1. **系统：进程与线程有什么区别**

* 进程是资源分配的最小单位，线程是程序执行的最小单位
* 线程是进程的一个执行流，一个进程可以由很多个线程组成
* 进程有独立地址空间，而线程是共享进程中的数据，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
* 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
* 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

1. **系统：线程池是什么**

* 线程池就是事先将线程放到一个容器中，当使用线程的时候，不用再去new出一个线程，直接从线程池取出来就可以了
* 优点1：降低资源消耗:重复利用线程池中的线程节省线程创建和销毁带来的消耗；
* 优点2：提高性能：当任务需求时，可以不用创建线程直接执行，主要是直接从线程池中取出线程去执行；
* 优点3：提高线程的可管理性：线程是稀缺资源，而且也是任务中不可少的资源，如果频繁的且无限制的创建会消耗系统资源，降低系统稳定性导致系统崩溃，内存溢出等等问题
* Java提供了多种创建线程池的方式:

ExecutorService executors = Executors.newFixexThreadPool(3);创建固定大小的线程池，例子中只能放下3个线程

1. **系统：线程安全和线程不安全**

* 线程安全就是说多线程访问同一代码，不会产生不确定的结果。编写线程安全的代码是低依靠线程同步。
* 线程安全的类：StringBuffer 、HashTable（和HashMap功能都一样，唯一不一样的是：HashTable为线程安全和key/value不可以为null值）、Vector（JDK1.0时候出来）、stack（栈先进后出）。C中的malloc和free
* 线程不安全：StringBuilder(相比StringBuilder效率更高)、HashMap（可以有null，值但仅能有一个）、ArrayList

1. **网络：TCP和UDP的区别**

* TCP面向连接（如打电话要先拨号建立连接），通过三次握手; UDP是无连接的，即发送数据之前不需要建立连接（发短信）
* TCP提供可靠的服务（APP客户端）。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达; UDP尽最大努力交付，即不保证可靠交付
* TCP面向字节流，是流模式（直播，强调实时性），实际上是TCP把数据看成一连串无结构的字节流; UDP是面向报文的，是数据报模式
* UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）
* 每一条TCP连接只能是点到点的; UDP支持一对一，一对多，多对一和多对多的交互通信
* TCP首部开销20字节; UDP的首部开销小，只有8个字节
* TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道
* TCP对系统资源要求较多，适合传输大量数据; UDP对系统资源要求少，适合传输少量数据

全双工即Full duplex Communication,是指在通信的任意时刻,线路上存在A到B和B到A的双向信号传输

1. **Cpp：指针和引用的区别**

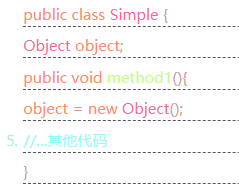
* 指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元，即指针是一个实体；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名
* 两者都是地址的概念；指针指向一块内存，它的内容是所指内存的地址；引用是某块内存的别名。
* 可以有const指针，但是没有const引用；
* 指针可以有多级，但是引用只能是一级（int \*\*p；合法 而 int &&a是不合法的）
* 指针的值可以为空，但是引用的值不能为NULL，并且引用在定义的时候必须初始化；
* 指针的值在初始化后可以改变，即指向其它的存储单元，而引用在进行初始化后就不会再改变了，从一而终。
* ”sizeof引用”得到的是所指向的变量(对象)的大小，而”sizeof指针”得到的是指针本身的大小；
* 指针和引用的自增(++)运算意义不一样

1. **Cpp：函数返回局部变量的引用会怎么样**

* 当函数返回引用类型时，没有复制返回值，相反，返回的是对象本身。
* 千万不要返回局部对象的引用！千万不要返回指向局部对象的指针！当函数执行完毕时，将释放分配给局部对象的存储空间。此时对局部对象的引用会指向不确定的内存！返回指向局部对象的指针也是一样的，当函数结束时，局部对象被释放，返回的指针就变成了不再存在对象的悬垂指针。
* 返回引用时，要求在函数的参数中，包含有以引用方式或指针方式存在的，需要被返回的参数。

1. **Cpp：内存泄露是什么**

* 不再会被使用的对象的内存不能被回收，就是内存泄露
* 在C++中，所有被分配了内存的对象，不再使用后，都必须程序员手动释放他们。所以，每个类，都会含有一个析构函数，作用就是完成清理工作，如果我们忘记了某些对象的释放，就会造成内存泄露。
* C++中，RAII可以用来防止内存泄漏
* 但是在Java中，我们不用（也没办法）自己释放内存，无用的对象由GC自动清理，这也极大的简化了我们的编程工作。但，实际有时候一些不再会被使用的对象，在GC看来不能被释放，就会造成内存泄露



1. **Cpp：析构函数是不是虚函数**

* 构造函数不能为虚函数，而析构函数可以且常常是虚函数。
* C++对象在三个地方构建：（1）函数堆栈；（2）自由存储区，或称之为堆；（3）静态存储区。无论在那里构建，其过程都是两步：首先，分配一块内存；其次，调用构造函数。好，问题来了，如果构造函数是虚函数，那么就需要通过vtable 来调用，但此时面对一块 raw memeory，到哪里去找 vtable 呢？毕竟，vtable 是在构造函数中才初始化的啊，而不是在其之前。因此构造函数不能为虚函数。
* 析构函数可以是虚函数，且常常如此。因为此时 vtable 已经初始化了；况且我们通常通过基类的指针来销毁对象，如果析构函数不为虚的话，就不能正确识别对象类型，从而不能正确销毁对象。
* 每个析构函数（不加 virtual） 只负责清除自己的成员。但可能有基类指针，指向的确是派生类成员的情况。（这是很正常的）。那么当析构一个指向派生类成员的基类指针时，程序就不知道怎么办了。所以要保证运行适当的析构函数，基类中的析构函数必须为虚析构。

1. **Cpp：父类先析构还是子类先析构**

* 构造的时候先构造基类后子类 析构的时候相反先析构子类后析构基类

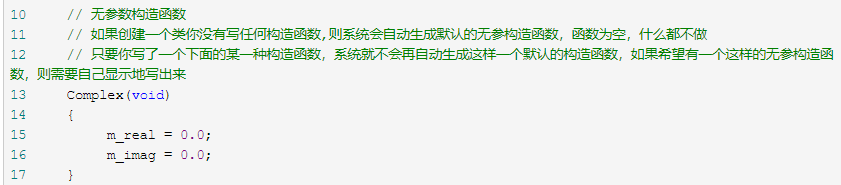
1. **Cpp：虚函数是什么**

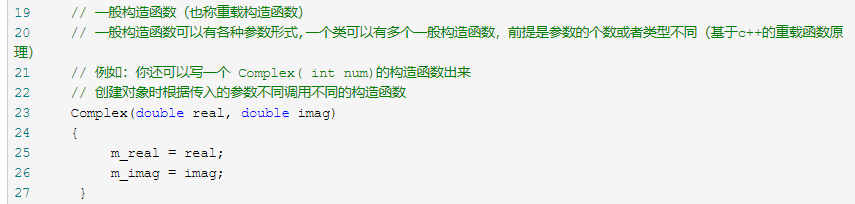
* 面向对象的语言有三大特性：继承、封装、多态。虚函数作为多态的实现方式，而多态体现在运行时绑定。通常通过基类指针指向子类对象实现多态。
* 其实更简单地来说，就是“在用父类指针调用函数时，实际调用的是指针指向的实际类型（子类）的成员函数”。
* 多态性使得程序调用的函数是在运行时动态确定的，而不是在编译时静态确定的。而虚函数则是加了virtual修饰词的类的成员函数。
* 什么样的函数不能声明为虚函数？1）不能被继承的函数。2）不能被重写的函数

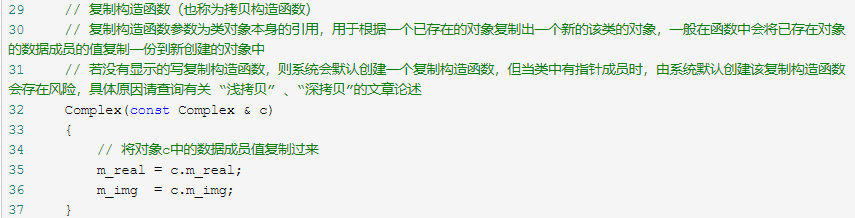
比如普通函数（只能重载不能重写），友元（不能继承），构造（不能继承），内联成员函数（编译展开），静态成员函数（编译确定，不能重写）

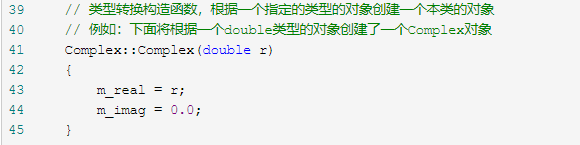
1. **Cpp：拷贝构造函数**

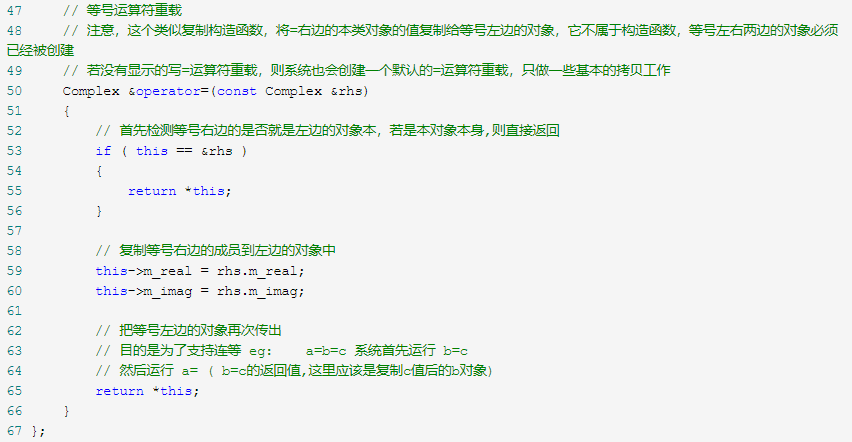
几种构造函数：无参数构造函数，一般构造函数，复制构造函数，类型转换构造函数，等号运算符重载











1. **Cpp：重载和重写**

方法重写(overriding)：

* 重写也叫子类的方法覆盖父类的方法，要求返回值、方法名和参数都相同。
* 子类抛出的异常不能超过父类相应方法抛出的异常。(子类异常不能超出父类异常)
* 子类方法的的访问级别不能低于父类相应方法的访问级别(子类访问级别不能低于父类访问级别)

方法重载(overloading):

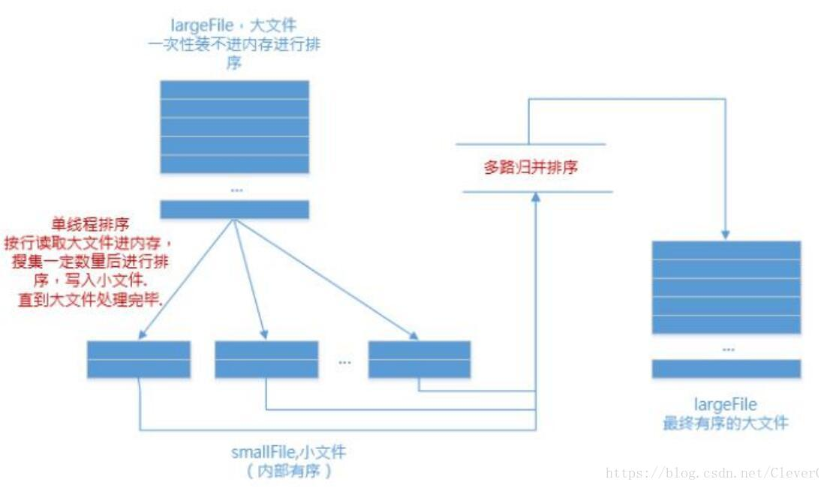
* 重载是在同一个类中的两个或两个以上的方法，拥有相同的方法名，但是参数却不相同，方法体也不相同，最常见的重载的例子就是类的构造函数，可以参考API帮助文档看看类的构造方法

1. **Cpp：内联函数和宏定义**

* 使用宏和内联函数都可以节省在函数调用方面所带来的时间和空间开销。二者都采用了空间换时间的方式，在其调用处进行展开
* 在预编译时期，宏定义在调用处执行字符串的原样替换。在编译时期，内联函数在调用处展开，同时进行参数类型检查。
* 内联函数首先是函数，可以像调用普通函数一样调用内联函数。而宏定义往往需要添加很多括号防止歧义，编写更加复杂。
* 内联函数可以作为某个类的成员函数，这样可以使用类的保护成员和私有成员。而当一个表达式涉及到类保护成员或私有成员时，宏就不能实现了(无法将this指针放在合适位置)。
* 可以用内联函数完全替代宏。在编写内联函数时，函数体应该短小而简洁，不应该包含循环等较复杂结构，否则编译器不会将其当作内联函数看待，而是把它决议成为一个静态函数。有些编译器甚至会优化内联函数，通常为避免一些不必要拷贝和构造，提高工作效率。频繁的调用内联函数和宏定义容易造成代码膨胀，消耗更大的内存而造成过多的换页操作。

1. **算法：如果要排序一个数组，1G，但只有100Mb的空间，用哪种排序算法**

* 多路归并排序



1. **算法：能说出几种排序方法**

* 插入排序，希尔排序，选择排序，堆排序，冒泡排序，快速排序
* 归并排序，基数排序，计数排序，桶排序



1. **数据库：事务是什么**

* 事务（Transaction)是由一系列对系统中数据进行访问与更新的操作所组成的一个程序 执行逻辑单元（Unit)。
* 狭义上的事务特指数据库事务。一方面，当多个应用程序并发访问数据库时，事务可以在这些应用程序之间提供一个隔离方法，以防止彼此的操作互相干扰。另一方面，事务为数据库操作序列提供了一个从失败中恢复到正常状态的方法， 同时提供了数据库即使在异常状态下仍能保持数据一致性的方法。
* A，Automicity，原子性，强调事务作为原子级别已经不可以再被分割,要么成功要么失败。
* C，Consistency，一致性，即状态转换必须是由一种正确的状态转换到另外一种正确的状态。
* I，Isolation，隔离性，即相互间必须不能被影响。
* D，Durabillity，持久性，即事务提交后将被永久保存，即便出现其他故障，事务处理结果也应得到保存。

1. **系统：进程间通讯（IPC）有哪些实现方法**

* 管道：通常指无名管道，是UNIX系统IPC最古老的形式

它是半双工的（即**数据只能在一个方向上流动）**，具有固定的读端和写端。

它只能用于具有**亲缘关系的进程**之间的通信（也是父子进程或者兄弟进程之间）。

它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

* FIFO，也成为明明管道，是一种文件类型

FIFO可以在**无关的进程**之间交换数据，与无名管道不同。

FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

* 消息队列：是消息的链列表，存放在内核中，一个消息队列由一个标识符（即队列ID）来标识

消息队列是**面向记录**的，其中的消息具有**特定的格式以及特定的优先级**。

消息队列**独立于发送与接收进程**。进程终止时，消息队列及其内容并不会被删除。

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。

* 信号量，与已经介绍过的 IPC 结构不同，它是一个计数器。信号量用于**实现进程间的互斥与同步**，而不是用于存储进程间通信数据。

信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

信号量基于**操作系统的 PV 操作**，程序对信号量的操作都是原子操作。

每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

支持信号量组。

* 共享内存，指两个或多个进程共享一个给定的存储区

共享内存是最快的一种 IPC，因为进程是直接对内存进行存取。

因为多个进程可以同时操作，所以**需要进行同步**。

**信号量+共享内存**通常结合在一起使用，信号量用来同步对共享内存的访问。

1. **Cpp：纯虚函数的作用是什么**

定义纯虚函数是为了实现一个接口，起到一个规范的作用，规范继承这个类的程序员必须实现这个函数。

Virtual xxx () = 0;

纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0”

抽象类是一种特殊的类，它是为了抽象和设计的目的为建立的，它处于继承层次结构的较上层。抽象类的定义： 称带有纯虚函数的类为抽象类。

1. **Cpp：虚函数在底层具体是如何实现的**

虚函数将实现方法留给了编译器

编译器的处理方法是：给每个对象添加一个隐藏成员，隐藏成员中保存了一个指向函数地址数组的指针。这个数组成为虚函数表vtbl。

虚函数表中存储了未类对象进行声明的虚函数的地址，如果派生类提供了虚函数的新定义，该虚函数将保存新函数的地址

调用虚函数时，程序将查看存储在对象中vtbl地址，然后转向响应的函数地址表

使用虚函数时，在内存和执行速度方面有一定的成本，包括

1. 每个对象都将增大，增大量为存储地址的空间
2. 对于每个类，编译器都创建一个虚函数地址表的数组
3. 对于每个函数调用，都需要执行一项额外的操作，即到表中查找地址

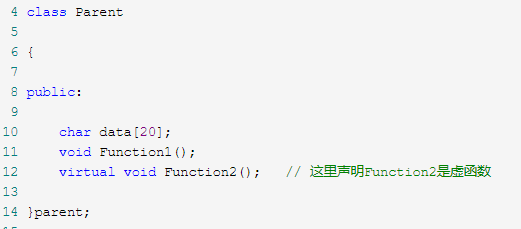
但是正是因为如此，虚函数才拥有了动态联编的能力

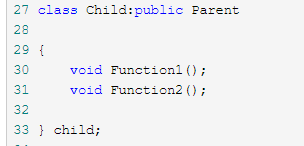
1. **Cpp：虚函数的使用方法**

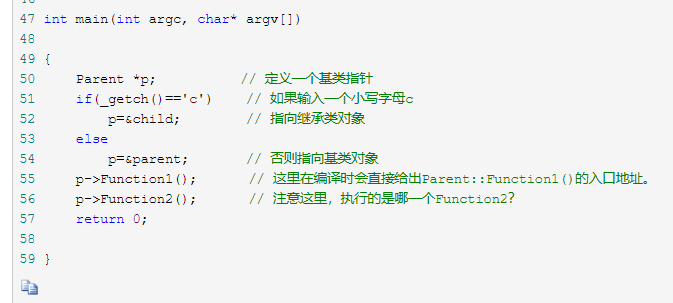
* 在基类用virtual声明成员函数为虚函数。这样就可以在派生类中重新定义此函数，为它赋予新的功能，并能方便地被调用。在类外定义虚函数时，不必再加virtual。
* 在派生类中重新定义此函数，要求函数名、函数类型、函数参数个数和类型全部与基类的虚函数相同，并根据派生类的需要重新定义函数体。

C++规定，当一个成员函数被声明为虚函数后，其派生类中的同名函数都自动成为虚函数。因此在派生类重新声明该虚函数时，可以加virtual，也可以不加，但习惯上一般在每一层声明该函数时都加virtual，使程序更加清晰。如果在派生类中没有对基类的虚函数重新定义，则派生类简单地继承其直接基类的虚函数。

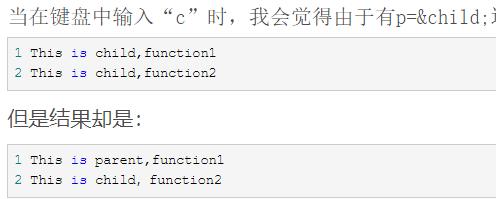
* 定义一个指向基类对象的指针变量，并使它指向同一类族中需要调用该函数的对象。
* 通过该指针变量调用此虚函数，此时调用的就是指针变量指向的对象的同名函数。通过虚函数与指向基类对象的指针变量的配合使用，就能方便地调用同一类族中不同类的同名函数，只要先用基类指针指向即可。如果指针不断地指向同一类族中不同类的对象，就能不断地调用这些对象中的同名函数。这就如同前面说的，不断地告诉出租车司机要去的目的地，然后司机把你送到你要去的地方。







虚函数最关键的特点是“动态联编”，它可以在运行时判断指针指向的对象，并自动调用相应的函数。





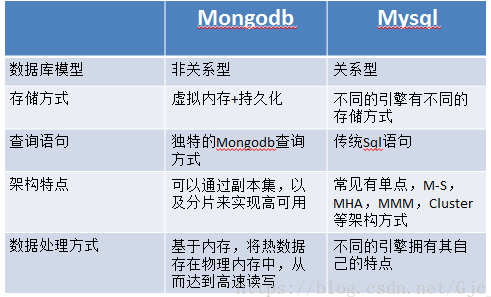
1. **Cpp：对stl库了解多少**

* STL是标准模板库的意思 standard template library
* 几个基本概念：容器 list，stack, vector, deque, queue, map, set

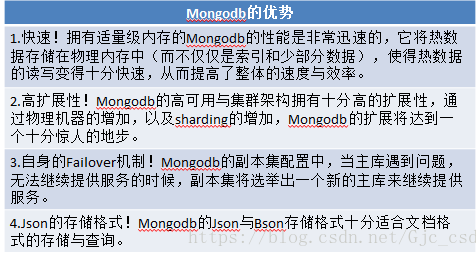
游标 iterator

算法 sort, count, for\_each, find 等

1. **数据库：Mongodb和mysql各有什么特点**

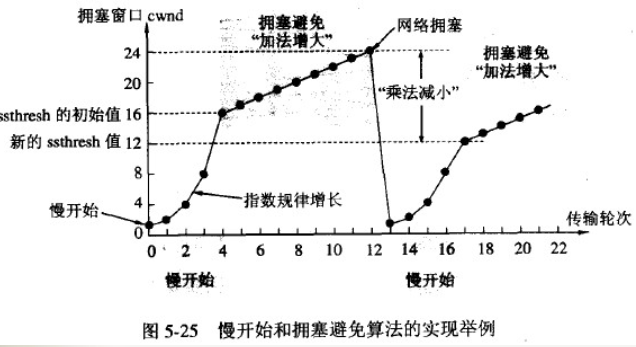


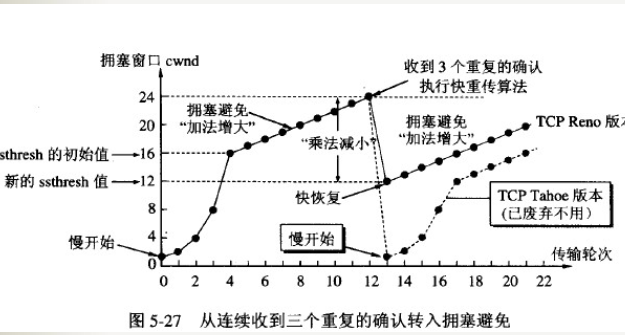




1. **网络：TCP拥塞控制的机制**

慢启动，快恢复





1. 网络：Lock\_client如果挂了怎么办
2. 网络：如果网络通信不畅的话，怎么防止误伤
3. **数据库：如何保证数据库的强一致性**

数据库一致性（Database Consistency）是指事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。保证数据库一致性是指当事务完成时，必须使所有数据都具有一致的状态。在关系型数据库中，所有的规则必须应用到事务的修改上，以便维护所有数据的完整性。

更新缓存策略：

* 失效：应用程序从cache取数据，没有得到，则从数据库取数据，然后将数据放入cache
* 命中：应用从cache取数据，若取到则返回
* 更新：把数据更新到数据库，然后使cache失效

（更新）如果先使cache失效，后更新数据库，是否可行

这么做引发的问题是，如果A,B两个线程同时要更新数据，并且A,B已经都做完了删除缓存这一步，接下来，A先更新了数据库，C线程读取数据，由于缓存没有，则查数据库，并把A更新的数据，写入了缓存，最后B更新数据库。那么缓存和数据库的值就不一致了。

如果先把数据库更新，然后把cache更新（不是失效），是否可行

这么做引发的问题是，如果A,B两个线程同时做数据更新，A先更新了数据库，B后更新数据库，则此时数据库里存的是B的数据。而更新缓存的时候，是B先更新了缓存，而A后更新了缓存，则缓存里是A的数据。这样缓存和数据库的数据也不一致。

如果更新数据时，更新数据库成功，而失效cache时失败怎么解决？

1、对删除缓存进行重试，数据的一致性要求越高，我越是重试得快。

2、定期全量更新，简单地说，就是我定期把缓存全部清掉，然后再全量加载。

3、给所有的缓存一个失效期。（绝招，任何不一致，都可以靠失效期解决，失效期越短，数据一致性越高。但是失效期越短，查数据库就会越频繁。因此失效期应该根据业务来定）

1. 数据库：如何让大容量数据库的seek时间差不多（查询效率优化问题）
2. 数据库：如何避免数据堆积在同一个位置
3. React和小程序是怎么渲染的，一段js代码怎么成为最后呈现在用户眼前的内容
4. 进程的调度
5. 进程的生命周期
6. TCP的状态
7. B树与B+树

<https://blog.csdn.net/z_ryan/article/details/79685072>