删除缓存，线程B查询发现缓存不存在，线程B去数据库查询得到旧值，线程B将旧值写入缓存，线程A将新值写入数据库。这时候就出现了脏读问题。

1. 先更新数据库，再删除缓存

这种情况出现脏读的情况要更低，因为这个条件需要发生在读缓存时缓存失效，而且并发着有一个写操作。而实际上数据库的写操作会比读操作慢得多，而且还要锁表，而读操作必需在写操作前进入数据库操作，而又要晚于写操作删除缓存，所有的这些条件都具备的概率基本并不大，但是还是会有出现的概率。

**延时双删策略：**

先删除(淘汰)缓存；再写数据库（这两步和原来一样）；休眠1秒，再次删除(淘汰)缓存。并且为了性能更快，可以把第二次删除缓存可以做成**异步**的，这样不会阻塞请求了，而且为了防止第二次删除缓存失败，这个异步删除缓存可以加上重试机制，失败一直重试，直到成功（将第二次删除的key放入消息队列）。

**为什么要两次删除？可以将前面第一次删除造成的脏数据进行再次删除，保证数据一致性。**

**这个1秒应该看具体的的业务场景，应该去评估下项目的读数据业务逻辑的耗时，然后写数据的休眠时间则在读数据业务逻辑的耗时基础上，加几百ms即可，这么做确保读请求结束，写请求可以删除读请求造成的缓存脏数据。**

# MQ

## MQ介绍

* Kafka：分布式的、分区的、多副本的日志提交服务，在高吞吐量场景下发挥较为出色、
* RocketMQ：低延迟、强一致、高性能、高可靠、万亿级容量和实时的可扩展性，在一些实时场景下发挥较为出色。

## MQ的优点

解耦：A系统发送个数据到BCD三个系统，接口调用发送，那如果E系统也要这个数据呢？那如果C系统现在不需要了呢？现在A系统又要发送第二种数据了呢？此时只需要增加一个MQ，A系统作为生产者放入队列中，其他系统作为消费者从队列中拿消息就行了。

异步：

削峰：多用户时间和少用户时间的处理。

## MQ的缺点

* 系统可用性降低：系统引入的外部依赖越多，越容易挂掉，本来你就是A系统调用BCD三个系统的接口就好了，人ABCD四个系统好好的，没啥问题，你偏加个MQ进来，万一MQ挂了咋整？MQ挂了，整套系统崩溃了，你不就完了么。
* 系统复杂性提高：硬生生加个MQ进来，你怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？
* 一致性问题：A系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是BCD三个系统那里，BD两个系统写库成功了，结果C系统写库失败了，你这数据就不一致了。

## Kafka的缺点

1. 运维成本高
2. 对于负载不均衡的场景，解决方案复杂
3. 没有自己的缓存，完全依赖Page Cache
4. Controller、Coordinator和Brocker在同一进程中，大量IO会造成其性能下降，任一角色出错可能会导致其他角色挂掉。

## Kafka重启操作

Kafka中的重启操作实质上是Brocker的重启，**某个Brocker下线重启后会选择相近的一个Brocker作为新的Leader，然后将原先的Brocker进行重启，重启完成后追赶新Leader的消息，达到最新同步后进行Leader切换。**（实质上就是一个一个Brocker的升级，而不是一次性全部升级）。

## Kafka生产者发送消息有哪些模式

**发后即忘**：它只管往 Kafka 里面发送消息，但是**不关心消息是否正确到达**，这种方式的效率最高，但是可靠性也最差，比如当发生某些不可充实异常的时候会造成消息的丢失。

**同步(sync)**：producer.send()返回一个Future对象并进行同步等待，就知道消息是否发送成功，**发送一条消息需要等上个消息发送成功后才可以继续发送**。

**异步(async)**：Kafka支持 producer.send() 传入一个**回调函数**，消息不管成功或者失败都会调用这个回调函数，这样就算是异步发送，我们也知道消息的发送情况，然后再回调函数中选择记录日志还是重试都取决于调用方。

## 发送消息的分区策略有哪些

* **轮询：依次发送消息**到所有的Partition中。
* **key指定分区**：通过哈希函数指定每个key发送到哪个partition中，缺点是当partition增加的时候，就很难保证整个关系了。
* **自定义策略**
* **指定Partition发送**

## Kafka如何保证数据顺序性

**全局有序性**：由于Kafka的一个Topic可以分为了多个Partition，Producer发送消息的时候，是分散在不同 Partition的。当Producer按顺序发消息给Broker，但进入Kafka之后，这些消息就不一定进到哪个Partition，会导致顺序是乱的。因此要满足全局有序，需要1个Topic只能对应1个Partition。

**局部有序性**：要满足局部有序，只需要在发消息的时候指定Partition Key，Kafka对其进行Hash计算，根据计算结果决定放入哪个Partition。这样Partition Key相同的消息会放在同一个Partition。此时，Partition的数量仍然可以设置多个，提升Topic的整体吞吐量。

## Kafka如何实现负载均衡?

Kafka 的负责均衡主要是**通过分区来实现**的，Kafka 采用的是**主写主读**的架构，**每个 broker 都有消费者拉取消息，每个 broker 也都有生产者发送消息，每个 broker 上的读写负载都是一样的**，broker中每个Partition有自己对应的副本，因此通过**主读主写**的方式实现负载均衡。

## Kafka的可靠性保证

1. acks：这个参数用来指定分片中有多少个副本收到消息后，生产者才认为此消息写成功。

* ack=1，默认为1，生产者发送消息，**只要 leader 副本成功写入消息，就代表成功**。这就表示当leader写入还没来得及同步就宕机，会造成数据丢失。
* ack=0，生产者发送消息后直接算写入成功，不需要等待响应。这个方案的问题很明显，**只要服务端写消息时出现任何问题，都会导致消息丢失**。
* acks = -1 或 acks = all。生产者发送消息后，需要等待 ISR 中的所有副本都成功写入消息后才能收到服务端的响应。这种情况的安全性是最高的。

1. 消息发送的方式：

* 可以通过同步或者异步来获取响应结果，采用**失败或重做**保证消息的可靠性。

1. 手动提交位移：

* 默认情况下，当消费者消费到消息后，就会**自动提交位移**。但是**如果消费者消费出错**，没有进入真正的业务处理，那么就可能会**导致这条消息消费失败**，从而丢失。我们可以开启手动提交位移，等待业务正常处理完成后，再提交offset。

## Kafka如何避免数据丢失

1. 必须是已提交的消息。kafka对于已经提交的消息（committed message）的定义是，生产者提交消息到broker，并等到多个broker确认并返回给生产者已提交的确认信息。而这多个broker是由我们自己来定义的，可以选择只要有一个broker成功保存该消息就算是已提交，也可以是令所有broker都成功保存该消息才算是已提交。不论哪种情况，kafka只对已提交的消息做持久化保证。
2. 要求保存消息的N个broker中，至少有1个存活。只要这个条件成立，kafka就能保证你的这条消息永远不会丢失。

## Kafka 的消息消费方式有哪些

1. **点对点形式**：当所有消费者属于同一个消费组，那么所有的消息都会被均匀的投递给每一个消费者，**每条消息只会被其中一个消费者消费**。
2. **发布订阅**:如果所有消费者属于不同的消费组，那么所有的消息都会被投递给每一个消费者，**每个消费者都会收到该消息**。

## Kafka重复消费情况

1. 消费者宕机、重启或者被强行kill进程，导致消费者消费的offset没有提交。
2. 设置自动提交后，如果在关闭消费者进程之前，取消了消费者的订阅，则有可能部分offset没提交，下次重启会重复消费。
3. 消费后的数据，当offset还没有提交时，Partition就断开连接。比如，通常会遇到消费的数据，处理很耗时，导致超过了Kafka的session timeout.ms时间，那么就会触发reblance重平衡，此时可能存在消费者offset没提交，会导致重平衡后重复消费。

## Kafka重复消费的解决办法

第一种方式其实可以尽量减少不必要的Rebalance（当消费速度过慢时有可能会触发rebalance, 这批消息被分配到另一个消费者，然后新的消费者还会消费过慢，再次rebalance），从而导致消息被多次重复消费。

1. 提高消费者的处理速度。例如：对消息处理中比较耗时的步骤可通过利用多线程处理等。在缩短单条消息消费的同时，根据实际场景可将max.poll.interval.ms值（消费者每次拉消息的最大间隔，如果超过这个间隔，服务端会认为消费者已离线）设置大一点，避免不必要的Rebalance（consumer组重新分配）。可根据实际消息速率适当调小max.poll.records（定义了poll()方法最多可以返回多少条消息，默认值为500）的值。
2. 引入消息去重机制。例如：生成消息时，在消息中加入唯一标识符如消息id等。在消费端，可以保存最近的max.poll.records条消息id到redis或mysql表中，这样在消费消息时先通过查询去重后，再进行消息的处理。
3. 保证消费者逻辑幂等。

## kafka 控制器是什么？有什么作用

在 Kafka 集群中会有一个或多个 broker，其中有一个 broker 会被选举为控制器，**它负责管理整个集群中所有分区和副本的状态**，kafka 集群中**只能有一个控制器**。

* 当某个分区的 leader 副本出现故障时，由控制器**负责为该分区选举新的 leader 副本**。
* 当检测到某个分区的ISR集合发生变化时，由控制器**负责通知所有 broker 更新其元数据信息**。
* 当为某个 topic 增加分区数量时，由控制器**负责分区的重新分配**。

## Kafka控制器的选举方式

kafka 中的控制器选举工作**依赖于 Zookeeper**，成功竞选成为控制器的 broker 会在Zookeeper中创建/controller临时节点。

每个 broker 启动的时候会去尝试读取/controller 节点的 brokerid的值。

* **如果**读取到的 **brokerid 的值不为-1**，表示已经有其他broker 节点成功竞选为控制器，所以当前 broker **就会放弃竞选**；

如果Zookeeper中**不存在**/controller 节点，**或者**这个节点的数据**异常**，那么**就会尝试去创建**/controller 节点，**创建成功的那个 broker 就会成为控制器**。

每个 broker 都会在内存中保存当前控制器的 brokerid 值，这个值可以标识为 activeControllerId。

Zookeeper 中还有一个与控制器有关的/controller\_epoch 节点，这个节点是**持久节点**，节点中存放的是一个整型的 controller\_epoch 值。controller\_epoch 值用于**记录控制器发生变更的次数**。

controller\_epoch 的**初始值为1**，即集群中的第一个控制器的纪元为1，当控制器发生变更时，**每选出一个新的控制器就将该字段值加1**。

每个和控制器交互的请求都会携带 controller\_epoch 这个字段，

* **如果请求的 controller\_epoch 值**小于**内存中的 controller\_epoch值**，**则**认为这个请求是向已经过期的控制器发送的请求，那么这个请求会**被认定为无效的请求**。
* 如果请求的 controller\_epoch 值**大于**内存中的 controller\_epoch值，那么说明**已经有新的控制器当选**了。

## kafka 为什么这么快

1. **顺序读写**磁盘分为顺序读写与随机读写，基于磁盘的随机读写确实很慢，但磁盘的顺序读写性能却很高，kafka 这里采用的就是顺序读写。
2. **Page Cache**为了优化读写性能，Kafka 利用了操作**系统本身的 Page Cache**，就是利用操作系统自身的内存而不是JVM空间内存。
3. **零拷贝**Kafka使用了零拷贝技术，也就是**直接将数据从内核空间的读缓冲区直接拷贝到内核空间的 socket 缓冲区**，然后再写入到 NIC 缓冲区，避免了在内核空间和用户空间之间穿梭。
4. **分区分段+索引**Kafka 的 message 是按 topic分 类存储的，topic 中的数据又是按照一个一个的 partition 即分区存储到不同 broker 节点。每个 partition 对应了操作系统上的一个文件夹，partition 实际上又是按照segment分段存储的。通过这种分区分段的设计，Kafka 的 message 消息实际上是分布式存储在一个一个小的 segment 中的，每次文件操作也是直接操作的 segment。为了进一步的查询优化，Kafka 又默认为分段后的数据文件建立了索引文件，就是文件系统上的.index文件。这种分区分段+索引的设计，不仅提升了数据读取的效率，同时也提高了数据操作的并行度。
5. **批量读写**Kafka **数据读写也是批量的而不是单条的**,这样可以避免在网络上频繁传输单个消息带来的延迟和带宽开销。假设网络带宽为10MB/S，一次性传输10MB的消息比传输1KB的消息10000万次显然要快得多。
6. **批量压缩**Kafka 把所有的消息都变成一个**批量的文件**，并且进行合理的**批量压缩**，减少网络 IO 损耗，通过 mmap 提高 I/O 速度，写入数据的时候由于单个Partion是末尾添加所以速度最优；读取数据的时候配合 sendfile 进行直接读取。

## 什么情况下 kafka 会丢失消息

Kafka 有**三次消息传递**的过程：生产者发消息给 Broker，Broker 同步消息和持久化消息，Broker 将消息传递给消费者。

这其中每一步都有可能丢失消息。

# 三高问题、分布式、设计模式和Docker

## 设计模式分类（创建型、结构型、行为型）

**创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。**

**工厂模式：**它提供了一种创建对象的最佳方式。在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。实现了创建者和调用者分离，工厂模式分为简单工厂、工厂方法、抽象工厂模式。

**抽象工厂模式：**抽象工厂简单地说是工厂的工厂，抽象工厂可以创建具体工厂，由具体工厂来产生具体产品。

**单例模式：**保证一个类只有一个实例，并且提供一个能够访问全局的访问点，在单例模式中，活动的单例只有一个实例，对单例类的所有实例化得到的都是相同的一个实例。这样就防止其它对象对自己的实例化，确保所有的对象都访问一个实例。

**建造者模式：**是将一个复杂的对象的构建与它的表示分离，使得同样的构建过程可以创建不同的方式进行创建。工厂类模式是提供的是创建单个类的产品，而建造者模式则是将各种产品集中起来进行管理，用来具有不同的属性的产品。

**原型模式：**原型设计模式简单来说就是克隆，表明了有一个样板实例，这个原型是可定制的。原型模式多用于创建复杂的或者构造耗时的实例，因为这种情况下，复制一个已经存在的实例可使程序运行更高效。

**结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。**

**代理模式：**通过代理控制对象的访问，可以在这个对象调用方法之前、调用方法之后去处理/添加新的功能。能够在原有代码乃至原业务流程都不修改的情况下，直接在业务流程中切入新代码，增加新功能。

**外观模式：**隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口，用这一个接口来隐藏实际的系统的复杂性。

**行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。**

**模板方法：**定义一个操作中的算法骨架，而将一些步骤放到子类中。模板方法使得子类可以不改变一个算法的结构即可重新定义该算法的某些特定步骤。

**策略模式：**定义了一系列的逻辑或相同意义的操作，并将每一个算法、逻辑、操作封装起来，而且使它们还可以相互替换。我觉得主要是为了 简化 if…else 所带来的复杂和难以维护。

**观察者模式：**观察者模式，是一种行为性模型，又叫发布-订阅模式，他定义对象之间一种一对多的依赖关系，使得当一个对象改变状态，则所有依赖于它的对象都会得到通知并自动更新。

## 设计模式六大原则

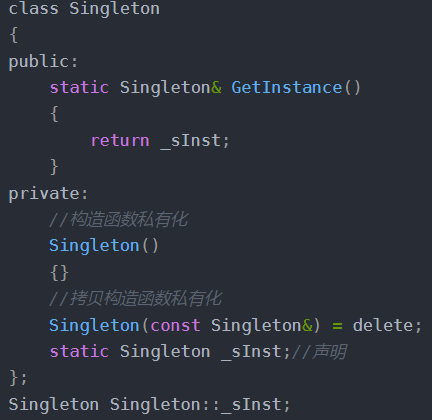
1. 单一职责：一个方法只负责一件事情， 一个类只负责一个职责，各个职责的程序改动，不影响其它程序。 这是几乎所有程序员都会遵循的一个原则。
2. 开放封闭原则：尽量通过扩展软件实体来解决需求变化，而不是通过修改已有的代码来完成变化，一个软件产品在生命周期内，是都会发生变化，既然变化是一个既定的事实，我们就应该在设计的时候尽量适应这些变化，以提高项目的稳定性和灵活性。
3. 里氏替换原则：子类可以扩展父类的功能，但不能改变父类原有的功能。子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法，子类中可以增加自己特有的方法。
4. 接口分离原则：使用多个隔离的接口，比使用单个接口要好，其实也是一个降低类之间的耦合度的意思，我们的目标就是要降低依赖，降低耦合。
5. 依赖反转原则：依赖倒置原则的核心思想是面向接口编程，我们要对接口编程，依赖于抽象的接口而不依赖于具体的实现。
6. 最少知道原则：大概意思就是一个类尽量减少自己对其他对象的依赖，原则是低耦合，高内聚，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。

## 单例模式（懒汉式和饿汉式）

C++单例模式和线程安全

（单例模式介绍）单例模式：一个类只能创建一个对象，即单例模式，该模式可以保证系统中该类只有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息，这种方式简化了在复杂环境下的配置管理。单例模式有两种实现方式，分别是饿汉模式和懒汉模式。

（饿汉模式）思想：不管将来是否使用，程序启动的时候就创建一个唯一的实例对象。C++实现如下，主要是通过私有化构造函数和析构函数实现单例：

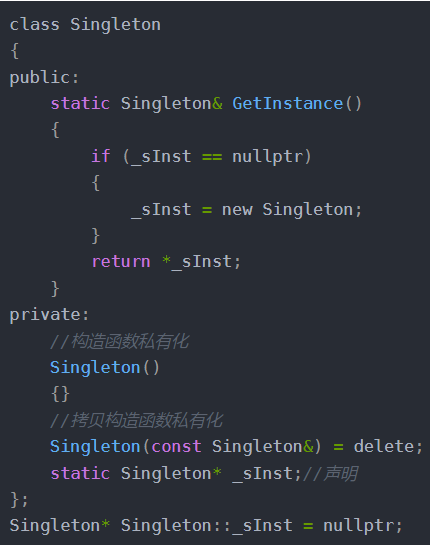


（饿汉模式的线程安全问题）饿汉模式下是没有线程安全问题的，因为饿汉模式通常在mian()函数调用之前实例化了对象，main()调用后多线程才会进入，所以不存在线程安全问题。

（饿汉模式优缺点）优点：实现简单，不存在线程安全的问题。缺点：在main函数之前初始化实例，如果程序中单例较多，程序启动慢，性能过低。其次如果两个单例类有依赖关系，无法保证创建初始化实例的顺序。

（懒汉模式）思想：使用的时候才创建并加载资源，进程启动无负担，并且多个实例的初始化顺序可以自由控制。

概念：如果单例对象构造十分耗时或者占用很多资源，比如加载插件啊，初始化网络连接啊，读取文件等等，而有可能该对象在程序运行时不会马上用到，那么如果在程序一开始就进行初始化（饿汉模式），就会导致程序启动时非常的缓慢。 所以这种情况使用懒汉模式（延迟加载）更好。



（懒汉模式的线程安全问题）懒汉模式在多线程并发情况下不是线程安全的，假设在多线程场景下，第一个线程进入时指针为空申请空间，但这个线程如果还没来得及退出，第二个线程进来判断时，指针仍为空，再次申请空间，导致无法实现单例模式下的线程安全。

（懒汉模式线程安全问题办法）锁机制、信号量机制等，其实就是线程同步问题。

## 后台三高问题

高并发：

1. 应用数据与静态资源分离，将静态资源放到静态资源服务器中，用户数据从主服务器中获取。
2. 池化技术：池化技术的核心是“预分配”和“循环使用”，常用的池化技术有：线程池、进程池、协程池等。
3. 集群和分布式：集群表示所有的服务器都有相同的功能，请求哪台主机都可以，主要起分流的作用。
4. 分布式：将不同的业务放到不同的服务器中，处理一个请求可能会用到多个服务器，加快服务器的处理时间。

高性能：

1. 客户端缓存：客户端缓存表示将静态信息保留到客户端中，当数据发生更新或者失效的时候再选择重新缓存。
2. Cache缓存：关于CPU和内存的访问速度上可以加入cache提高它们的访问速度。
3. Io优化：避免使用阻塞io、非阻塞io的方式，在IO上可以选用多路IO复用+线程池等模型去替换。

高可用：

1. 主备切换
2. 熔断（当即将超过系统的承载能力时，自动采取保护措施防止崩溃）
3. 限流保护（使用验证码或者答题的方式限制用户访问）
4. 服务降级（关闭成交记录等、只开启支付功能，参考双11）

## 服务高可用策略（限流、降级、容灾）

1. 负载均衡：将请求均衡地发送到各个服务结点，避免出现过载或者饥饿的现象。最常用的算法有：轮询法、随机法、加权随机法、最小连接法、一致性哈希等
2. 分流：不同流量，分而治之，避免相互不影响。
3. 限流：过载保护，防止缓存雪崩，如计数器算法、滑动窗口算法、漏桶和令牌桶算法等。
4. 降级：对非核心链路进行降级，优先保障核心链路。
5. 容灾：应付各种不可抗拒的因素和人为错误，常见的做法有存储冗余（HDFS），服务多地部署等。
6. 监控：实时监测系统关键指标，及时发现问题，做到服务可观测。

漏桶和令牌桶的本质差别？

漏桶的本质是总量控制，令牌桶的本质是速率控制。令牌桶可以用来保护自己，主要用来对调用者频率进行限流，为的是让自己不被打垮。所以如果自己本身有处理能力的时候，如果流量突发（实际消费能力强于配置的流量限制），那么实际处理速率可以超过配置的限制。而漏桶算法，这是用来保护他人，也就是保护他所调用的系统。主要场景是，当调用的第三方系统本身没有保护机制，或者有流量限制的时候，我们的调用速度不能超过他的限制，由于我们不能更改第三方系统，所以只有在主调方控制。这个时候，即使流量突发，也必须舍弃。因为消费能力是第三方决定的。

## 常见的系统性能调优策略

1. 代码：代码上要去检查是否使用了合适的数据结构，使用火焰图等工具查看其性能瓶颈，保证代码的性能后再去处理其他问题。
2. 数据库调优：

SQL调优：可以使用慢查询日志来定位到出问题的SQL，通过explain、profile等工具进行调优。

架构层面调优：读写分离、主从库负载均衡、水平和垂直分库分表等。

连接池调优：根据具体的连接池监控数据以及当前的业务量做出对应的判断进行调优。

1. 缓存：

如果数据量小，并且不会频繁地增长清空，可以考虑本地缓存，否则可以考虑redis等缓存服务。

考虑缓存穿透、缓存击穿、缓存雪崩等问题处理。

1. 异步：

针对客户端的某些请求，在服务端可能要针对这些请求做一些附属的事情，而并不关心得到的结果，可以选择异步的方式去进行处理这些事情。

常见做法：额外**开辟新的线程或者采用线程池**的策略让处理请求的线程先返回结果，其他线程异步地进行处理。或者可以采用**消息队列**的思想，一些额外任务可能不需要这个系统来处理，可以将其封装成一个消息丢到消息队列里面。

## 负载均衡的作用和实现方式

负载均衡（Load Balancer）是指把用户访问的流量，通过「负载均衡器」，根据某种转发的策略，均匀的分发到后端多台服务器上，后端的服务器可以独立的响应和处理请求，从而实现分散负载的效果。负载均衡技术提高了系统的服务能力，增强了应用的可用性。

负载均衡的实现主要有三种方案：基于DNS的负载均衡（客户端）、基于硬件的负载均衡（服务器端）、基于软件的负载均衡（服务器端）。

（基于DNS的负载均衡）通过对DNS服务器做出对应配置，可以实现用户的“就近原则”分流，减轻了单个集群的负载压力，例如南方的用户通过DNS服务器拿到的是广州服务器集群的IP，北方拿到的是北京的服务器集群的IP。基于DNS的负载均衡实现简单，但是会有配置修改之后的生效不及时问题，因为DNS一般是多级缓存的结构，会导致IP更改不及时。

（基于硬件的负载均衡）可以通过网络设备来分散大量的请求，例如F5 Network Big-IP这样的设备，这样的设备就类似于网络交换机，性能强大高达几百万/秒，缺点就是设备比较昂贵。

（基于软件的负载均衡）软件负载均衡是指使用软件的方式来分发和均衡流量。软件负载均衡，分为7层协议和4层协议。网络协议有七层，基于第四层传输层来做流量分发的方案称为4层负载均衡，例如 LVS，而基于第七层应用层来做流量分发的称为7层负载均衡，例如 Nginx。这两种在性能和灵活性上是有些区别的。基于4层的负载均衡性能要高一些，一般能达到几十万/秒的处理量，而基于7层的负载均衡处理量一般只在几万/秒 。

DNS负载均衡可以实现在地域上的流量均衡，硬件负载均衡主要用于大型服务器集群中的负载需求，而软件负载均衡大多是基于机器层面的流量均衡。在实际场景中，这三种是通常组合在一起使用。

## 负载均衡算法

1. 随机法，拿到服务器的地址列表，产生一个随机数选择服务器。这种做法缺点，因为按照概率产生的随机数，导致部分机器在一段时间之内无法被随机访问到。此外，基于随机法，可以对每个机器设置权重的做法
2. 轮询法：挨个轮询服务器处理，也可以设置权重。加权轮询算法适合于服务器性能不一样的集群，权重的存在可以使得请求分配更合理。最常见的这种轮询方法就是DNS解析来，DNS解析的过程就是把域名发到本地域名服务器，找不到在找根域名服务器，然后继续查找上一级，如果找到来最终返回一个IP地址，这样实现轮询版负载均衡算法。
3. 最小连接法

当新的请求出现是，遍历服务器节点列表并选取其中活动连接数最小的一台机器相应当前请求。所以需要监控每台服务器处理请求连接的数量，实现比较复杂。

1. 一致性哈希算法（分布式缓存一致）

一致性哈希是将整个哈希值空间组织成一个虚拟的圆环。如假设哈希函数H的值空间为0-2^32-1（哈希值是32位无符号整形），这个时候整个空间按顺时针方向组织，0和2^32-1在零点中方向重合。接下来，把服务器按照IP或主机名作为关键字进行哈希，确定其在哈希环的位置。然后，对请求来说，就可以使用哈希函数H计算值为key的数据在哈希环的具体位置h，根据h确定在环中的具体位置，从此位置沿顺时针滚动，遇到的第一台服务器就是其应该定位到的服务器。

存在的问题：当集群中节点较少时，可能出现节点在哈希空间中分布不平衡的问题。hash环倾斜。比如这种情况，会造成A节点出现故障，存储在A上的数据要全部转移到B上，大量的数据导可能会导致节点B的崩溃，之后A和B上所有的数据向节点C迁移，导致节点C也崩溃，由此导致整个集群宕机。这种情况被称为雪崩效应。

解决办法——虚拟节点，将现有的物理节点通过虚拟的方法复制多个出来。

## 一致性哈希算法（分布式缓存一致）

为什么要引入一致性哈希？

在分布式集群中，现假设有一台Redis服务器宕机了，那么为了填补空缺，要将宕机的服务器从编号列表中移除，后面的服务器按顺序前移一位并将其编号值减一，此时每个key就要按h = Hash(key) % 2重新计算。

同样，如果新增一台服务器，规则也同样需要重新计算，h = Hash(key) % 4。因此，系统中如果有服务器更变，会直接影响到Hash值，大量的key会重定向到其他服务器中，造成缓存命中率降低，而这种情况在分布式系统中是十分糟糕的。

基于上面的缺点提出了一种新的算法：一致性哈希。一致性哈希可以实现节点删除和添加只会影响一小部分数据的映射关系，由于这个特性哈希算法也常常用于各种均衡器中实现系统流量的平滑迁移。

算法部分：

简单的说，一致性哈希是将整个哈希值空间组织成一个虚拟的圆环。如假设哈希函数H的值空间为0-2^32-1（哈希值是32位无符号整形），这个时候整个空间按顺时针方向组织，0和2^32-1在零点中方向重合。接下来，把服务器按照IP或主机名作为关键字进行哈希，确定其在哈希环的位置。然后，对请求来说，就可以使用哈希函数H计算值为key的数据在哈希环的具体位置h，根据h确定在环中的具体位置，从此位置沿顺时针滚动，遇到的第一台服务器就是其应该定位到的服务器。

一致性哈希的扩容，我们可以针对集群中某个结点的压力情况进行选择性扩容，步骤为：

1、针对需要扩容的数据库节点增加从节点，开启主从同步进行数据同步。

2、完成主从同步后，对原主库进行禁写。

3、同步完全完成后，断开主从关系，理论上此时从库和主库有着完全一样的数据集。

4、修改一致性Hash范围的配置，并使应用服务重新读取并生效。

5、确定所有的应用均接受到新的一致性Hash范围配置后，放开原主库的禁写操作，此时应用完全恢复服务。

6、启动离线的定时任务，清除冗余的数据，由此完成了针对结点的扩容任务。

## 分布式相关内容（cap、base理论）

**cap理论：**

* Consistency：一致性，对某个客户端来说，读操作能够返回最新对写操作结果
* Acailability：可用性，非故障结点在合理的时间内返回合理的响应
* Partition tolerance：分区容错性，当出现网络分区后，系统能够继续提供服务（必须要有的）

**base理论：**

* BA：基本可用，分布式系统在出现故障的时候，允许损失部分可用性，保证核心可用
* S：软状态，允许系统存在中间状态，而该中间状态不会影响系统整体可用性
* E：最终一致性，系统中的所有数据副本经过一段时间后能够达到一致的状态

**BASE理论本质上是对AP方案的一个补充。**

基础理论：分布式主备一致性，分布式缓存一致性

场景分类：文件系统（hdfs）、数据库（Hbase、Mysql）、缓存（Redis）、消息处理（Kafka、RocketMQ）、监控（Zookeeper）

高可用策略如上。

## Cap理论的应用场景

**CP：强一致性：**比如在mysql主从我们可以选择主数据库和从数据库都完成之后返回正确，强一致性的体现；zookeeper选择优先保证一致性。zookeeper保证访问请求都能保持一致的结果，同时具有容错性，但是不保证每次访问请求都是可用的，当zookeeper的master节点如果因为网络故障导致与其它节点失去联系时，剩余的节点会进行选举产生新的master节点，但是在这过程需要一定的时间，并且在选举这段时间内，整个zookeeper集群是不可用的。

**AP：可用性优先：**比如在mysq中主从环境中主数据库完成之后，从数据库部分完成之后就返回正确，提高了下响应速度，可用性变大，但一致性会受影响；Eureka选择优先保证可用性。Eureka的节点之间是平等的。如果某个节点服务器宕机了，Eureka不会像zookeeper那样进行停止服务进行选举，而是让请求自动切换到另外的可用Eureka节点上，等到宕机的节点恢复后，再将其纳入集群中，只是不保证查询到的信息可能不是最新的。同时，如果15分钟内超过85%的节点都没有心跳，那么Eureka就认为客户端和注册中心出现了网络故障，就执行这几种策略：

1. Eureka不在从注册列表中移除因为长时间没收到心跳而应该过期的服务
2. Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其他节点上 (即保证当前节点依然可用)
3. 当网络稳定时，当前实例新的注册信息会被同步到其他节点中

## 分布式事务方案（分布式一致性实现）

分布式事务就是要要在分布式系统中实现事务，它其实是由多个本地事务组合而成。

1. **二阶段提交**

是一种强一致性设计，2PC 引入一个事务协调者的角色来协调管理各参与者（也可称之为各本地资源）的提交和回滚，二阶段分别指的是准备（投票）和提交两个阶段。

**准备阶段**协调者会给各参与者发送准备命令，同步等待所有资源的响应之后就进入第二阶段即提交阶段（注意提交阶段不一定是提交事务，也可能是回滚事务）。

假如在第一阶段所有参与者都返回准备成功，那么协调者则向所有参与者发送**提交事务命令**，然后等待所有事务都提交成功之后，返回事务执行成功。

1. **三段式提交**

三段式表示：准备阶段、预提交阶段和提交阶段，首先会在准备阶段去访问参与者自身情况是否能够执行事务，如果全部正常工作则进入预提交阶段，然后进入提交阶段进行提交。

三段式提交针对二段式进行了一定的改进：引入了参与者超时机制，并且增加了预提交阶段使得故障恢复之后协调者的决策复杂度降低，但整体的交互过程更长了，性能有所下降，并且还是会存在数据不一致问题。

1. **TCC**

TCC中表示Try 指的是预留，即资源的预留和锁定，注意是预留；Confirm 指的是确认操作，这一步其实就是真正的执行了；Cancel 指的是撤销操作，可以理解为把预留阶段的动作撤销了。

两段式和三段式都是数据库层面的，而 TCC 是业务层面的分布式事务，所以TCC**可以跨数据库、跨不同的业务系统来实现事务。**

1. **本地消息表**

本地消息表其实就是利用了各系统本地的事务来实现分布式事务。

本地消息表就是会有一张存放本地消息的表，一般都是放在数据库中，然后在执行业务的时候将执行的业务和消息放入消息表中的操作放在同一个事务中，这样就能保证消息放入本地表中业务肯定是执行成功的。

然后再去调用下一个操作，如果下一个操作调用成功，消息表的消息状态可以直接改成已成功。如果调用失败则会有后台任务定时去读取本地消息表，筛选出还未成功的消息再调用对应的服务。

1. **消息事务（RocketMQ能够很好支持）**

先给 Broker 发送事务消息即半消息，半消息不是说一半消息，而是这个消息对消费者来说不可见，然后发送成功后发送方再执行本地事务。

再根据本地事务的结果向 Broker 发送 Commit 或者 RollBack 命令。

1. **最大努力通知**

就本地消息表来说会有后台任务定时去查看未完成的消息，然后去调用对应的服务，当一个消息多次调用都失败的时候可以记录下然后引入人工，或者直接舍弃。这也算是尽最大努力了。

## 数据库一致性和cap一致性有什么区别

**数据库 ACID 的一致性**也称之为**内部一致性**，可以说是是事务开始前和结束后，数据库的完整性约束没有被破坏。比如 A 向 B 转账，不可能 A 扣了钱，B 却没收到。

**分布式 CAP 的一致性**也称之为**外部一致性**，可以说是为在分布式系统中，写操作后再读，就必须返回写入的值。比如分布式数据库A、B、C，A 中写入数据x = 1，写完马上读 B 和 C，就一定要读出x = 1，读出来我们就称之为符合一致性。

**两者区别**，内部一致性注重于事务前后数据的完整性，而外部一致性则注重于读写数据的一致性。

## 为什么要用分布式锁

使用分布式锁的目的就是保证同一时间只有一个**客户端**可以对共享资源进行操作。

1. 允许多个客户端操作共享资源

这种情况下，对共享资源的操作一定是幂等性操作，无论你操作多少次都不会出现不同结果。在这里使用锁，无外乎就是为了避免重复操作共享资源从而提高效率。

1. 只允许一个客户端操作共享资源

这种情况下，对共享资源的操作一般是非幂等性操作。在这种情况下，如果出现多个客户端操作共享资源，就可能意味着数据不一致，数据丢失。

## 分布式锁的实现方式

1. MySQL实现锁：

基于数据库的实现方式的核心思想是：在数据库中创建一个表，表中包含方法名（或者请求名）等字段，并在方法名字段上创建唯一索引，想要执行某个方法，就使用这个方法名向表中插入数据，成功插入则获取锁，执行完成后删除对应的行数据释放锁。

使用基于数据库的这种实现方式很简单，但是对于分布式锁应该具备的条件来说，它有一些问题需要解决及优化，所以可以采用Redis或者zookeeper进行实现。

1. Redis实现分布式锁：

核心思想是：1、获取锁的时候，使用setnx加锁，并使用expire命令为锁添加一个超时时间，超过该时间则自动释放锁，锁的value值为一个随机生成的UUID，通过此在释放锁的时候进行判断。2、获取锁的时候还设置一个获取的超时时间，若超过这个时间则放弃获取锁。 3、释放锁的时候，若是该锁，则执行delete进行锁释放。

////\*\*\*\*

Redis 锁主要利用 Redis 的 **setnx 命令（原子性操作）**。

* 加锁命令：SETNX key value，当键不存在时，对键进行设置操作并返回成功，否则返回失败。
* 解锁命令：DEL key，通过删除键值对释放锁，以便其他线程可以通过 SETNX 命令来获取锁。
* 锁超时：EXPIRE key timeout, 设置 key 的超时时间，以保证即使锁没有被显式释放，锁也可以在一定时间后自动释放，避免资源被永远锁住。

\*\*\*\*/////

1. 基于Zookeeper实现分布式锁：

利用Zookeeper创建瞬时节点有序的特性，在多线程并发创建瞬时节点，会得到有序的节点序列，我们规定序号最小的线程获得锁。其他线程利用观察器监听自己序号前一个序号的存在状态，前一个线程执行完成，删除自己序号的节点，下一个序号的线程得到通知，执行自己的任务。

## 数据库、redis、zookeeper实现分布式锁的优缺点

一个好的分布式锁应该具备几个条件：1、在分布式系统环境下，一个方法在同一时间只能被一个机器的一个线程执行；2、高可用以及高性能的获取锁与释放锁；3、具备可重入特性（保证锁的有效时间内该锁都是属于请求者，不会被中断）；4、具备锁失效机制（过期机制），防止死锁；5、具备非阻塞锁特性，即没有获取到锁将直接返回获取锁失败。

**MySQL的优缺点：**

* 因为是基于数据库实现的，数据库的**可用性**和**性能**将直接影响分布式锁的可用性及性能，所以，数据库需要多机部署、数据同步、主备切换等操作；
* **不具备可重入**的特性，因为同一个线程在释放锁之前，行数据一直存在，无法再次成功插入数据，所以，需要在表中新增一列，用于记录当前获取到锁的机器和线程信息，在再次获取锁的时候，先查询表中机器和线程信息是否和当前机器和线程相同，若相同则直接获取锁。
* **没有锁失效机制**，因为有可能出现成功插入数据后，服务器宕机了，对应的数据没有被删除，当服务恢复后一直获取不到锁，所以，需要在表中新增一列，用于记录失效时间，并且需要有定时任务清除这些失效的数据。
* 不具备**阻塞锁**特性，获取不到锁直接返回失败，所以需要优化获取逻辑，循环多次去获取。

**Redis优点：**

Redis有很高的性能，且Redis命令对分布式锁的支持较好，实现起来比较方便。

**Zookeeper优缺点：**

使用zookeeper实现分布式锁的优点是，具备高可用、可重入、阻塞锁特性，可解决失效死锁问题。因为需要频繁的创建和删除节点，性能上不如Redis方式。

**Redis和Zookeeper实现分布式锁对比：**

* 从获取锁的角度来说，Redis 分布式锁，其实需要自己不断去尝试获取锁，比较消耗性能。 Zookeeper分布式锁，获取不到锁，注册个监听器即可，不需要不断主动尝试获取锁，性能开销较小。
* 比较redis和zookeeper的性能来说，zookeeper需要频繁的创建和删除节点，性能上不如Redis方式。
* 另外一点就是，如果是 Redis 获取锁的那个客户端出现 bug 挂了，那么只能等待超时时间之后才能释放锁；而 Zookeeper的话，因为创建的是临时结点，只要客户端挂了，结点就没了，此时就自动释放锁。

## Redis实现分布式锁的key和value如何设置（方法名+唯一ID）

分布式锁的key设置为对应的方法名，value应该设置为一个指定的唯一ID，释放前获取这个值，判断是否是自己的锁避免误删。

这里的value可以基于redis提供的自增id，incr来实现唯一id。

* incr的自增是原子性的，是线程安全的，同一时间点只能有一个线程修改。
* 返回值是自增之后的值。
* 操作简单，直接执行指令获取返回值就行了。

（对应的业务场景）假设业务逻辑的执行时间是7秒：（1）业务逻辑1没执行完，3秒后锁被自动释放。（2）业务逻辑2获取到锁，执行业务逻辑，3秒后锁被自动释放。（3）业务逻辑3获取到锁，执行业务逻辑。（4）业务逻辑1执行完成，开始调用del释放锁，这时释放的是业务逻辑3的锁，导致业务逻辑3的业务只执行1s就被别人释放。出现没有上锁的情况。

## 分布式一致性协议Raft协议

Raft算法是一个用户管理日志一致性的协议，它将分布式一致性问题分成多个子问题：Leader选举、日志复制、安全性、日志压缩等。在Redis主从架构中就是使用了Raft协议：

1. **raft角色介绍：**

Leader角色：接受客户端请求，并向Follower同步请求日志，当日志同步到大多数结点后告诉Follower进行提交日志。

Follower角色：接受并持久化Leader 同步的日志，在Leader告知可提交后进行日志提交。

Candidate：Leader选举过程中的临时角色。

1. **任期：**

在分布式系统中，各个结点的时间同步是一个很大的难题，为了识别过期时间，raft协议引入了term（任期）的概念。

Raft将时间划分为任意不同长度的任期，每个任期的开始都是一次选举，一个或者多个候选人会试图成为Leader，但是某些情况下选票被瓜分，没有选出领导人，那么会开始下一个任期，并且立即开始下一次选举。Raft保证在一次任期内最多只有一个Leader。

1. **Leader哨兵选举流程：**

* 每个做主观下线的哨兵向其他哨兵结点发送命令，该哨兵会先看看自己有没有投过票，如果自己已经投过票给其他哨兵了，在2倍故障转移的超时时间（默认故障转移时间为3分钟）自己就不会成为Leader，否则要求将自己设置为领导者。
* 如果该哨兵发现自己的票数已经超过一半并且超过了quorum（配置文件中的一个阈值）那么它成为Leader。
* 如果在一个任期中没能选出Leader，那么等2倍故障转移超时时间后，Candidate增加epoch重新投票。

## Docker常用命令

Pull、run、exec、delete、stop

## Docker容器的在宿主机中的pid

使用linux 系统命令 ps 进行查找容器id；再通过容器id在系统中查找该进程（ps -aux）。

## Docker介绍

基于linux内核的cgroup，namespace等技术，对进程进行封装隔离，属于操作系统层面的虚拟化技术。

docker的特点

1. 轻量：在一台机器上运行的多个docker容器可以共享这台机器的操作系统内核；它们能够快速启动，只需占用很少的计算和内存资源。镜像是根据文件系统进行构建的，能够共享一些公共文件。这样能够尽量降低磁盘用量，并且能够更快的下载镜像。
2. 标准：docker容器基于开放式标准，能够在linux、windows以及云上等设备运行。
3. 安全：docker容器的隔离不仅限于彼此隔离，还独立于底层的基础措施。因此假如应用出现问题，也只是单个容器的问题。

简介：快速扩展、迁移方便。

## Docker和虚拟机

特点：系统资源利用更高效、更快启动、更容易维护与扩展。

虚拟机的核心是模拟硬件，让虚拟机的操作系统认为自己跑在一个真实的物理机器上，用软件模拟出来CPU，内存，硬盘，网卡，让虚拟机里面的操作系统觉得自己是在真实的硬件上，所以在操作系统上的进程同样也认为自己运行在一台真实的物理机上。

容器和虚拟机最大的差别就是容器并没有去模拟硬件，而是在操作系统上面做了手脚，让进程以为自己运行在了一个全新的操作系统上，这是他们之间最大的差别，因此容器一般都很小，启动起来就很快。

对于虚拟机来说，可以安装不同的操作系统，比如可以在win上装linux。

对于容器而言，因为是在操作系统上进行操作，所以容器内的操作系统和宿主机是一样的，本质上容器就是一个进程，和宿主机上面的其他进程没有什么区别。（uname -a查看内核状态）

## Docker如何实现进程隔离

每个容器的实质是进程，但和直接在宿主机执行的进程不同，容器进程运行于属于自己的独立的命名空间例如pid、net、ipc等命名空间。因此容器有自己的文件系统，自己的网络配置，自己的进程空间等，容器内的进程是运行在一个隔离的环境里面的，使用起来就好像是在一个独立于宿主的系统下操作一样。

## Docker搭建过程

**（1）文件系统的搭建：**

docker采用了联合文件系统，它能够将不同文件夹中的层联合（Union）到了同一个文件夹中，针对镜像分层的文件，最终会被docker通过联合的方式进行组装。

首先是bootloader，这部分启动部分是汇编代码，CPU从一个固定位置读取第一行汇编代码开始运行， bootloader会初始化CPU，内存，网卡这些信息，这部分主要作用是把操作系统的kernel代码从硬盘加载到内存中，然后跳转到kernel到main函数入口开始执行kernel代码，其实就是linux的内核代码，启动完成后会重新初始化CPU，内存，网卡等设备，然后开始运行内核代码，最后启动init进程开始执行kernel，然后kernel会挂载文件系统。

**（2）网络系统**

搭建好文件系统后还需要去建立网络环境，容器一般可设置三种网络支持分别是无网络模式、和宿主机一套网络、网桥模式。网桥模式是docker默认的模式，一个网桥有两个端口，两个端口是不同的网络去对接两个不同的网卡，在容器内的虚拟网卡连接网桥的一端，在宿主机连接另一段，这样容器内就能和宿主机进行联系了。

**（3）隔离进程**

通过不同的隔离模式实现不同的隔离，隔离模式有UTS隔离、用户隔离、IPC隔离、PID隔离、文件系统隔离、网络隔离。

UTS隔离主要是用来隔离hostname和域名

用户隔离表示容器里和宿主机用户可以做映射，里面虽然看到的是root用户，但实际上并不是，不能随便修改系统。

IPC隔离是进程通讯的隔离，表示容器里面和外面的进程不能进行进程间的通讯了，保证了较强的隔离型。

**（4）资源管理**

为了防止某个进程出错后抢占了所有的资源，可以通过cgroups对容器的资源进行分配。

# RPC介绍

## RPC介绍

[Thrift](https://thrift.apache.org/" \t "/Users/learna/Documents\x/_blank)是一个跨语言的服务部署框架，thrift通过IDL接口定义语言来定义[RPC](http://www.ibm.com/developerworks/cn/aix/library/au-rpc_programming/" \t "/Users/learna/Documents\x/_blank)的接口和数据类型，然后通过thrift编译器生成不同语言的代码，并由生成的代码负责RPC协议层和传输层的实现。

优点：

1. 一站式解决序列化和RPC支持
2. 数据类型丰富，基本数据类型包括bool、byte、i16（short）、i32（int）、i64（long）、double、string，特殊的数据类型binary、struct类型、容器list、set、map
3. RPC和序列化性能可以
4. 很多项目支持thrift，包括hive、spark

缺点：

1. 文档少（2）兼容性差，版本可能出现不兼容

## RPC协议具体介绍（Thrift）

[Thrift](https://thrift.apache.org/" \t "/Users/learna/Documents\x/_blank)是一个跨语言的服务部署框架，thrift通过IDL接口定义语言来定义[RPC](http://www.ibm.com/developerworks/cn/aix/library/au-rpc_programming/" \t "/Users/learna/Documents\x/_blank)的接口和数据类型，然后通过thrift编译器生成不同语言的代码，并由生成的代码负责RPC协议层和传输层的实现。

优点：

1. 一站式解决序列化和RPC支持
2. 数据类型丰富，基本数据类型包括bool、byte、i16（short）、i32（int）、i64（long）、double、string，特殊的数据类型binary、struct类型、容器list、set、map
3. RPC和序列化性能可以
4. 很多项目支持thrift，包括hive、spark

缺点：

1. 文档少（2）兼容性差，版本可能出现不兼容

## HTTP和RPC的区别

两者都常用于实现服务，在这个层面最本质的区别是RPC服务主要工作在TCP协议之上（也可以在HTTP协议），而HTTP服务工作在HTTP协议之上。由于HTTP协议基于TCP协议，所以RPC服务天然比HTTP更轻量，效率更胜一筹。

RPC接口即相当于调用本地接口一样调用远程服务的接口，HTTP接口是基于http协议的post接口和get接口（等等，2.0版本协议子支持更多）。

RPC主要用于公司内部服务调用，性能消耗低，传输效率高，服务治理方便。HTTP主要用于对外的异构环境，浏览器调用，APP接口调用，第三方接口调用等等。

## RPC请求过程

1. 服务消费端（client）以本地调用的方式调用远程服务；
2. 客户端桩 Stub（client stub） 接收到调用后负责将方法、参数等组装成能够进行网络传输的消息体（序列化）；
3. 客户端桩Stub（client stub） 找到远程服务的地址，并将消息发送到服务提供端；
4. 服务端桩 Stub收到消息将消息反序列化为具体的对象；
5. 服务端桩 Stub根据RPC请求(RpcRequest)中的类、方法、方法参数等信息调用本地的方法；
6. 服务端桩 Stub得到方法执行结果并将组装成能够进行网络传输的消息体：RPC响应（RpcResponse）（序列化）发送至消费方；
7. 客户端桩 Stub（client stub）接收到消息并将消息反序列化为具体的对象:RpcResponse ，这样也就得到了最终结果。

## 使用RPC需要注意的地方（同步调用和异步调用）

1. **同步调用，**在微服务架构模式下，跨服务的同步调用会存在雪崩的危险。场景就是，通常一个业务的调用会跨 N（N 一般大于 2） 个服务进程，整个调用链路上的同步调用等待的限制会由最慢（或脆弱）的服务决定，例如A-B-C 像这样一个链路，A 同步调用 B 并等待B返回结果，B 同步调用 C 并等待服务C返回结果，以此类推，就像一组齿轮链，这样一级一级地传动，这很容易产生**雪崩效应**。若 C 服务挂住了，会导致前面的服务全部都因为等待超时而占用大量不必要的线程资源。

**RPC 的同步调用**确保请求送达对方并收到对方响应，若没有收到响应，框架则抛出超时异常。这种情况下调用方是无法确定调用是成功还是失败的，需要根据业务场景（是否可重入，幂等）选择重试和补偿策略。因此，微服务架构下，内部主服务链之间的 RPC 调用需要异步化，服务之间的调用请求和等待结果相互之间解耦。

1. **异步调用，**RPC 框架不阻塞调用方线程，调用方不需要立刻拿到返回结果，甚至调用方根本就不关心返回结果，具体就是使用Future对象接受异步请求返回的结果。

## 异步接口调用的有序性

使用一致性hash负载均衡， 保证同一个rpc请求的id调用到同一个服务提供者上，然后服务提供者针对同一个rpc请求id放入一个内存队列，串性化执行调用。

## 调用RPC接口的幂等性保证

解决方案：第一是基于数据库的唯一索引，调用一次rpc接口就插入一条rpc记录到数据库中，这样即使网络出现抖动情况下，先查询数据库的请求rpc id。第二种方案就是redis+token机制，第一步：客户端发起获取token请求，服务端生成token，传输给客户端，并把token存入redis中设置过期时间。第二步：用户携带者token请求，服务端先查询token，如果存在，则正常进行接口调用并从redis删除此token。如果不存在，说明参数不合法或者重复请求，返回提示。

# 项目相关

## 为什么想到用epoll多路复用和线程池？（线程开销大的地方）

**为何用Epoll？**

如果使用阻塞IO模型的话，每次应用进程从发起 IO 系统调用，至内核返回成功标识，这整个期间是处于阻塞状态的，造成了资源的浪费，如果使用非阻塞IO模型的话进程会反复调用recvfrom等待返回成功提示（轮询方式），而且这两种IO模型能够同时监听的socket数量有限，为了能够同时监听更多socket，我们会采用一个多路IO复用机制，也就是select、poll、epoll这些，而我们的epoll在监听数量、响应速度上优势更大，所以项目采用了epoll多路IO复用。

**为什么用线程池？**

对操作系统来说,创建一个线程的代价是十分昂贵的, 需要给它分配内存、列入调度，同时在线程切换的时候还要执行内存换页，CPU 的缓存被清空,切换回来的时候还要重新从内存中读取信息,破坏了数据的局部性；所以针对线程创建开销大以及切换繁琐的问题，采用了资源复用以及限制线程数的思想：

资源复用：尽可能的复用资源，这里的资源当然是线程资源，提前预热全部或部分线程，由一个组件负责线程的分配和调度执行。

限制线程数：限制线程数是为了保护系统，避免线程创建过多导致内存不足。

## 多路IO复用（select、poll、epoll）

1. Select

Select采用了等待队列的数据结构，当新建完多个socket后保存其fd（文件描述符）并放入等待队列，调用select首先会将fd\_set（文件描述符列表）从用户态拷贝到内核态，当有事件就绪或者超时后以轮回的方式查看哪个fd是可操作的，相当于是进行了两次遍历，因此这样会比较浪费时间，默认可监视的数量是1024。**缺点1能够同时监听的用户太少，缺点2以轮回的方式调用太浪费时间，缺点3每次调用select需要将fd集合从用户态拷贝到内核态，开销太大。**

select其实是在**内核**中开辟一块数组空间用来存储用户空间传递的文件描述符进行监控,监控这件事也是内核做的，我们只需要传递文件描述符，**在内核空间进行检测，当有了就绪事件就拷贝到用户空间进行处理。**

1. Poll

Poll和select传参不一样，poll传过去的不是数组结构而是链表结构，打破了select1024的限制。

1. Epoll（1G的内存上能监听约10万个端口）

Epoll在内核里面有一个相应的数据结构去存储数据，这个数据结构就是**eventpoll**，它主要存储两方面的信息：需要监听的fd（文件描述符）集合**（红黑树实现，因为要利于添加和删除，还要便于搜索以免重复添加）**和就绪列表**（双向链表）**用来存放就绪的socket集合。Eventpoll主要采用了回调的方式去代替轮询检查，当在eventpoll中注册一个socket时，会在等待队列保存eventpoll的引用，如果socket读缓存区写入数据后会采用回调的方式将其放入eventpoll中的就绪队列。

所以当调用epoll\_wait检测是否有事件发生时只需要检测eventpoll中的就绪队列是否为空即可。

## Epoll水平触发和边沿触发

**水平触发（非阻塞IO浪费资源）**：只要缓冲区有数据，epoll\_wait就会一直被触发，直到缓冲区为空；

**边沿触发（回调形式）**：只有所监听的事件状态改变或者有事件发生时，epoll\_wait才会被触发；

epoll边沿触发时，假设一个客户端发送100字节的数据，而服务器设定read每次读取20字节，那么一次触发只能读取20个字节，

然后内核调用epoll\_wait直到下一次事件发生，才会继续从剩下的80字节读取20个字节，

由此可见，这种模式其工作效率非常低且无法保证数据的完整性，因此边沿触发不会单独使用。边沿触发通常与非阻塞IO一起使用，其工作模式为：epoll\_wait触发一次，在while（1）循环内非阻塞IO读取数据，直到缓冲区数据为空（保证了数据的完整性），内核才会继续调用epoll\_wait等待事件发生。

**叙述阻塞recv下，内核接收数据全过程。**

**如下图所示，进程在recv阻塞期间，计算机收到了对端传送的数据（步骤①）。数据经由网卡传送到内存（步骤②），然后网卡通过中断信号通知cpu有数据到达，cpu执行中断程序（步骤③）。此处的中断程序主要有两项功能，先将网络数据写入到对应socket的接收缓冲区里面（步骤④），再唤醒进程A（步骤⑤），重新将进程A放入工作队列中。**

## 如何实现一个线程安全的队列

在入队的时候利用互斥锁unique\_lock，我们要做的第一件事是锁定互斥锁以确保没有其他人正在访问该资源。然后，我们将元素推入队列。当unique\_lock超出范围时，它会自动释放。

## 线程池/多线程的线程数量怎么处理

一个进程的线程最大线程数目受到操作系统虚拟内存大小的限制。

考虑到多进程并发的需求（CPU密集型任务、IO密集型任务、混合型任务），线程池的大小设定通常满足：**线程等待时间所占比例越高，需要越多线程。线程CPU时间所占比例越高，需要越少线程。**

得到经验公式：最佳线程数目 = （线程等待时间与线程CPU时间之比 + 1）\* CPU数目

CPU密集型任务应配置尽可能小的线程，通常配置CPU核心数 + 1的线程数目。+1的意义是，计算密集型的线程恰好在某时因为发生一个页错误或者因其他原因而暂停，刚好有一个“额外”的线程，可以确保在这种情况下CPU周期不会中断工作。

IO密集型任务应配置尽可能多的线程，因为IO操作不占用CPU，不要让CPU闲下来，应加大线程数量，如配置两倍CPU个数+1。

## LevelDB原理

leveldb 是一个持久化的 key/value 存储，key 和 value 都是任意的字节数组(byte arrays)，并且在存储时，key 值根据用户指定的 comparator 函数进行排序。同时，由于磁盘的随机写效率远远不及顺序写，leveldb将数据落盘的随机写都转换为顺序写。

为了保证数据完整性，levelDB每次的写入请求(包括修改和删除), 都会先写入bin log到磁盘，然后再进行数据修改。这个过程也就是WAL技术，即Write-Ahead Logging，关键点就是先写日志，再写磁盘。

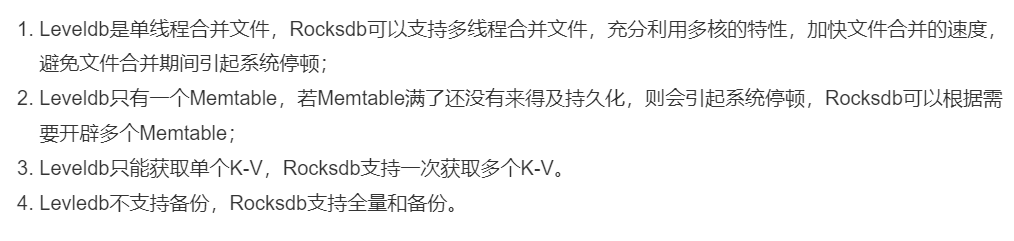
数据首先会写入到内存中一个叫做memtable的结构中，这是一个在内存中的可修改文件，其底层实现是一个跳表(SkipList)。当写入的内容超过某个阈值大小的时候，会新增一个可修改内存文件，新的数据会写入这个新的内存文件中，之前的内存文件执行持久化操作，加上不可修改的限定。这个就的内存文件会写入SStable文件，也就是Sorted String Table，意思是数据按照键的顺序存储在磁盘上，有顺序就有大小，levelDB把最大值和最小值记录下来，可以在查询的时候快速判断要查询的key可能在哪个SSTable文件当中，从而加快查询效率，这个就是MANIFEST。

这样子可以满足高效写的要求，但是涉及到IO，读就变慢了，随着写入越来越多，查找的磁盘文件就多了，因为键是有序的，所以按照键，从逻辑上对写入磁盘的文件进行了一个分层，一层如果太多了，效率其实也不高，所以会有一个阈值，文件数多的时候会对文件做一个归并，压入下一层，上一层就可以做一个类似索引的效果，每一层是上一层数量的十倍，通常情况下三层就可以满足数据要求。

## LSM-Tree和B+的区别

1. LSM-Tree的设计思路是将数据拆分为多个大段，并且采用顺序写入的方式；B+将数据拆分为固定大小的Block，一般是4KB大小，和磁盘一个扇区的大小对应，Page是读写的最小单位。
2. 在数据的更新和删除方面，B+Tree可以做到原地更新和删除，这种方式对数据库事务支持更加友好，因为一个key只会出现一个Page页里面，但由于LSM-Tree只能追加写，并且在L0层key的rang会重叠，所以对事务支持较弱，只能在Segment Compaction的时候进行真正地更新和删除。
3. LSM-Tree的优点是支持高吞吐的写（可认为是O（1）），这个特点在分布式系统上更为看重，针对读取LSM-Tree可以达到O（logN）的复杂度；而B+tree的优点是支持高效的读（稳定的OlogN），但是在大规模的写请求下（复杂度O(LogN)），效率会变得比较低，因为随着insert的操作，为了维护B+树结构，节点会不断的分裂和合并。操作磁盘的随机读写概率会变大，故导致性能降低。

## RocksDB的优势点



## Hbase数据库

**Hbase是一个高效的分布式数据库，因为HBase是在HDFS的基础之上构建的，HDFS是分布式文件系统**。HBase在HDFS之上提供了**高并发的随机写和支持实时查询**，这是HDFS不具备的。Hbase不同与普通的数据库，采用了列族存储的方式，**HBase一张表的数据会分到多台机器上的。**

**Client**客户端提供了访问HBase的接口，并且维护了对应的cache来加速HBase的访问。而**Zookeeper**存储HBase的元数据（meta表）索引，无论是读还是写数据，都是去Zookeeper里边拿到meta元数据的索引**告诉给客户端去哪台机器读写数据**。**HRegionServer**它是处理客户端的读写请求，负责与HDFS底层交互，是真正干活的节点。

总结大致的流程就是：client请求到Zookeeper，然后Zookeeper返回HRegionServer地址给client，client得到Zookeeper返回的地址去请求HRegionServer，HRegionServer读写数据后返回给client。

## 文件存储、块存储、对象存储

**文件存储**：文件存储表示用户根据文件路径，文件名组合在一起作为唯一标识符来访问具体的文件。而又多个文件及目录、子目录组合起来的这些数据结构叫做文件系统，同样在网络中，不同的服务器可以通过类似文件系统的方式访问这台服务器的文件，这样的系统焦作网络文件系统。

**块存储**：计算机的块设备通常划分为多个逻辑块设备，每一块赋予一个用于寻址的编号，它不需要去关心这些数据的组织方式和结构，由文件系统去决定其组织方式，所以块存储的使用者一般是文件系统、分区软件以及数据库等。

**对象存储**：相比于文件存储的树状结构、以及块存储的排他性（假如被某个客户端挂载后其他客户端无法访问的情况），对象存储提供了一个统一的底层存储系统管理这些文件和底层介质的组织结构，然后给每个文件一个唯一的标识，其它系统需要访问某个文件，直接提供文件的标识就可以了。

这么设计的好处就是简化了对象ID到对象数据之间的映射关系（索引）。比如说，你需要维护对象ID到数据起始位置之间的映射关系即可，而不像文件那样需要考虑因修改、插入、删除文件中的某一小段数据而造成的复杂映射关系。正因为映射关系简单了，同样规模的索引可以指向更大的数据规模，从而给人一种对象存储能存储更多数据的感觉。

## 服务器收不到客户端消息如何排查

1，抓包。分别抓客户端和服务端的网络数据包，看数据是否发送出去或者接收到。（TCP协议是可靠的，不存在丢包问题）

2，如果服务端没有发出响应包，则排查服务端的发包流程是否正常。

3，如果服务端有发出数据包，客户端也有收到，那么排查客户端的收包流程是否正常，特别是粘包，拆包的处理。

# 场景题

## 延时任务实现

1. 数据库轮询，将订单保存在数据库当中，然后设定一个定时任务，一段时间之后定时的扫描数据库比对订单的时间判断是否过期。（缺点）轮询数据库的方式效率低下，每次都需要扫描很多的记录，同时如果轮询数据库的方式会对部分过期订单的处理不够及时，有的订单可能扫描之后立马过期了，但是这个时候系统是不感知的，只能等到下一次扫描才能处理订单。
2. 使用环形队列可以实现高效的延时消息，环形队列其实就一个数组，可以从逻辑上实现环形功能，每一个格子上用链表连接一个订单的集合，按照时间轮算法的思想来实现延时消息，例如对于五分钟的延时任务，可以创建一个大小为300的环形队列，在启动的时候启动一个时针，每隔一秒时针就往前移动一格，例如在1的时候来了一个任务，需要将这个任务保存到2的这个格子上，因为300秒正好回到1，下一秒一定执行这个任务，对于任务来说保存的应该是一个数据结构，包括了执行时间（对应的执行函数，订单编号等），时钟每到一个就可以扫描整个链表，比较保存的数据结构的执行时间，超时即执行。使用环状队列可以很好的避免对于数据表的全表扫描，而且实时性可以精确到秒。

## 秒杀系统搭建（高可用，高并发）

（1）秒杀页面的处理，可以从以下几个方面考虑优化：

> 动静分离。 将页面的静态资源等部署到Nginx或者CDN（内容分发网络），这样可以加快秒杀页面获取。

> 静态资源合并获取。通过将多个请求合并为单个请求，一次获取多个静态资源，这样可以加快秒杀页面获取。

> 服务降级。秒杀页面做服务降级处理，将商品推荐列表、评论等做降级处理，少显示或者不显示。秒杀页面需要登录才能查看，对未登录用户直接返回登录界面。

> 服务监控。对流量进行监控，使用令牌桶算法等限流算法对流量进行控制。有必要时将部分任务进行熔断。

> 页面数据缓存。将页面数据缓存到Redis中，减少数据库操作

（2）商品下单，可以考虑从以下的几个方面做出优化：

> 防机器人刷单。对下单操作增加填写验证码步骤，如：55+44=？、“你好”的小写拼音、选出所有飞机等问题，将非法请求过滤掉。

> 商品下单预扣库存。数据库表设计的时候需要设置锁库存字段。进行秒杀的时候，减少库存将在Redis中使用分布式锁进行操作。其它后续操作可以使用MQ进行操作。

> 商品下单可以进行异步操作，如双次验价等操作可以使用多线程。

（3）支付的角度，可以从以下的角度优化方案：

> 将支付划分为一个单独的系统，只开放对应的支付接口。因为支付系统是金融敏感的，所以应该保证支付系统的高可用。

> 回滚机制。建议使用分布式事务，对支付业务进行TCC事务（补偿型事务），因为支付系统是金融敏感的。

同时，秒杀系统一般会引入MQ、Redis、MySQL、Nginx等中间件，需要对每个中间件进行高性能、高并发、高可用的分析。

# Linux和git常用命令

## Linux常用命令集合

Top -H -p <pid>：能查看相应进程内的线程情况

cat /proc/cpuinfo：查看主机核数

cat /proc/meminfo：查看内存剩余情况

Top：显示内存、CPU、进程的情况

Linux使用top命令输出程序使用的虚拟内存、物理内存和共享内存情况，如果想具体分析物理内存等具体使用信息，可以查看proc文件提供的smaps文件，proc文件是linux内核文件和用户空间交换文件的重要方式。

Ps -aux：显示进程的端口占用情况(netstat)

Pmap：显示对应进程的内存映射关系

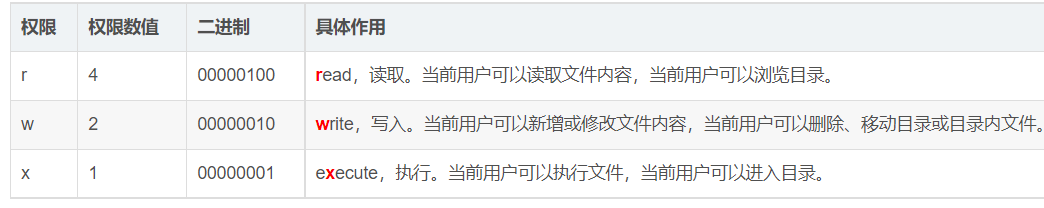
Netstat：显示路由表、实际的网络连接以及每一个网络接口设备的状态信息。

-a (all) 显示所有选项，默认不显示LISTEN相关。  
-t (tcp) 仅显示tcp相关选项。  
-u (udp) 仅显示udp相关选项。  
-n 拒绝显示别名，能显示数字的全部转化成数字。  
-l 仅列出有在 Listen (监听) 的服务状态。

-p 显示建立相关链接的程序名  
-r 显示路由信息，路由表  
-e 显示扩展信息，例如uid等  
-s 按各个协议进行统计  
-c 每隔一个固定时间，执行该netstat命令。

Chmod命令：每个用户的角色和权限划分都有特殊规定，利用chmod来决定某个用户通过某种方式对文件（目录）进行读、写、执行等操作。

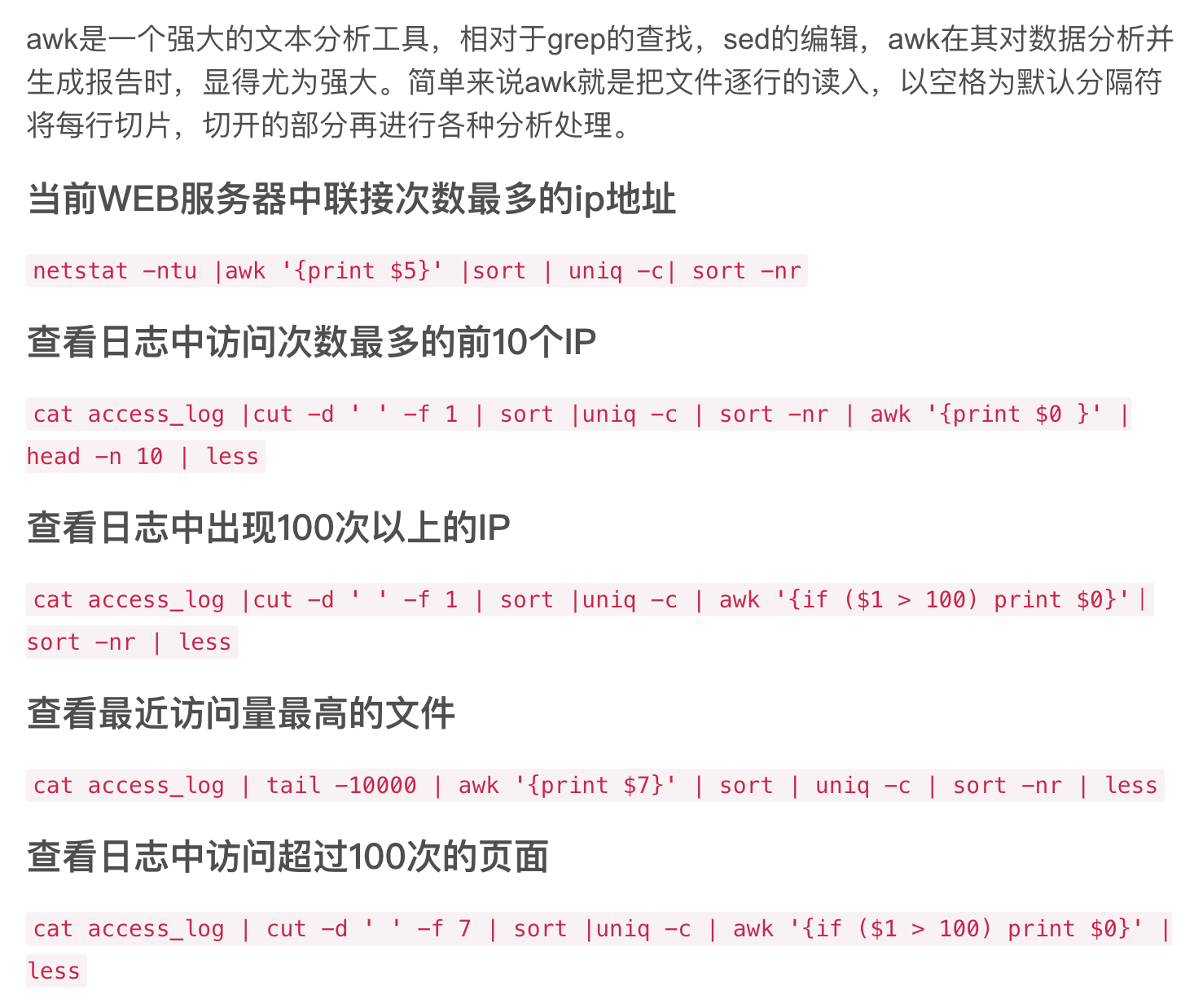
操作文件或目录的用户，有3种不同类型：文件所有者、群组用户、其他用户。最高位表示文件所有者的权限值，中间位表示群组用户的权限值，最低位则表示其他用户的权限值，所以，chmod 777中，三个数字7分别对应上面三种用户，权限值都为7。

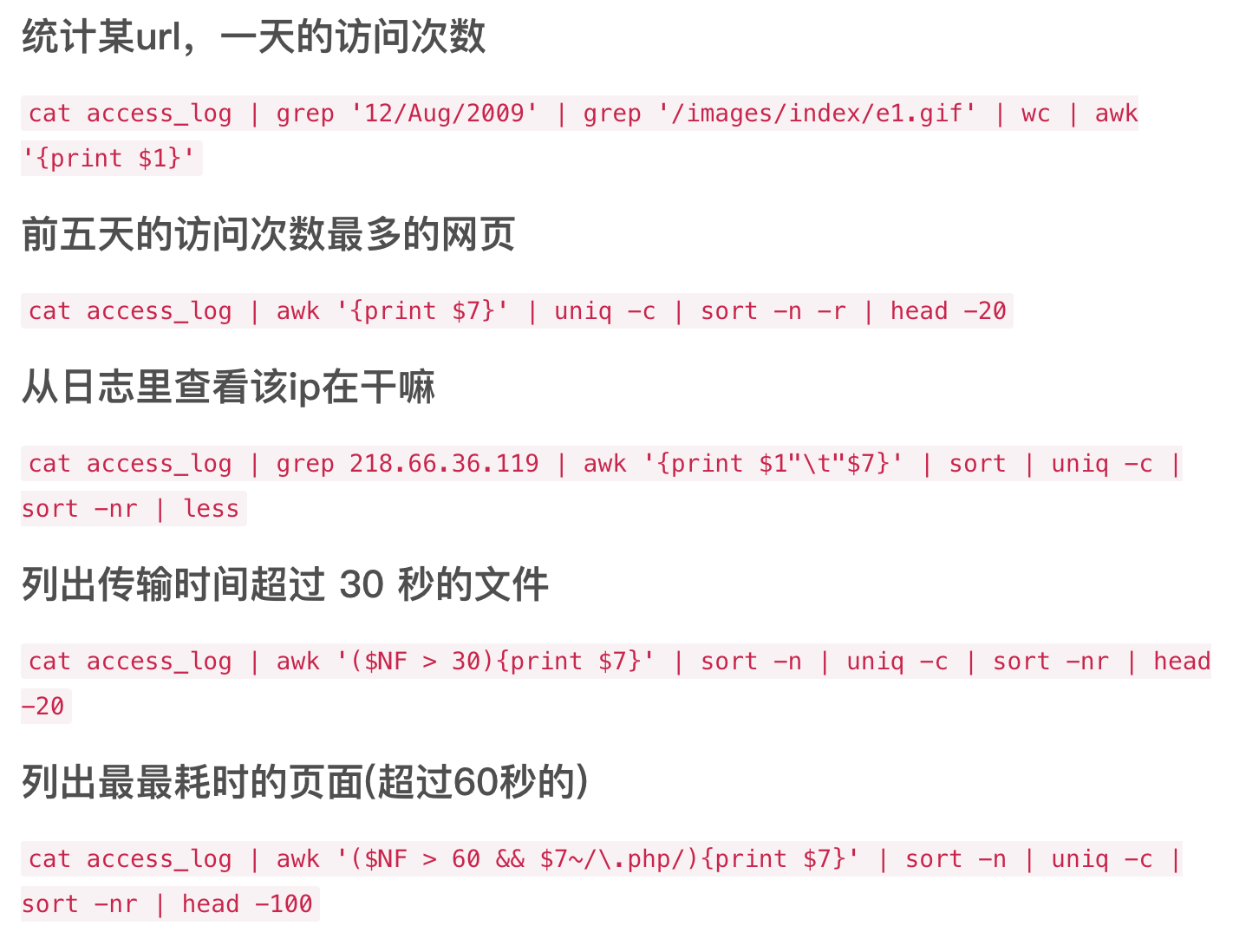


## Linux top命令实现原理

* 打开“/proc/stat”文件
* 读取该文件的首行，该行记录了从系统启动以来，cpu的使用信息

## Linux 统计日志里面出现最多的IP





## Linux如何查看日志信息

可以用head或者tail获取前面/后面的日志信息，一般会配合grep使用，查找复合条件的字符串

Sed可以根据行号/时间范围去查找。

## Git常用命令

git init 在当前目录新建一个仓库

git clone [url] 克隆一个远程仓库

git diff [local branch] origin/[remote branch] 比较本地分支和远程分支的区别

git status 查看文件状态

git add . 将工作区的被修改的文件和新增的文件提交到暂存区，不包括被删除的文件

git add -A . A指all，将工作区被修改、被删除、新增的文件都提交到暂存区

git commit -m [massage] 将暂存区所有文件添加到本地仓库

git log 显示所有commit日志

git push 将文件添加到远程仓库

git push -f 强制提交，当我们本地reset到旧的版本时，然后普通push会被拦截，因为是本地比远程库还要旧

git push origin [branch-name] 推送当前本地分支到指定远程分支

git branch 查看当前分支

git checkout [branch] 切换分支

git merge [branch-name] 用于合并指定分支到当前分支

# 数据结构相关

## find的使用

找子串在母串中的位置s.find("jk") == s.npos; npos表示未找到，成功找到则返回首位置。

找到首次和末次出现的位置s.find\_first\_of(flag); 和 s.find\_last\_of(flag);

从某个固定位置开始找s.find("b",5);

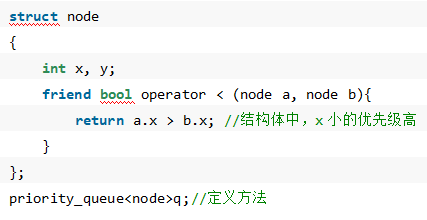
反向开始找s.rfind("jk");

## 优先队列

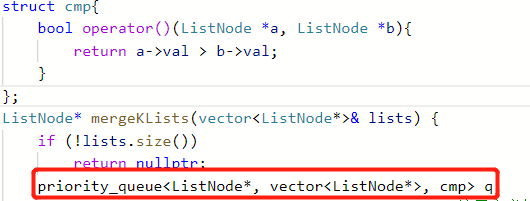
(1)默认是从大到小输出。

(2)从小到大：priority\_queue<**int**,vector<**int**>,greater<**int**> >q;

(3)结构体，重写结构体里面的小于号就行了



(4)结构化的第二种方式（针对链表如何定义？）



## 归并排序

归并排序是稳定的吗？时间空间复杂度是多少？

归并排序是稳定的，且这个时间复杂度是稳定的，不随需要排序的序列不同而产生波动。因为归并排序是一个分而治之的思想，归并排序的过程中，需要对当前区间进行对半划分，直到区间的长度为1。也就是说，每一层的子区间，长度都是上一层的1/2。这也就意味着，当划分到第logn层的时候，子区间的长度就是1了。而归并排序的merge操作，则是从最底层开始（子区间为1的层），对相邻的两个子区间进行合并，这个合并的过程需要扫描所有的区间，时间复杂度是o(n)，所以总的时间复杂度是很稳定的O(nlogn)。

## 前缀树解释

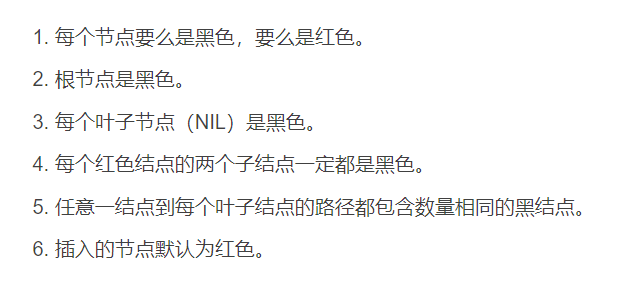
前缀树是N叉树的一种特殊形式。通常来说，一个前缀树是用来存储字符串的。前缀树的每一个节点代表一个字符串（前缀）。每一个节点会有多个子节点，通往不同子节点的路径上有着不同的字符。子节点代表的字符串是由节点本身的原始字符串，以及通往该子节点路径上所有的字符组成的。**Trie的核心思想是空间换时间。利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。**

## 跳表解释

为了存储多个不同行的数据，我们可以用链表将他们串起来，这样查询数据的时间复杂度是O（n），于是我们可以将链表中的部分结点拿出来构建出一个新的链表放在其上层，根据这种思路依次以二分的形式构建上层链表，查询的时候可以根据一层一层的查询实现类似二分的查找，能够有效提高检索速度，跳表通过牺牲空间换取时间的方式提升查询性能，查询的时间复杂度变为log（n）。跳表的插入是当插入最底层的链表后，是否需要在上面几层插入索引靠随机函数，理论上为了达到二分的效果，每一层的结点数是下一层的一半，因此也是以二分之一的概率依次向下进行插入，因此每次插入数据后会以随机函数的方式随机其插入的索引层数，当数据足够多的时候，数据的分布也符合了我们的“二分”要求。

## 红黑树解释

红黑树是一个非严格的平衡二叉树，红黑树中节点要么标记为红色，要么标记为黑色，红黑树的根节点和空叶子节点为黑色，每个根节点到叶子节点经过的黑色节点个数一样，是一个接近于平衡的一个二叉树。



## 红黑树和跳表相比的优缺点

**红黑树优点：**不像是AVL一样追求极致平衡，减少了旋转次数，如果搜索，插入删除次数几乎差不多，红黑树还是比较适合的。

**红黑树缺点：**数据量大,树高度会很高导致查找速度慢

**跳表的优点：**因为是从上层主键比较，所以能像是有序数组的二分查找，提高查询效率。

**跳表的缺点：**需要额外维护多个链表，暂用额外链表的空间。