## 面试

explain select \* from paper\_question\_v2 where paper\_id = 0 order by ea\_item\_id;

这两个是联合索引,一个where,一个order, 是完全走index,也不会有filesort

如果是两个单列索引的话,只会use where的index,后面的order是用的filesort,因为mysql只会用一个索引.联合索引就没有这个问题的.

Order by是可以走索引的,但是前面一般都有where条件,where条件那也是有索引的咯,由于mysql只能用一个索引,所以order的索引和where的索引一般都会用where的索引,但是如果你order by id的话就会导致mysql用了order后面的这个主键索引,但是不用where的索引,导致性能的低下了,一个索引的坑.

对比传统数据库仅仅记录了数据的当前值，时序数据库则记录了所有的历史数据。同时时序数据的查询也总是会带上时间作为过滤条件.

metric: 度量，相当于关系型数据库中的table。

data point: 数据点，相当于关系型数据库中的row。

timestamp：时间戳，代表数据点产生的时间。

field: 度量下的不同字段。比如位置这个度量具有经度和纬度两个field。

1个分片最多被一个worker处理数据，假如总共有20个worker但是只有10个分片，那就只有10个worker能处理数据，另外10个worker闲置

C语言是有个链接过程的,最后会形成单一的可执行文件MME,而java呢,编译完就会有一个个class文件,这些class文件会在运行的时候加载到虚拟机里,然后再动态链接起来.

JVM指令

加载系列:iload iload bipush sipush istore istore

运算系列:iadd isub imul idiv land求与 dcmpg比较

控制转移指令:ifeq iflt ifle ifne ifgt ifge ifnull ifnonnull lookupswitch goto

对象创建指令:new newarray

方法调用指令:invokevirtual调用虚方法 invokestatic调用静态方法  invokeinterface调用接口中的方法 invokespecial调用特殊方法,私有方法 父类方法.

抛出异常指令:athrow

但是会为java应用提高很大灵活性,java里天生可以动态扩展的语言特性就是依赖运行期间动态加载和动态链接这个特点实现的

链接就是把符号引用变成直接引用.

java虚拟机执行代码的时候,热点代码编译执行,冷代码解释执行(jvm指令-二进制机器码)

栈帧 操作数

所谓操作数栈就是指令把局部变量表的数据压入栈的地方.

所有的关键就在于invokevirtual这个指令太叼了!!!!!!!!

这货指令上来就会确定数据的实际类型比如是C!!!!!!然后拿方法的名字参数列表返回值去C里找对应的方法,这样就找到对的了.

而且这个指令肯定是在运行时才会触发,都没运行哪来的指令啊 .

所以多态的类型是运行时确定的,都靠这个指令啊!!!!!这个玩意就是动态分派.

虚方法表

CORS(跨域资源共享) CSRF(跨站伪造请求)

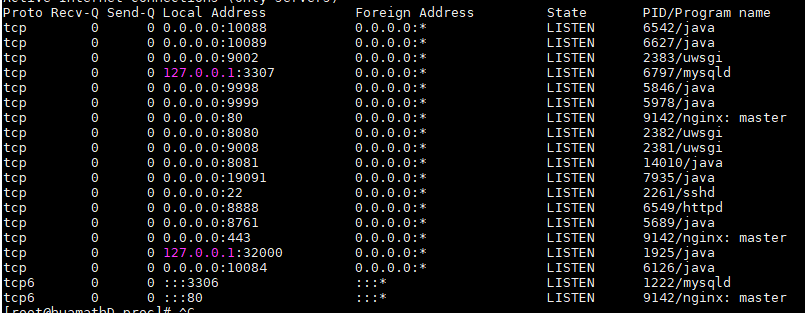
主要原因就是浏览器的同源策略如果你在www.qignbei.com服务器下别人给你写了个cookie的话,那么如果你点击了其他的域名,比如www.qing2bei.com的话,cookie是不会带过去的,如果cookie不能用的话,那么session就不能用了,所以如果你的网站有两个域名的话其中一个设置的cookie,另一个其实是不能用的,主要是cookie不能带到新的域名里面去.

如果要是可以跨域的话,那么就会有CSRF风险,比如你把你的cookie带到一个新的黑客域名里去,如果你的cookie里面有账户密码的话就比较危险了.

所以现在的用户的登陆信息基本都是用的token,不是用的session了,因为token的机制是不需要用cookie的,而是添加在http头里面,这个是可以跨域的.

而且你同一个网站域名为啥不一样呢??

netstat -ntpl 查看有哪些进程在监听tcp的端口,左边就有端口,右边有进程id



netstat -ntulp |grep 80   //查看所有80端口使用情况·

netstat -an | grep 3306 //查看3306端口的使用情况

CPU只操作寄存器里面的数据.

二分查找mid = high+low/2 这里会有int范围一处问题,要用

Mid = low + 1/2(high-low)

Windows的回车是/r/n unix的回车是/n

t1.join();

T1是以前的线程,当前线程是主线程,执行完这句的结果就是当前线程不执行了,等待t1线程执行完继续执行.

Linux系统的内存空间由两个部分构成：物理内存和SWAP（位于磁盘）。物理内存是Linux活动时使用的主要内存区域；当物理内存不够使用时，Linux会把一部分暂时不用的内存数据放到磁盘上的SWAP中去，以便腾出更多的可用内存空间；而当需要使用位于SWAP的数据时，必须先将其换回到内存中。

也就是说一旦查看发现swap了，就说明内存不足过。

Select 是三个套接字集合 read一个 write一个 出异常的一个。

Poll只有一个集合

Epoll也是三个集合，但是不会轮询所有套接字。

长链接基本就是websocket实现了,别的都不太好.

但是在没有消息的情况下，服务器先拖一段时间，等到有消息了再回复。这个机制暂时地解决了实时性问题，但是它带来了新的问题：以多线程模式运行的服务器会让大部分线程大部分时间都处于挂起状态，极大地浪费服务器资源。另外，一个HTTP连接在长时间没有数据传输的情况下，链路上的任何一个网关都可能关闭这个连接，而网关是我们不可控的，这就要求Comet连接必须定期发一些ping数据表示连接“正常工作”。websocket双工.

平衡二叉树,AVL树是一种特殊的二叉排序树,他的特殊之处就在于他的所有节点的平衡因子的绝对值都小于等于1.

平衡因子是指一个节点的左子树高度减右子树高度.

也就是说当你在往这个AVL树里面插入数据的时候,每插一个就要查看一下插入这个节点有没有破坏树的平衡,如果破坏了,就在插入后左旋右旋.

比如 1

2

3

这样1的左子树高度是0,右子树高度是2,就不行了,我们左旋一下变成

2

1 3

这样就平衡了,而且还是二叉排序树的结构.

但是注意这种

7

10

9

这种,这种是不能直接左旋的,因为左旋了9就跑10右边去了.

这种7的平衡因子是-2 10的平衡因子是1,这两个平衡因子符号不对,

我们就要先把这两个的平衡银子的符号调整成正确的,那么我们就要先把9和10交换位置,把这个变成

7

9

10 这样的话直接左旋就行了,这就是平衡二叉树.

所有叶节点(NIL)是空的黑色的。（NIL节点无数据，是空节点）

红色节点必须有两个黑色儿子节点。

从任一节点出发到其每个叶子节点的路径，黑色节点的数量是相等的

fork exec wait

fork()是全部复制，vfork()是共享内存，而clone()是则可以将父进程资源有选择地复制给子进程

Linux复制进程时候的写时复制技术,意思就是父子进程相同的数据就共享,如果被写入不同的东西了那么子进程就复制父进程的.

Fork复制任务 Join主进程等待子进程完,那么就可以利用join进行任务合并

selector的select

等待网络数据到达网卡->内核从网卡中读取数据到IO缓冲区.

从数据缓冲区复制数据到用户进程.

IO多路复用的机制就是,当accept到一个socket后,不去创建线程,而是将这些socket(链接)注册到多路复用器上面,专门搞一个线程不停的轮询复用器,看看哪些链接有请求来了,如果发现有请求的话就开启一个线程对这个请求进行处理,也就是一个请求一个线程.

//先创建一个Callable的实现类对象,我这个里面的call方法返回的是一个user

ACallable a = **new** ACallable();

//然后创建一个FutureTask<User>

FutureTask<User> task = **new** FutureTask<User>(a);

//然后把这个task当作参数创建一个thread并启动

**new** Thread(task).start();

//当这个线程启动后就相当于我们开了一个异步线程,在做一件事

//我们主线程现在可以继续的做自己的事

**for**(**int** i = 0;i<10;i++) {

System.***out***.println("我玩了一次,看你啥时候弄完" + i);

}

User user = task.get();

至于g()什么时候返回、f()怎么拿到值，提供两种方式：

1、future.get(),

2、以及第二种通常更优的方式：回调函数。

至于g()什么时候返回、f()怎么拿到值，提供两种方式：

1、future.get(),

2、以及第二种通常更优的方式：回调函数。

Mac地址是下一跳的地址老换

一个网卡只有一个mac,却可以有多个ip

ARP(网络层协议)通过ip找mac,也是一跳一跳的找.

由于链路层有路径MTU的存在,所以ip层会存在分片.

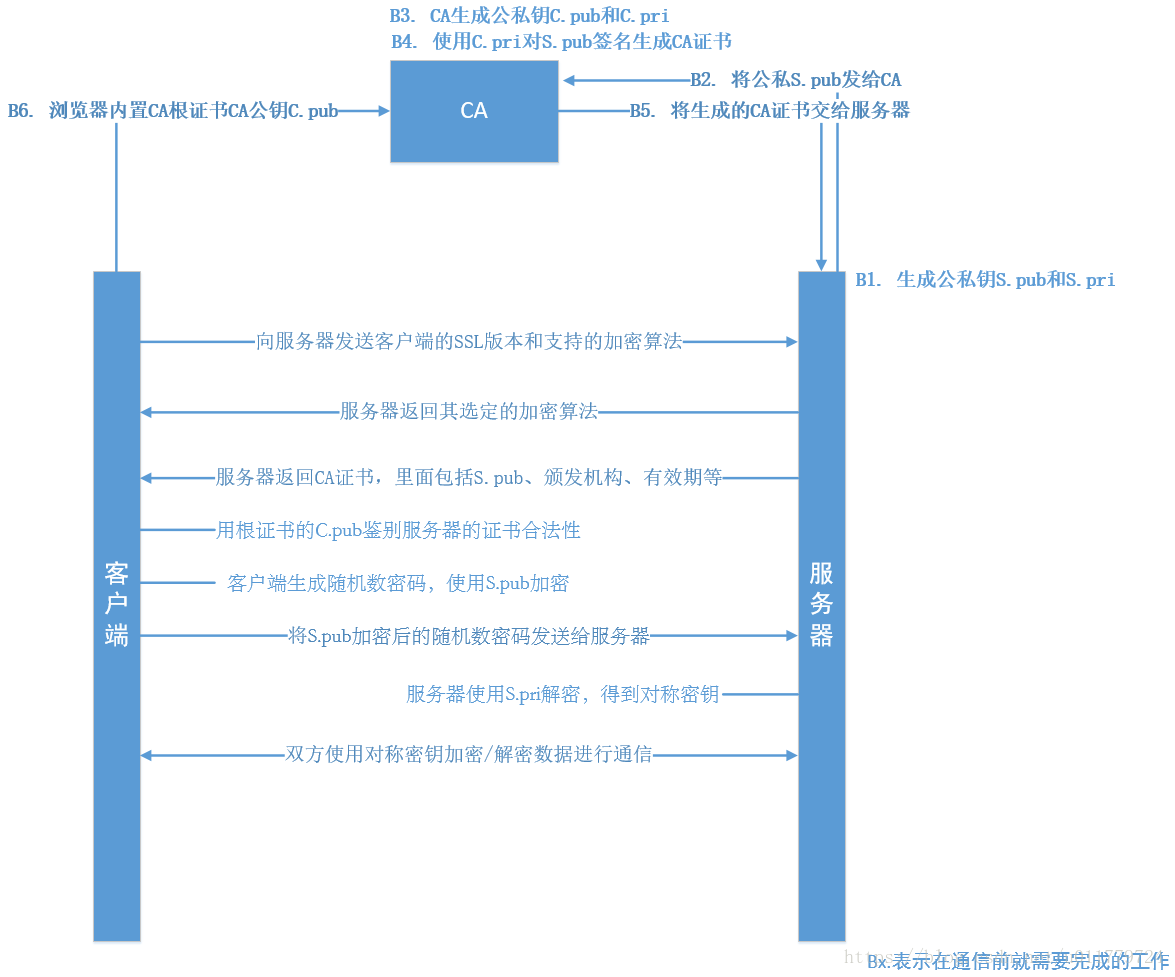
非对称加密 私钥加密 公钥可以解密 公钥加密 私钥也可以解密。

一般两端需要交互，都是先把自己的公钥给对方（公钥可以给很多人），私钥只有自己有，别人拿到了你的公钥，给你发消息时就会用公钥加密发给你，加密后的消息别人截获了也没有，别人没有你的私钥也解不了。

一般都是服务器把公钥发给用户，私钥服务端自己保存，用户收到服务器的公钥后把自己的公钥用服务器的公钥加密，发给服务器，服务器用自己的私钥解开后就拿到了用户的公钥，这样两端都保存了对方的公钥，发的消息都用对方的公钥加密了，通信就是安全的了。

问题就是第一步服务器把自己的公钥发给用户的时候还是不安全的。

解决方案就是服务端把自己的公钥告诉一个第三方机构 CA机构，CA把自己的信息啥的和服务器的公钥用自己的私钥加密成一个证书发给服务器，服务器把这个证书发给客户端，因为浏览器维护了大部分CA机构的名称地址和公钥信息，就可以验证这个证书是不是合法的对的，如果是真的，就用对应的CA机构的公钥解密这个证书拿到服务器的公钥就可以继续后面的流程了。



S listen

C syn syn-send

S ack syn-revicer

C ack establelished

S 收到后变成establelished

由于挥手是双方都可以主动发送的fin消息,发送了只是告诉你我不给你了,但是你还是可以给我发的,所以得四次.

进程是无法直接操作IO设备的,其必须通过系统调用来内核协助完成IP动作,而内核会为每个IO设备维护一个buffer.

首先以前我们说过linux是没有进程和线程的区别的,对于内核来说确实是这样的,线程和进程在内核里都是task.

进程间通信 套接字是可以在不同机器进程中通信的方式!!!

DASH（Dynamic Adaptive Streaming over HTTP）即自适应流媒体传输.

简单概括来说，就是在服务器端提前存好同一内容的不同码率、不同分辨率的多个分片以及相应的描述文件MPD，客户端在播放时即可以根据自身性能以及网络环境选择最适宜的版本。

Number of shards 分片数量

Number of relicas 每一片的备份数量

es有boolean类型 keyword和普通的text和int等date

分页查询是通过from和size这两个参数加上就是分页了 sort和group啥的也有

还可以范围查询 大于小于查询

推荐只把需要索引的id字段存到es,不需要的字段配合mysql使用.

因为检索时会用到内存,不合理内存会爆炸.

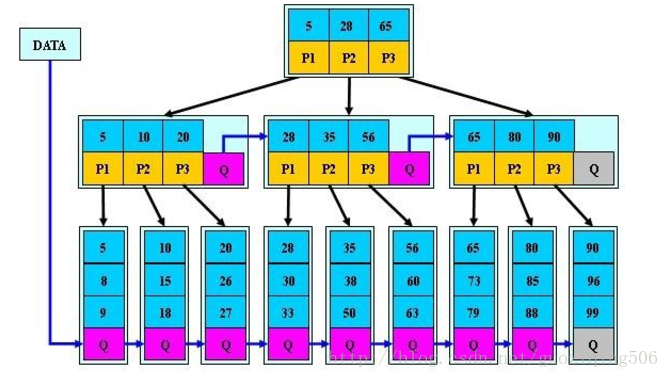
RTMP 协议也要客户端和服务器通过"握手"来建立 RTMP Connection，然后在Connection上传输控制信息。

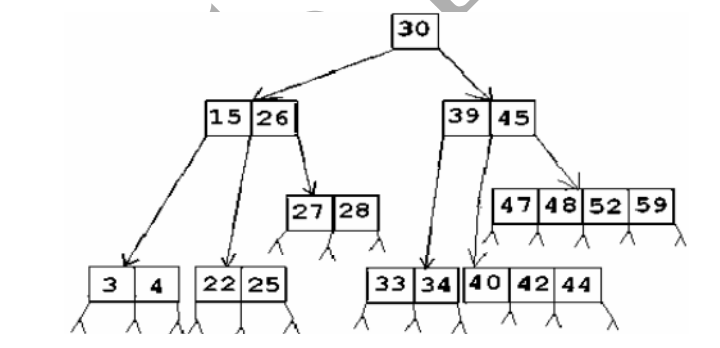
http living stream 可以从名字上看出来是基于http的.

上面的那几个都是基于传输层的,而这个是基于http的,苹果公司开发的基于http的流媒体协议

开闭原则(可以扩展不能改) 迪理米特(多private) 里氏替换 依赖倒置(都依赖接口) 单一职责

Error 运行时异常 普通异常





B+树主要是,中间节点都是索引,子叶节点才是数据,这样我们可以把中间的索引放在一起,这样可以一次性拉出所有索引.

git revert hash值,他这个不会删除节点,而是再最后新加一个节点和你要撤销之前的节点一模一样,这样的话你再push就会把远端也影响了.

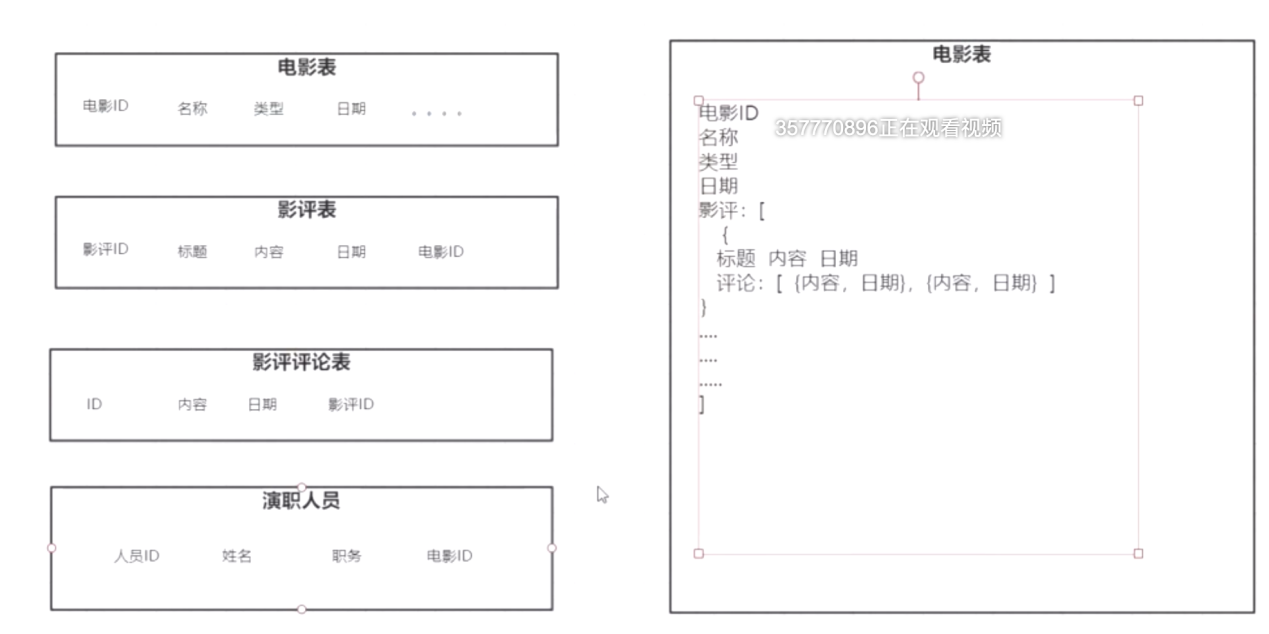
连接池耗尽问题 大key操作时间长 导致查询时间过长,连接不能释放,导致连接池耗尽

定期删除 + 惰性删除

比如你的四个docker要升级，是滚动升级的，其实是先新建了一个docker，然后在把老的干掉一个，又新建一个后，再干掉一个老的，直到全部替换，所以上一次线最少需要一个额外的资源的， 这个叫滚动升级。

原地升级的就是就操作原来的这四个，不去占用其他的资源。

左边使用mysql,维护好几张表的关联信息,右边是mongo,一个表就可以.



数据库->collection->doc->filed

mysql的命令行是sql语法的.

mongodb的命令行是js语法的.

Db.emp.insert(json串)

Db.emp.find({name:”luban”})

Db.emp.createindex也是可以操作索引的.也可以对list加索引,快速的知道list是否包含这个,还可以加联合索引,也有explain

Skip limit sort group啥的 先对某个filed group 然后查个total avg啥的没毛病

我发现, mongodb对于文档中的list的操作确实是很方便,可以操作第一个或最后一个这样,还可以对list中的所有元素统一经常curd,还可以push和pop.

那么一个partition同时同一个group只能有一个consumer去消费,如果你想扩大消费的并发量,你去扩几个partination就行了.如果你consumer少了就是一个consumer拖好几个partination,如果多了,就会有的consumer闲着呆着.

一个group中的所有consumer对一条消息只会消费一次是这样的

也就是说leader follower都是partination级别的,备份因子也是指定follower有多少个,leader是参与读写业务的,follower只是备份.leader挂了会选新的follower

其实这个offset是为一个partition的一个group维护的.

producer可以配置同步发送还是异步发送 broker可以配置是收到消息直接返回给producder还是leader写完就返回,还是follower写完才返回. producer发消息的时候也是有个buffer的,满了就发.

consumer的commit如果是auto的话也会存1000个然后一起commit,这样就会有重复消费,比如存了999个了 挂了, consumer再起来后offset是没有过这个999个的.consumer拉消息也是一批批拉的.

.index是根据offset找到.log中消息的位置的索引

.timeindex是根据时间找到对应时间的消息在.log文件中位置的索引.

消息是放在log文件里的,还会维护里索引文件,通过offset找到消息位置的索引和通过时间找到消息的索引

多个broker中会有一个被选为总控制器,controller,总用比如一个leader挂了组织选一个新的leader,一般都是从isr(已经同步完数据的)里面选.

还有一个东西叫做组协调器 一个group有一个,是一个group有一个,不是一个topic

就是你的一个group哪个消费哪个partination都是需要组协调器分配的,如果你新增了一个新的consumer也是需要rebanlance的,重新分配.

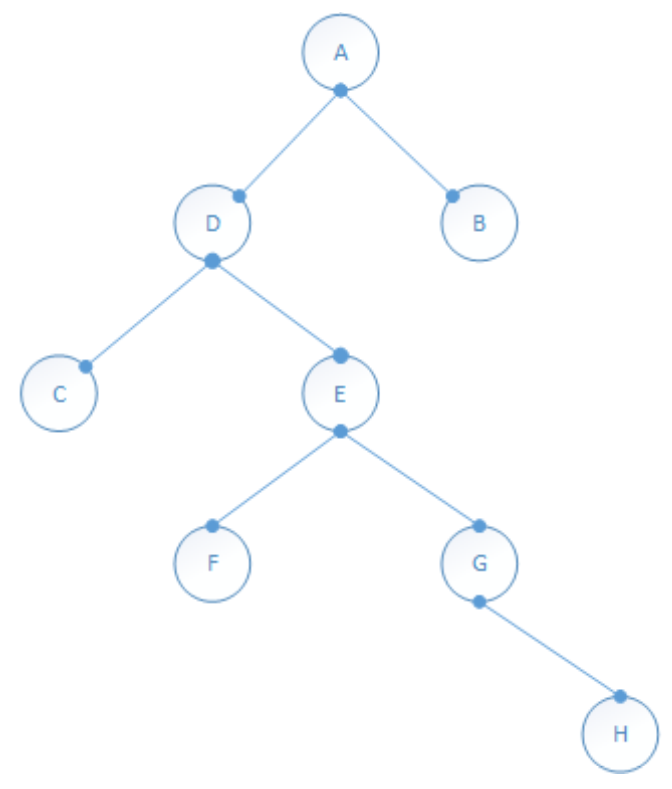
.index是根据offset找到.log中消息的位置的索引

.timeindex是根据时间找到对应时间的消息在.log文件中位置的索引.

hw就是可以被消费到的消息的index.leo就是最大的offset.

hw就是leader和follower都同步完的index.

kafka为什么这么快,主要是因为他的磁盘操作都是顺序读,顺序写,人家本来就是队列么.



先根

ADCEFGHB

中根

CDFEGHAB

后根

CFHGEDBA

undolog保存了事务发生之前的数据的一个版本

如果你俩的主键id爱着,那么你俩的物理存储位置也是挨着的,存储的前后顺序是和id对应的,因为这个是按照id来存的.可以看到顺序摆放,挨着的都挨个罗列在一个页里.

聚集索引的好处就是由于每一个子叶节点就是一个page里的一行,而且在page里的行都是挨着的.这样的好处就是

你通过索引找到的是全部的行数据,而且如果范围查找的话更快.

聚集索引只能有一个.

非聚集索引的意思就是索引顺序和物理顺序没关系的索引,比如你对name进行了索引.其实这个索引是根据name来的,name的B+树的从上到下的大小排列顺序和物理顺序肯定是没有关系的啊.而且这种索引的子叶节点存的时候数据真实的页和行的地址,所以查到索引后还要二次查询一下通过地址找到真正的数据.但是估计二次查询并不会消耗很大的性能.

后面这个二次查询学名叫做回表

表空间又由段 （ segment ） 、区 （ extent ） 、页 ( page ） 行(row) 组成 。

所以解决幻读问题也可以用lock in share model配合nextkeylock,rr并不能解决幻读问题,必须用户lockinsharemode触发出自增锁

计算机里取余操作和按位与是一样的

Select where amount < 3000 这样是有问题的,因为如果amount是null的话,是捞不出来的,所以也侧面反应来,default的重要性!!!!!

1,业务优化 2,中间件优化。3 代码优化

缓存  批处理 预处理 异步

(队列 削峰)。 并行化(比如smartloader)

对于那种需要晚上12:00清零的数据,我们把数据设置在redis里后,我们不用自己再去手动的删key,我们可以直接把超时时间算出来,然后让它到12点的时候自动超时.

还有就是如果你要异步的落mysql的话,不要起go协程去落,可以发事件,然后收到事件的哪个服务慢慢的落mysql

先创建一个serversokcer一个fd,然后对这个fd先bind然后listen,然后你在用epoll\_create创建一个epoll的efd,然后在用epoll\_ctl函数把fd和efd注册一起.

然后你在死循环调用epoll\_wait函数,就可以不停的收到已经有消息或者新注册的客户端socket了.

而且如果是收到了新的链接的socket就把这个新的fd注册到efd里面.其实和nio是一个样子的.

select和poll都只提供了一个函数:select或者poll函数。  
而epoll提供了三个函数，epoll\_create,epoll\_ctl和epoll\_wait，epoll\_create是创建一个epoll句柄；epoll\_ctl是注册要监听的事件类型；epoll\_wait则是等待事件的产生。

epoll除了提供select/poll那种IO事件的水平触发（Level Triggered）外，还提供了边缘触发（Edge Triggered），这就使得用户空间程序有可能缓存IO状态，减少epoll\_wait/epoll\_pwait的调用，提高应用程序效率。

水平触发(level-trggered)

只要文件描述符关联的读内核缓冲区非空，有数据可以读取，就一直发出可读信号进行通知，  
当文件描述符关联的内核写缓冲区不满，有空间可以写入，就一直发出可写信号进行通知  
LT模式支持阻塞和非阻塞两种方式。epoll默认的模式是LT。

边缘触发(edge-triggered)

当文件描述符关联的读内核缓冲区由空转化为非空的时候，则发出可读信号进行通知，  
当文件描述符关联的内核写缓冲区由满转化为不满的时候，则发出可写信号进行通知  
两者的区别在哪里呢？水平触发是只要读缓冲区有数据，就会一直触发可读信号，而边缘触发仅仅在空变为非空的时候通知一次，

LT(level triggered)是缺省的工作方式，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表．

水平触发和边缘触发模式区别

读缓冲区刚开始是空的

读缓冲区写入2KB数据

水平触发和边缘触发模式此时都会发出可读信号

收到信号通知后，读取了1kb的数据，读缓冲区还剩余1KB数据

水平触发会再次进行通知，而边缘触发不会再进行通知  
所以，边缘触发需要一次性的把缓冲区的数据读完为止，也就是一直读，直到读到EGAIN为止，EGAIN说明缓冲区已经空了，因为这一点，边缘触发需要设置文件句柄为非阻塞

## Go

Go 1.5开始， Go的GOMAXPROCS默认值已经设置为 CPU的核数， 这允许我们的Go程序充分使用机器的每一个CPU,最大程度的提高我们程序的并发性能。

但其实对于IO密集型的场景，我们可以把GOMAXPROCS的值超过CPU核数，在笔者维护的某个服务中，将GOMAXPROCS设为CPU核数的2倍，压测结果表明，吞吐能力大概能提升10%

在以前的版本中，它默认为1。

从Go 1.5开始，默认值为核心数。如果您在较新的Go版本中对此不满意，则只需要显式设置即可

GO默认给应用分配的ROCS值为物理机总核数 这样是不行的 因为应该设置程容器的总核数

平衡二叉树和红黑树的查询效率是差不多的,但 是添加和删除操作红黑树的旋转操作比平衡二叉树少.

也就是面向流的协议是无消息保护边界的.

也就是面向消息的通信是有消息保护边界的

假设客户端发送了两个数据包D1 D2给服务端,由于服务端一次读取到字节数是不确定的,(下面也说到了,他可能一次性就读了1m啊)所以可能存在很多情况,

1,服务端分两次读取到了两个独立的数据包,分别的D1和D2,这样就没有粘包和拆包.

2,服务端一次接收到了两个数据包,也就是客户端使用了nagle算法,把D1和D2给粘一起了,这就是封包粘包咯

3,还有就是接收了两次,第一个接收到了D1和D2的一部分,第二次接收到了D2的剩余的部分,那么这样D2就是被拆开了,这样就发送了拆包(**拆包不是把粘起来的包拆开,而是发送的时候一个包的数据分了多次发送**)的操作.

还有就是如果tcp的接收滑窗非常小,而数据包D1和D2都比较大,那么就会发生第5种情况,每个包都需要分多次才能全部接收,这样就是发生了多次的拆包.

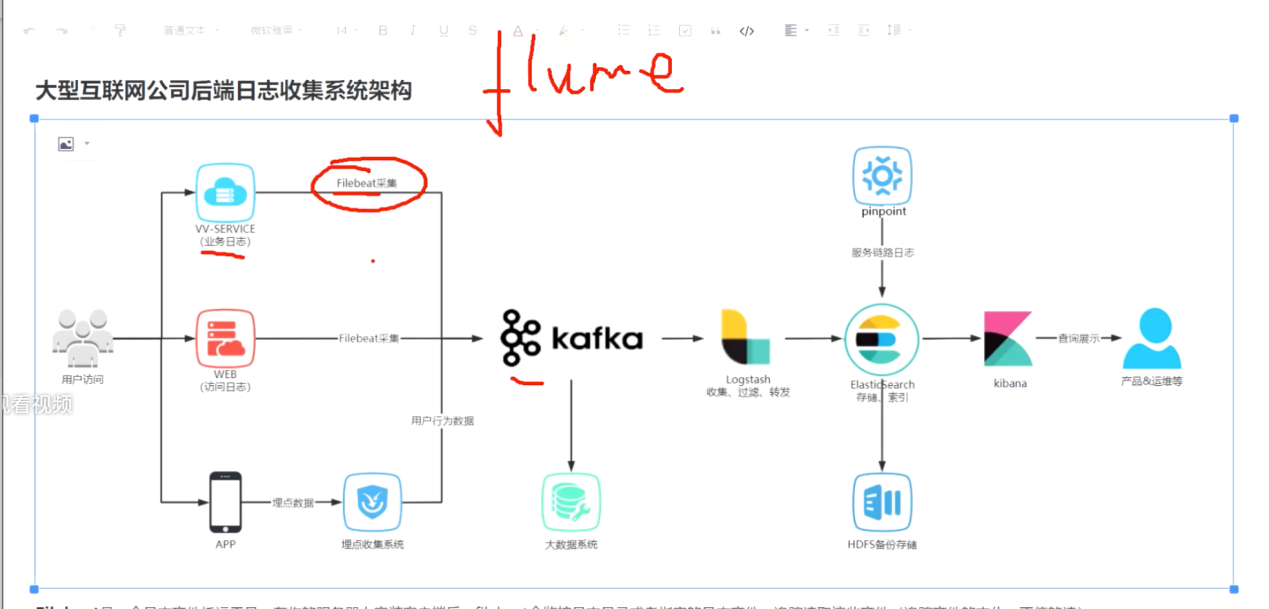
注意wait和await是会把锁交出去的，当前线程进入等待模式不会继续执行,而signal和notify是不会把锁交出的,所以说他只是把其他线程唤醒，但是其他线程还是获取不到锁，但是notify后线程还是会继续往下跑，如果你跑出syn代码块了或者unlock了,被通知的线程才能去获取锁继续跑，

mysql就是一个有状态服务,因为你新扩一个节点,这个节点是不能用的,因为他没有之前的数据,而我们比如exam就是无状态服务,你随时可以扩机器就可以提供服务,所以也就是需要存储数据的就是有状态咯,或者新加入的服务可以自动的去同步数据

api想调用某个服务的rpc, 服务的信息肯定是从consul里面拿到的,本地会存的,估计如果这个服务节点变化了,也是consul推到这个api的,api自己决定去访问哪个服务.不可能每次rpc都去consul去问一下,这样的话所有的rpc流量都会打到consul,它早炸了.

我们定义的idl和pb文件其实也是一种依赖倒置

Git revert hash值的话,是这样的,你reset hash值的话,你这次提交是没有了,但是呢,如果你这次提交已经推到远端了,你把本地分支的这次提交删了,远端还是没删的,如果你想把这次的撤销影响的代码后续也影响到远端的话,就要用git revert hash值,他这个不会删除节点,而是再最后新加一个节点和你要撤销之前的节点一模一样,这样的话你再push就会把远端也影响了.



现在日志系统kafka和应用程序之间一般还会有一层flume或者filebeat,用来缓冲.

而且还可以做一些配置和代理,类似于头条的databus.

分布式调用链系统 pinpoint zinkin

也是相当于起一个分布式调用链服务收集信息,然后存储,然后ui展示,也是有自己的服务的.

logstash主要是一个管道的功能,可以从输入(比如日志文件)读出数据,然后过滤数据,然后传输数据到输出端(比如mysql es等数据源)

[http://godoc.org](http://godoc.org/" \t "/Users/yangwenshuo/Documents\\x/_blank)

搜索开源包

type database map[string]dollars

func (db database) ServeHTTP(w http.ResponseWriter, req \*http.Request) {

for item, price := range db {

fmt.Fprintf(w, "%s: %s\n", item, price)

}

}

多用这种方式...................... 把固定类型的map和slince type 成一个类型,然后这个类型就可以挂函数了.

type I6 uint64  
  
func (i I6) Print() {  
 fmt.Println(i)  
}  
  
func main() {  
 var i int32 = 100  
 i6 := I6(i)  
 i6.Print()  
}

如果一个变量被强转成另一个类型,而另一个类型是有自己的方法的,那么这个变量也就有了这个类型的这个方法了!!!!!!

func main() {

db := database{"shoes": 50, "socks": 5}

mux := http.NewServeMux()

mux.Handle("/list", http.HandlerFunc(db.list)) 这是一个强转函数的操作

mux.Handle("/price", http.HandlerFunc(db.price))

log.Fatal(http.ListenAndServe("localhost:8000", mux))

}

type database map[string]dollars

func (db database) list(w http.ResponseWriter, req \*http.Request) {

for item, price := range db {

fmt.Fprintf(w, "%s: %s\n", item, price)

}

}

func (db database) price(w http.ResponseWriter, req \*http.Request) {

item := req.URL.Query().Get("item")

price, ok := db[item]

if !ok {

w.WriteHeader(http.StatusNotFound) // 404

fmt.Fprintf(w, "no such item: %q\n", item)

return

}

fmt.Fprintf(w, "%s\n", price)

}

package http

type HandlerFunc func(w ResponseWriter, r \*Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r \*Request) {

f(w, r)

}

首先我们可以把函数定义成一个类型,这个类型呢也可以有自己的方法,上面的这个方法就是执行函数自己.

还有就是函数类型的强转只认参数列表和返回值,和别的都没关系,你是一个方法,只要和我的函数的参数列表和返回值相同,照样可以强转.

上面这个是个注册函数的操作,看到了么,可以直接就给强转了.

还有就是如果你的参数想传一个函数的话,那么你可以先把这个函数type出来成一个类型,这个操作起来比较方便.

func wrapMyBankeList(c \*gin.Context) {

go语言里首先函数也是个类型,这个类型在随意不过了,只要参数和返回值一样就可以替换,你看gin的handle函数,就是十分简单的参数和返回值啊.

Uint 如果我们搞一个bitmap的话, 数组得用uint的.

<https://github.com/developer-learning/learning-golang>

golang的这些东西都是相对于package来说的,比如go get是下载一个package

Go build是编译一个package,你可以去任何一个package下编译,只不过如果你的这个包是main包的话会给你生成一个可执行文件.

为了满足这些需求，Go语言的构建工具对包含internal名字的路径段的包导入路径做了特殊处理。这种包叫internal包，一个internal包只能被和internal目录有同一个父目录的包所导入。例如，net/http/internal/chunked内部包只能被net/http/httputil或net/http包导入，但是不能被net/url包导入。不过net/url包却可以导入net/http/httputil包。

所有以\_test.go为后缀名的源文件在执行go build时不会被构建成包的一部分，它们是go test测试的一部分。

组合模式就是什么文件夹文件的哪个,处理文件夹和文件都一个函数,把文件和文件夹拉平了.



这种test结尾的文件不能被依赖

[https://goproxy.io](https://goproxy.io/)

goproxy

一个project下面可以多有多main函数,这些main函数不在同一个direction下面就可以.

但是这些direction里面的package都要叫main,main函数的package必须叫main,但是direction可以叫别的.

我们平时遇到的panci其实都是下面调用了panic给扔上来的,只不过都是go源码调用的平时我们看不到,但是如果你不调用panic,就不会有panic了.

func swap(arr []int32, i, j int) {  
 arr[i], arr[j] = arr[j], arr[i]  
}

这个交换这样写才好.

*//注意这两个函数要是同步得,且同步对象是同一个才能玩notify和wait***public synchronized void** method1() **throws** InterruptedException {  
   
 **if**(i==1) {  
   
   
 *//因为这是一个同步方法，锁对象是this，所以是this.wait();* **this**.wait();  
 }**else** {  
 **for**(**int** i = 0;i<20;i++) {  
 System.out.println(**"A打印啦啦啦啦"**);  
 }  
 i=1;  
 **this**.notify();  
   
 *//通知后马上就把自己给wait了,等待别人得通知.* **this**.wait();  
   
 **for**(**int** i = 0;i<20;i++) {  
 System.out.println(**"C打印啦啦啦啦"**);  
 }  
 }  
   
}

单例模式注意要把里对象的构造方法私有.

**notify之后马上就wati了,wait就会把锁释放出去了**

我发现了,很多存数据的东西比如bigtable,rocketmq,kafka和hot这种都是习惯把数据和索引分开,数据放到一个长数组里,比如bigtable的数组,kafka的rocketmq的日志,和hot的数据,然后索引放另一个地方.

对于变量也是一样的,如果变量名是大写,那么别的包也可以通过包名访问这个变量,如果变量名是小写,就不行了.

make([]int, 5) 指定了长度而不是容量

type A interface {

Print()}

type B struct {}

func (B)Print() {}

var \_ A = new(B)

判断一个结构体是否实现了一个接口

import \_ "net/http/pprof"

这一种非常干净的导入包的方式，由于在当前文件中，被导入的包是匿名的，因此你无法访问包内的任何符号。（如果导入的包不是匿名的，而在程序中又没有使用到其内部的符号，那么编译器将报错。

**func** (e **\***emptyCtx) **String**() **string** {

**switch** e {

**case** background:

**return** "context.Background"

**case** todo:

**return** "context.TODO"

}

**return** "unknown empty Context"}

**var** (

background = new(emptyCtx)

todo = new(emptyCtx))

switch也可以直接比较指针

每个包一般都定义了一个不同的名字空间用于它内部的每个标识符的访问。每个名字空间关联到一个特定的包，让我们给类型、函数等选择简短明了的名字，这样可以在使用它们的时候减少和其它部分名字的冲突。当我们修改了一个源文件，我们必须重新编译该源文件对应的包和所有依赖该包的其他包。

func Fprint(w io.Writer, a ...interface{}) (n int, err error) {  
 p := newPrinter()  
 p.doPrint(a)  
 n, err = w.Write(p.buf)  
 p.free()  
 return  
}

fmt包的这个函数可以帮助你写流

为了禁止编译器对未使用导入包的错误报告，我们可以用空白标识符来引用一个被导入包中的符号。同样的，将未使用的变量fd赋值给一个空白标识符也可以禁止编译错误。这个版本的程序就可以编译通过了。

import (

"fmt"

"io"

"log"

"os")

var \_ = fmt.Printf // For debugging; delete when done.

var \_ io.Reader // For debugging; delete when done.

func main() {

fd, err := os.Open("test.go")

if err != nil {

log.Fatal(err)

}

// TODO: use fd.

\_ = fd

}

type Memo struct {

f Func

mu sync.Mutex // guards cache

cache map[string]result

}

得习惯于这种定义,一个结构体,有主要的数据属性,和锁对象和一个传进来的函数

这个函数是外面定义的在什么情况下执行的什么可定制的操作.

type arr struct {  
 i int  
}  
func (a \*arr) print() {  
 fmt.Println(a.i)  
}  
func main() {  
 a := &arr{i:10}  
 a.print()  
}

可以看到,这个函数虽然是挂在了指针上,但是呢,结构体也是可以直接调用的.

该协程需要从in通道读数据，还需要定时打印已经处理的数量，有2件事要做，所有不能使用for-range，需要使用for-select，

var send\_only chan<- int // channel can only receive data

var recv\_only <-chan int // channel can only send data

只有函数的签名但是却没有函数体是什么情况

一是函数签名使用Go,然后通过该包中的汇编文件来实现它

比如，在标准库 sync/atomic 包中的函数基本只有函数签名。比如：atomic.StoreInt32

// StoreInt32 atomically stores val into \*addr.func StoreInt32(addr \*int32, val int32)

它的函数实现在哪呢？其实只要稍微留意一下发现该目录下有一个文件：asm.s，它提供了具体的实现，即通过汇编来实现：

二是通过//go:linkname指令来实现

比如，在标准库 time 包中的 Sleep 函数：

// Sleep pauses the current goroutine for at least the duration d.// A negative or zero duration causes Sleep to return immediately.func Sleep(d Duration)

它的实现在哪里呢？在 time 包中并没有找到相应的汇编文件。

按照 Go 源码的风格，这时候一般需要去 runtime 包中找。我们会找到 time.go，其中有一个函数：

// timeSleep puts the current goroutine to sleep for at least ns nanoseconds.//go:linkname timeSleep time.Sleep

func timeSleep(ns int64) {

...

}

如果我们用bitmap来表示数据的话, 对不同的bitmap求交集 求合集什么的会非常的方便的啊.

主函数返回时，所有的goroutine都会被直接打断，程序退出

结构体的属性的大写小写,小写包内可以用,不是出了这个包就不能直接用了,是以包作为边界的.

type Wheel struct {

X, Y, Radius, Spokes int

}

结构体相同的类型可以写在一起定义

seen := make(map[string]struct{}) // set of strings// ...if \_, ok := seen[s]; !ok {

seen[s] = struct{}{}

// ...first time seeing s...

}

Struct{}是这样用的

defer语句的执行顺序和写的顺序相反.

var prereqs = map[string][]string{  
 "algorithms": {"data structures"},  
 "calculus": {"linear algebra"},  
 "compilers": {  
 "data structures",  
 "formal languages",  
 "computer organization",  
 },  
 "data structures": {"discrete math"},  
 "databases": {"data structures"},  
 "discrete math": {"intro to programming"},  
 "formal languages": {"discrete math"},  
 "networks": {"operating systems"},  
 "operating systems": {"data structures", "computer organization"},  
 "programming languages": {"data structures", "computer organization"},  
}

比如我们要设计一个学科的map,也就是你想学习某一个学科必须要先学会他之前依赖的学科,设计这个的时候我们就用上面的那个map,就非常好了.学名叫做拓扑排序

,这是一个图.

Golang也是有log包的,无非也就是创建一个全局变量,这个变量指定好日志的文件目录啊,打印格式啊,然后logger.warn就可以了.

大部分编程语言使用固定大小的函数调用栈，常见的大小从64KB到2MB不等。固定大小栈会限制递归的深度，当你用递归处理大量数据时，需要避免栈溢出；除此之外，还会导致安全性问题。与此相反，Go语言使用可变栈，栈的大小按需增加（初始时很小）。这使得我们使用递归时不必考虑溢出和安全问题

在Go中，函数运行失败时会返回错误信息，这些错误信息被认为是一种预期的值而非异常（exception），这使得Go有别于那些将函数运行失败看作是异常的语言。虽然Go有各种异常机制，但这些机制仅被使用在处理那些未被预料到的错误，即bug，而不是那些在健壮程序中应该被避免的程序错误。对于Go的异常机制我们将在5.9介绍。

Go这样设计的原因是由于对于某个应该在控制流程中处理的错误而言，将这个错误以异常的形式抛出会混乱对错误的描述，这通常会导致一些糟糕的后果。当某个程序错误被当作异常处理后，这个错误会将堆栈跟踪信息返回给终端用户，这些信息复杂且无用，无法帮助定位错误。

正因此，Go使用控制流机制（如if和return）处理错误，这使得编码人员能更多的关注错误处理。

我们应该在每次函数调用后，都养成考虑错误处理的习惯，当你决定忽略某个错误时，你应该清晰地写下你的意图

func main() {

fmt.Println(test1())

}

func test1() (int, string) {

return 1, "werer"

}

这样可以直接打印出来两个返回值.

并不一定是所有的函数都挂在logic上, 对于examination说 可以抽象一个exam的结构, 很多函数都可以挂在exam结构上, 这个exam结构上挂了转成model放入函数 也挂了转成thrif的函数t,这都是这个exma的结构体的transfer函数, 面向对象变成咯, 函数都挂在结构上, 而且对曰这个结构体的update db什么的也可以弄成这个结构踢的函数

userquestion也可以设置成函数,他们的updatedb的操作都可以弄成挂在他身上的函数

所有的getfrondb和rpc和redis其实都可以用loader来写

go install 和go build都要到main.go目录下执行

func deDuplicatePointListByName(pointList []\*examination.ExamPoint) []\*examination.ExamPoint { existPointNameMap := make(map[string]bool) resultPointList := make([]\*examination.ExamPoint, 0, len(pointList))

for \_, point := range pointList {

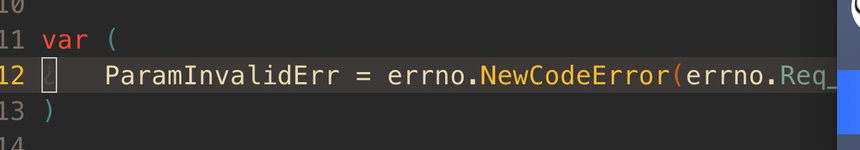
if \_, ok := existPointNameMap[point.GetPointName()]; !ok { existPointNameMap[point.GetPointName()] = true resultPointList = append(resultPointList, point)

}

} return resultPointList

}

一个package下同一个函数名字只能有一个,一个包之内的函数是可以直接使用的.



我们可以提前创建好一些直接可以用的err的变量,到时候直接用就可以了.

多个扫表实例 不要用分布式锁锁资源,而是要锁协程,多个实例去竞争一个分布式锁,获取到锁的再去扫表,这样就没有什么问题了.

type Middleware func(EndPoint) EndPoint

快照id太众要了,没有快照id, paper变了, 你都找不到过去的数据的.

\_ = os.Setenv("TZ", "Asia/Shanghai") 设置时区

github.com/allegro/bigcache/

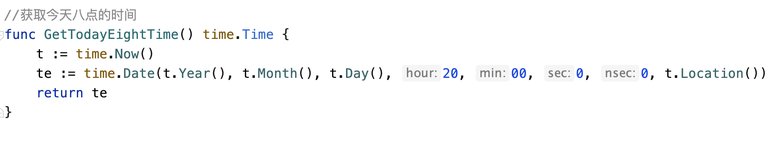
允许对值为 nil 的 slice 添加元素，但对值为 nil 的 map 添加元素则会造成运行时 panic

go携程重启有损

最后写可见性的,删的话先删有可见性的

对于批改生成图片声音这种,和考试判卷这种,把原材料和结果要分开,不要放一起.

em哪些结构也可以放在model,model里面可以放任何的数据结构,不仅仅是db的model.



go get 命令可以借助代码管理工具通过远程拉取或更新代码包及其依赖包，并自动完成编译和安装。整个过程就像安装一个 App 一样简单。

所以说go get是下载app用的,会自动安装的.

go get是安装功能 还可以在gomod里面管理依赖

gosum的校验和1, 可以校验的本地缓存的go包是不是完整的,2 可以校验版本是否匹配,比如你依赖commonv1.1.2, 他的这个tag的sum是xxxa,你本地是的xxxb,那对不上了就不行了.

检索表和实体表区分开

有的表和业务相关的就是索引表,有的呢就是明细的实体表

不要老是if xxx{

return

}

而是多用if else 这样逻辑清楚

type SubAnsInPaper []\*OneSubQuestion

这样可以type一下

go里的string是被设计成不可写的, 如果比把string强转成[]byte的话,就不能对这个[]byte做写操作 不然就会报错的啊

float转int直接括号强转

if res, ok := data.(\*BankeData); ok {

   bankeMap := make(map[int64]\*cl

强转成这个类型,看看能不能成

go mod graph | grep -i xxxx

*为服务直接不允许循环调用,不然会有调用风暴,这样就要求了为服务的强自治,但是确实有的一个服务不能完成任务,所以需要聚合服务,对上面有一个聚合服务,服务的自制 trade就是聚合服务*

api - sercei -dao是这样的。层宇层之间是互相不接的,所操作的数据都是互相抽象的,对于servcei层来说,只认订单这个对象,并不认你的订单下面有多少表

菱形分层

服务的边界(意思就是一个服务的功能,一个服务不能啥都干啊)

rand.Seed(time.Now().UnixNano())

 r := rand.Float64() \* float64(weightSum)

更新字串的正确姿势：将 string 转为 rune slice（此时 1 个 rune 可能占多个 byte），直接更新 rune 中的字符 func main() { x := "text" xRunes := []rune(x) xRunes[0] = '我' x = string(xRunes) fmt.Println(x) // 我ext }

意思是字符串可以转成[]byte, 也可以转成[]rune,但是注意的有的字符,是会占用两个rune的.法文单词 café 中的 é 就占了两个字符

case ' ', '\t': return true }

Case的多场景判断

以小写字母开头的字段成员是无法被外部直接访问的，所以 struct 在进行 json、xml、gob 等格式的 encode 操作时，这些私有字段会被忽略，导出时得到零值

// 关闭 resp.Body 的正确姿势 if resp != nil { defer resp.Body.Close() }

可以使用相等运算符 == 来比较结构体变量，前提是两个结构体的成员都是可比较的类型：

Ioutil.readfile

golang结构体每个元素赋值的时候都需要后面加, json拼串的时候最后一个才不需要加,

var lsls = []string{"1", "2", "3"} for \_, i := range lsls { go func() { fmt.Println(i) }() }

这玩意打印的是333

我们的一些过 程结构体的filed的命名, 最好是用小写, 因为java的类的变量一般也都是private.

go test -timeout 99999s

落数据的时候落id,pack的时候通过id窜数据好处太多了 数据变化的话 改下窜数据就行了 丝毫不需要刷老数据

Strconv errors fmt time encoding/json

"[github.com/json-iterator/go](http://github.com/json-iterator/go)"

golang空串往结构体上json解析会报错!!!!!

for遍历nil的list和map是没有问题的

json串往list和map解析的时候必须要加上&

数据耦合而不是函数的参数耦合

jsoniter 只要是对nil 进行marsharl。就会得到一个内容是null的字符串!!!!!

如果一个结构体里面还有一个结构体指针,一个字符串往这个结构体json转的时候里面的结构体确没有赋值,那么里面的结构体就是一个空指针.

一个指针数组是可以append nil。而且nil还会占一个长度

keshiIdMap := make(map[int64]struct{})

keshiIdMap[item.GetKeshiId()] = struct{}{}

只要key做去重用struct{}

time.Now().AddDate(0, -90, 0)

分别是操作年月日的

if !ok {

videoMap[video.QuestionId] = make([]\*model.QuestionVideo,0)

}

videoMap[video.QuestionId]  = append(videoMap[video.QuestionId] , video)



各种新定义的数据要放在最上面,因为如果你定义在最下面的话很可能跑不到你要的地方就放回了啊

三种模式: on/auto/off，默认为 auto: $GOPATH下的项目不开启，非$GOPATH下的开启

type WaitGroupWrapper struct {

 sync.WaitGroup

}

func (w \*WaitGroupWrapper) Wrap(cb func()) {

 w.Add(1)

 go func() {

  cb()

  w.Done()

 }()

}

这样add一个done一个不会出现问题

go语言map遍历时删除是安全的， 且可以完全删除



sort.Slice(results, func(i, j int) bool {

        return results[i].Value < results[j].Value

})

其实这个sort.Slice也是可以自定义比较规则的

有个once包的

结构体的方法如果是小写的话出了这个包就用不了了,即使这个方法是结构体的.

method只有一个,通过method的不同参数对应到多个logic

头条的context从前面一直传到后面,请求里的压测http头。gorm都能知道

不应该把方法都弄到和包关联,而是要和结构体关联

如果想知道某个channel容量或者队列的大小，可以调用cap("channel")方法，而如果想知道当前channel中元素的个数可以调用len("channel")函数。cap、make、len都是go自带的内建函数。

 err = dal.InsertPaperRecord(tx, recordModel)

 if err != nil {

  logs.CtxWarn(ctx, "insert paper record err %v", err)

  tx.Rollback()

  return err

 }

 err = dal.UpdatePaperQuestion(ctx, tx, \*item.GetQuestion().QuestionId, eaItemContent)

 if err != nil {

  tx.Rollback()

  logs.CtxWarn(ctx, "update paper question err %v", err)

  return err

 }

}

这种多个rollabck的,弄成一个函数,rollback在外面不久可以了



字符串转时间

不能循环import method imoprt logic。logic import dal。dal再improt method

a = append(a[:i], a[i+1:]...) 删除一个切片的元素

各种计算不是应该在每次请求的时候,  而是提前就弄好,  在另一个地方去计算.

time.Unix(timestamp, 0)

func DivInt32(a,b int32) float64{

 if b == 0 {

  return 0

 }

 return float64(a) / float64(b)

} 这种东西抽成函数多好啊

V2接口主要是可以兼容老版本的app

一个logic里面的一个数组对象的,没一个对象都是一个logic

slice并发写入不会报错。map并发写入会报错

ticker := time.NewTicker(common.INSTANCE\_LOOP\_TIME \* time.Second)

 for {

  select {

  case <-ticker.C:

   logic.HandleExaminationInstanceCreate(ctx)

  }

 }

起一个协程做事,一定要写好一个recover,因为这个panic框架抓不到

go get -u的时候是一个project的粒度,但是import的粒度确实最小的那个包

float类型不要作为map的key

data := map[string]interface{}{

  "app\_id":    common.MatrixAppId,

  "app\_key":   common.MatrixAppKey,

  "all\_point": true,

  "tree\_ids":  treeIds,

 } map定义的时候就初始化了,并且是一个interface的map

sort.Sort(xxx) 这个是可以自定义排序方式的排序

一个接口就返回这个接口需要的东西,不要搞那么多冗余.

还有就是不要让客户端搞记住前面一个接口返回内容在下一个接口使用的操作.

gomod下载的那些版本文件都在gopath里面的pkg里面.

我们go build就是编译生成可执行文件,go install就是把生成的可执行文件移动到gopath的bin目录里面.

go generate命令是在Go语言 1.4 版本里面新添加的一个命令，当运行该命令时，它将扫描与当前包相关的源代码文件，找出所有包含//go:generate的特殊注释，提取并执行该特殊注释后面的命令。比如生成那个msgpb

还有就是go语言自带的go fmt代码格式化命令

Go test 如果前后代码不变的话是不会缓存结果的,加上这个命令才不会有缓存 -count=1

func main() {

 var err error

 defer func() {

  if err := recover(); err != nil {

  }

  if err != nil {

   fmt.Println("err bu shi nil")

  }

 }()

中间里面的那个err的作用范围只有在recover的if里面,也就是说无论他是什么,都不会对外面的err有任何影响,因为他是在if里面新建的!!!!!!!!!!,所以他的作用范围就是if里

rep := regexp.MustCompile("(L|l)evel(\\d+)")

 fmt.Println(rep.FindString("dfdflevel123英语课"))

从一个字符串中获取符合规则的字符串



结构体的元素是可以直接取地址的,但是强转的就没有办法直接取地址了

import 下划线（如：import \_ hello/imp）的作用：当导入一个包时，该包下的文件里所有init()函数都会被执行，然而，有些时候我们并不需要把整个包都导入进来，仅仅是是希望它执行init()函数而已。这个时候就可以使用 import \_ 引用该包。即使用【import \_ 包路径】只是引用该包，仅仅是为了调用init()函数，所以无法通过包名来调用包中的其他函数。

Import 前面是个. 这样的意思就是在调用这个包下面的函数不用加包名字了

如果import前面是个字符串,就是为了避免相同的包名字冲突把这个包给重命名了.

一个包或者一个文件里可以有多个init文件,执行顺序是从上到下执行,执行完所有init后才会执行正经内容,包括main函数.

这个是比较快的json

"[github.com/json-iterator/go](http://github.com/json-iterator/go)"

var json\_iterator = jsoniter.ConfigCompatibleWithStandardLibrary

Time.Now().Add(time.Durtation(60) \* time.Second)

这里要说的时候,要操作的时间的时候,要这样把数字强转成time.Duration()类型.

api的路径后面如果少一个/的话就会触发307帮你重定向到正确的地方.

如果一个时间的unix的时间是0,那么他的time对象是0001年0001月01日,并不是0时,但是这个时间由于在1970年之前,那么转出来的时间戳就是个负数的啊.

字符串转时间

t1, err := time.ParseInLocation(“2006-01-02 15:04:05”, 时间的串, time.Local )

一个tasklist append一个东西后,如果触发了扩容,那么地址就变了啊

如果一个普通的结构体你不打上json的tag,那么序列化出来的字段名就是结构体元素的名字.

type Product struct {

    Name      string  `json:"name"`

    ProductID int64   `json:"-"` // 表示不进行序列化

    Number    int     `json:"number"`

    Price     float64 `json:"price"`

IsOnSale  bool    `json:"is\_on\_sale,string"`

指定类型为string,这样这个字段打印出来就是"is\_on\_sale":"false"

而不是这样"IsOnSale":true

ProductID int64   `json:"product\_id,omitempty"`

这样的话就是如果这个字段为空就不会序列化了.

 ProductID int64   `json:"product\_id,string"`

Number    int     `json:"number,string"`

这样的话int的元素就会按照字符串打印,

"product\_id":"10","number":"10000"

strings repalce必要用1.2

import 下划线（如：import \_ hello/imp）的作用：当导入一个包时，该包下的文件里所有init()函数都会被执行，然而，有些时候我们并不需要把整个包都导入进来，仅仅是是希望它执行init()函数而已。这个时候就可以使用 import \_ 引用该包。即使用【import \_ 包路径】只是引用该包，仅仅是为了调用init()函数，所以无法通过包名来调用包中的其他函数。

go语言的字符串是可以直接比较大小的.

match, err := regexp.MatchString("11", "11")

如果你想自定义一个error的话,只需要定义一个结构体实现Error() string这个函数就可以了,结构体的元素随便是啥,一般都是code和msg.因为error本身就是个接口啊.

对于把接口对象转成真正的实现类对象,这样操作就可以了.

If res,ok:= err.(\*xxxError);ok{}

有的时候你用go mod拉一个工程,拉不下来看提示可能是你没有这个工程的权限,但是你确依赖了他. 平时我们依赖的包可以拉下来说明人家是对外开放的.

New(struct)出来是一个指针

给了Maintainer,scm你就可以配置了,这也会影响到.

go的结构体如果你要弄成json的话,结构体名字要大写,后面还要补上json的注释,不然是识别不出来的哦.

json还是比较只能的,多一个少一个的,顺序什么的都无所谓的,它好像就是用key当作map去捞的.

结构体也是可以用%v打印的,万物皆可用%v打印.

json是结构 还是json是字符串是不一样的.

Json unmarl的时候数组和map都必须加上地址!!!!!!!!!!!!!

数组元素的内容是结构体和是指针都是可以的.

如果实现了结构体中的String方法，在使用fmt打印时候会调用该方法，  类似于java中的tostring

结构体会转化为JSON对象，并且只有结构体里边以大写字母开头的可被导出的字段才会

被转化输出，而这些可导出的字段会作为JSON对象的字符串索引,也就是结构体想要可以序列化成json的话,元素必须的是大写的那部分才会被转化.

如果JSON中的字段在Go目标类型中不存在，json.Unmarshal()函数在解码过程中会丢弃 该字段,而不是报错,所以json的转换是根据key去填的,多几个少几个,都不会影响其他字段的解码与编码.

3 转化一个map类型的数据结构时，该数据的类型必须是 map[string]T(T可以是

encoding/json 包支持的任意数据类型

type Transport struct {

// Proxy指定用于针对特定请求返回代理的函数。

// 如果该函数返回一个非空的错误，请求将终止并返回该错误。

// 如果Proxy为空或者返回一个空的URL指针，将不使用代理

Proxy func(\*Request) (\*url.URL, error)

// Dial指定用于创建TCP连接的dail()函数。

// 如果Dial为空，将默认使用net.Dial()函数

Dial func(net, addr string) (c net.Conn, err error)

看到没有,上面的两个元素,就是函数类型,go直接有就有一个函数类型的哦!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

go func() {

defer close(exit)

go func() {

fmt.Println("b")

}()

}()

Go 运行一个不要定义的func是这样子操作的

go中的方法和传统的类的方法不太一样，方法和类并非组织在一起，传统的oop方法和类放在一个文件里面，而go语言只要在同一个包里就可，可分散在不同文件里。go的理念就是数据和实现分离，

go的结构体和属于这个结构体的方法可以在不同的文件里，但是要在同一个包里

把一个interface{}类型的东西转成他真正的类型。

if res, ok := data.TfPlanDH.Data.(\*data.TfPlanData); ok {

if err = conn.Table("tf\_plan").Where("plan\_id =?", tfPlan.PlanId).Updates(tfPlan).Error; err != nil {

  logs.CtxWarn(ctx, "UpdateTfPlan Updates error: %s", err)

  return err

 }这种更新的时候传进来的结构体需要有id的

这种就会像java那样，你的结构体的元素不是空就会去更新，是null的就会保持原来的。

func test(arr \*[]string) (s string){

这样的话，看到了么，返回值列表既定义了一个类型，顺便也创建了这个返回的变量，代码里对s进行复制操作就可以了，但是这种返回方式必须要用小括号扩起来

go func(){fmt.Println("Hello inline")}()

        go printHello()

但是呢，go运行一个用户线程的时候，函数名后面必须是个（）

所以即使第一行的是一个匿名函数，后面也要加一个（）

:= 结构不能使用在函数外，函数外的每个语法块都必须以关键字开始

var (

    a int

    b string

    c []float32

    d func() bool

    e struct {

        x int

    }

)          批量定义变量

go语言很多的那流程操作都是在init里面，如果不看init单撸代码的话可能看不到

channel是一个定义比较简单的携程之间的消息队列啊，因为channel是多个的。

看到了么，函数指针的执行。

        if (eventActFun)

        {

            eventActFun();

声明时的\*eventActFun的\*只是声明作用，下面用的时候还是要用eventActFun

返回值（\*函数名）（参数列表）

是在函数名前面顶了一个\*，这个就是函数指针。

var ptr \*\*int; 指针的指针

go的携程是用户线程。

numbers = append(numbers, 2,3,4) 还可以追加切片和数组。

slice1 := make([]type, len) len是切片的初始i长度，指定0就可以，我们可以往上在append。

/\* 拷贝 numbers 的内容到 numbers1 \*/

copy(numbers1,numbers) 切片复制

range就是foreach呗

**那种既不是大写也没有指定结构体的方法就是包内可以调用，那种大写的是包名调用（这种是类似与静态）。 而实现接口的方法或者结构体自己的方法就是结构体指针调用。**

**包名调用的还有就是那种文件var定义的全局变量，还有就是包下的结构体信息，其他包想创建这个包定义的结构体的时候。**

**还有就是如果一个结构体的元素名字不是大写的,但是结构体名字是大写的,那么出了这个包后其他包即使创建了这个结构体(因为结构体的名字是大写的),但是也是无法访问这个结构体的元素的,因为结构体的名字是小写的.**



go的依赖会先去vendor里找，然后在去gopath里面找。

Go的数组会如果长度超了会扩容和java的listy差不多，关键的是lsit的扩容是list对象里面的buff变成新的，list指针不会变，而go的扩容后数组的地址都变了！！！！

注意，init的顺序，上面的还没跑到他的init函数，你就去调用他。

Fmt.printf(“%p”)就是按照指针打印

经过验证,那种string init的啥遍历的时候指针是同一个.

经过验证,那种结构体数组在遍历的时候指针也是同一个,对于结构体指针数组在遍历的时候指针就不是同一个了.

还有就是在遍历数组的数组的时候,其中的一个item就是一个指针了,不要在加&这东西了.

Fmt.Printf(“%T”)是打印类型,如果你还要加个&的话,就变成数组的指针了,以后就把数组当作指针来看了.

对于那种创建一个数组并塞一个元素并赋值的代码,不要那样先创建一个切片,在给切片append然后再赋值,而是要xxx = []\*model.xxx{ aaa } 这样就可以了

结构体切片的根据结构体的某一个元素排序代码

sort.Slice(results, func(i, j int) bool {

        return results[i].Value < results[j].Value

})

这里就是根据Value元素的大小排序results这个数组.

啥是数组 [10]int 只要是中括号里面有数字的就是数组,没数字的就是切片,不看别的.

V2 := v1[0,6]

V3 := v1[2,5]  
注意这样的话,v2和v3其实都是走的v1底层的那个数组,你改了v2是会影响到v3的.

但是go语言有个copy内置函数.

我们复制切片的时候就可以用copy来操作.

V2 := make([]int, 6)

Copy(v2, v1[0:6])

这样就是复制内容到了v2.

定时任务也要go 起来啊。这种扫表的,不然你里面是个死循环下面的执行不了,所以死循环的定时任务要用一个新的协程.

抽象能力.

对于从数据库的查的数据,一定要把两个返回值都判断下,不要只判断error,因为err不为null,但是数据可能是有问题的,数据库嘛.

for {

    DoSomething()

    if done {

        break

    }

}

Do while

h.Res.ErrNo, h.Res.ErrTips = -1, err.Error()

一次行赋值两个

数据耦合,而不是参数耦合,不是程序耦合,有没有发现经常一个参数层层的传下去...,如果通过数据耦合就可以避免这个问题.

数据结构的定义要复杂而不是算法复杂.

各种计算不应该是在每次请求的时候,而是提前就算好,请求的时候直接给数据.

一定要注意接口的幂等性, 幂等性最好的方法还是加唯一索引.

把db层只做简单的更新,插入等,事务啊,我们放在logic层.组合的db操作我们也放在上面的层.

有些东西是基础服务,比如班课啊,很多东西都是依赖着班课,基础服务的概念要知道.

对于什么东西我们要冗余呢,对于那种不方便循环rpc的我么就冗余在表里避免查询时间太长.

函数内容一定不能过长.

函数不要全关联到包,而是要关联到logic对象上面.

不要在代码里面的所有的结构都用thrift的或者是pb的,而是要多用自己定义的数据结构,也就是定义在logic里面的数据结构.

logic一般都会有一个newlogic的代码,这个newlogic看起来是没什么用,但是类似于上次的那种选择题的数组默认有abcd的这种,这个时候newlogic就可以初始化一些东西了.

尽量把一些方法都关联在logic结构体上,这样的话可以方便使用logic结构体的元素数据,而不是用参数传递来传递去,不要经常搞那些大写开头的代码,那种是类似于静态的代码包名来调用.

以全局数据为中心去操作来维护,其他的各种函数都是来操作这个全局数据的,而不是数据的传递,围绕一个很大的数据结构把代码给串起来.

Api-service-dao的关系是这样的,层与层之间是互相不接触的,所有的操作的数据都是互相抽象的.

对于service层来说,只认订单,并不认你下面有多少表,(这就是互相抽象的意义)

而上文所说的定义的复杂的数据结构就是在service层.

### golang性能



go里面只要是channel里的东西都是分配在堆中的,channel是在两个协程之间传递的 栈肯定是不行的,必须在堆上

channel其实是有大锁的,如果协程比较多操作一个channel, 性能也是很差的 就是互斥锁

goruntine池的好处还有就是。减少栈空间不够导致的栈分配导致的栈拷贝引起的性能下降





atomic是效率比较高的,lock是保护的一段逻辑,而atomic的操作是保护的一个变量 这个和java差不多 cas嘛

### 源码阅读

var rmdirs []func()for \_, d := range tempDirs() {

dir := d // NOTE: necessary!

os.MkdirAll(dir, 0755) // creates parent directories too

rmdirs = append(rmdirs, func() {

os.RemoveAll(dir)

})

}// ...do some work…for \_, rmdir := range rmdirs {

rmdir() // clean up

}

上图的意思是,我们要先创建一些文件夹,然后在把这些文件夹删除,正常就是先循环创建,然后再循环删除,用闭包的话可以玩另一种操作,就是闭包是可以把外面的局部变量和匿名函数绑在一起,这样的话我们先搞出一个匿名函数,这个函数就是删除dir,然后用闭包的方式就可以弄出一个绑定了变量的匿名函数的list,然后循环执行这个list就可以了..........................

func double(x int) (result int) {

defer func() { fmt.Printf("double(%d) = %d\n", x,result) }()

return x + x

}

\_ = double(4)

这个的结果是8,defer是会在return里面的语句执行完才会执行的.

func triple(x int) (result int) {

defer func() { result += x }()

return double(x)

}

fmt.Println(triple(4)) // "12"

defer函数可以在返回前操作返回值的.

for \_, filename := range filenames {

f, err := os.Open(filename)

if err != nil {

return err

}

defer f.Close() // NOTE: risky; could run out of file descriptors

// ...process f…

}

这种循环的defer要注意,在所有文件被操作完之前,是不会有任何defer被执行的

### 面向对象

独立自主的就叫函数,而挂在结构体上的就是方法....

方法的前面的那个 (l \*COmputelogic) 这个l叫做接收器.

方法也可以这么玩

type Path []Point// Distance returns the distance traveled along the path.func (path Path) Distance() float64 {

sum := 0.0

for i := range path {

if i > 0 {

sum += path[i-1].Distance(path[i])

}

}

return sum

}

我们可以给任意类型的数据比如string啊 int啊什么的给type ss string了,然后在给这个ss上挂上方法.

type Values map[string][]string// Get returns the first value associated with the given key,// or "" if there are none.

func (v Values) Get(key string) string {

if vs := v[key]; len(vs) > 0 {

return vs[0]

}

return ""

}

各种定义的什么 map list啥的各种结构都可以type成类型然后在上面挂方法.

func (list \*IntList) Sum() int {

if list == nil {

return 0

}

return list.Value + list.Tail.Sum()

}

nil也是可以调用method的.

没有继承

type Point struct{ X, Y float64 }

type ColoredPoint struct {

Point

Color color.RGBA

}

我们可以除了用语法糖直接访问里面结构的数据,也可以直接用外面的结构访问里面的方法.

看起来这个像是是一个继承,但是golang里是没有继承的概念的,继承是is a的概念

而golang这个应该是has a的概念,

因为如果你一个函数的参数是point,你把colorpoint传进去是不行的.

所以这些东西其实也就是一个语法糖,你用c去直接访问point的方法其实就是c.point.point的方法的简写.

这种匿名成员可以是结构体也可以是结构体指针,都一样的都可以直接.访问

type Atest struct {  
 I int  
 \*Btest  
}  
func (b \*Btest) Print() {  
 fmt.Println(11)  
}  
func main() {  
 a := Atest{  
 I: 1,  
 }  
 a.Print()  
}

这样所以a里面的b是nil的,但是仍然可以调用b的方法的,但是如果方法里面要是访问了b的内容,那么就会空指针异常了.

distanceFromP := p.Distance // method value

fmt.Println(distanceFromP(q)) // "5"

上面说结构体指针调函数的时候其实是方法的选择器,这就看出来了,这个确实是选择器,而且还是有返回值的

### 断言

类型断言是一个使用在接口值上的操作。语法上它看起来像x.(T)被称为断言类型，这里x表示一个接口的类型和T表示一个类型。一个类型断言检查它操作对象的动态类型是否和断言的类型匹配。

目前我们遇到过的断言也就是判断error的类型,其实error也是一个接口呢.

var w io.Writer = os.Stdout

f, ok := w.(\*os.File) // success: ok, f == os.Stdout

b, ok := w.(\*bytes.Buffer) // failure: !ok, b == nil

以前老觉得go的error没有类型,不知道传过来的是什么error 我想根据不同的err做不同的判断,其实呢, 我们自己定义类型实现error接口,然后在外面用断言就可以判断出是不是某种类型的error了

func writeString(w io.Writer, s string) (n int, err error) {

type stringWriter interface {

WriteString(string) (n int, err error)

}

if sw, ok := w.(stringWriter); ok {

return sw.WriteString(s) // avoid a copy

}

return w.Write([]byte(s)) // allocate temporary copy

}

判断一个动态的对象是否包含某个函数..

func formatOneValue(x interface{}) string {

if err, ok := x.(error); ok {

return err.Error()

}

if str, ok := x.(Stringer); ok {

return str.String()

}

// ...all other types...

}

注意断言有两种使用方式,

1,判断一个接口变量是否是他的某个实现类

2,判断一个对象,是否实现了某个接口.

当设计一个新的包时，新手Go程序员总是先创建一套接口，然后再定义一些满足它们的具体类型。这种方式的结果就是有很多的接口，它们中的每一个仅只有一个实现。不要再这么做了。这种接口是不必要的抽象；

func sqlQuote(x interface{}) string {

switch x := x.(type) {

case nil:

return "NULL"

case int, uint:

return fmt.Sprintf("%d", x) // x has type interface{} here.

case bool:

if x {

return "TRUE"

}

return "FALSE"

case string:

return sqlQuoteString(x) // (not shown)

default:

panic(fmt.Sprintf("unexpected type %T: %v", x, x))

}

}

Interface{}的对象可以用.(type)的这个断言.

断言有两种方式,一种是 xx, ok := xx(interface{}).(类型或接口)判断这个东西是不是这个类型或者是否实现了这个接口.

还有一种是 switch xx.(type) 可以获取这个xx的类型和包含了那些接口

func Sprint(x interface{}) string {

type stringer interface {

String() string

}

switch x := x.(type) {

case stringer:

return x.String()

case string:

return x

case int:

return strconv.Itoa(x)

// ...similar cases for int16, uint32, and so on...

case bool:

if x {

return "true"

}

return "false"

default:

// array, chan, func, map, pointer, slice, struct

return "???"

}

}

上面的例子就是利用断言来看这个interface{}是不是实现来这个接口,

还有一点要注意的是

X := x.(type)

得这么写 此时 x可以用swich来表示type 也还是这个变量

### 并发

Go语言中的并发程序可以用两种手段来实现。本章讲解goroutine和channel，其支持“顺序通信进程”（communicating sequential processes）或被简称为**CSP**。CSP是一种现代的并发编程模型，在这种编程模型中值会在不同的运行实例（goroutine）中传递，尽管大多数情况下仍然是被限制在单一实例中。第9章覆盖更为传统的并发模型：多线程共享内存，如果你在其它的主流语言中写过并发程序的话可能会更熟悉一些。第9章也会深入介绍一些并发程序带来的风险和陷阱。

channel是协程之间传递小弟的东西.

它可以让一个goroutine通过它给另一个goroutine发送值信息。每个channel都有一个特殊的类型，也就是channels可发送数据的类型

ch := make(chan int)

ch <- x // a send statement

x = <-ch // a receive expression in an assignment statement

<-ch

ch = make(chan int) // unbuffered channel

ch = make(chan int, 0) // unbuffered channel

ch = make(chan int, 3) // buffered channel with capacity 3

引用类型

Channel还支持close操作，用于关闭channel，随后对基于该channel的任何发送操作都将导致panic异常

done := make(chan struct{})

go func() {

io.Copy(os.Stdout, conn) // NOTE: ignoring errors

log.Println("done")

done <- struct{}{} // signal the main goroutin

channel也可以有struct{}的类型.

func main() {

naturals := make(chan int)

squares := make(chan int)

// Counter

go func() {

for x := 0; ; x++ {

naturals <- x

}

}()

// Squarer

go func() {

for {

x := <-naturals

squares <- x \* x

}

}()

// Printer (in main goroutine)

for {

fmt.Println(<-squares)

}

}

如您所料，上面的程序将生成0、1、4、9、……形式的无穷数列。像这样的串联Channels的管道（Pipelines）可以用在需要长时间运行的服务中，每个长时间运行的goroutine可能会包含一个死循环，在不同goroutine的死循环内部使用串联的Channels来通信。但是，如果我们希望通过Channels只发送有限的数列该如何处理呢？

如果发送者知道，没有更多的值需要发送到channel的话，那么让接收者也能及时知道没有多余的值可接收将是有用的，因为接收者可以停止不必要的接收等待。这可以通过内置的close函数来关闭channel实现：

当一个channel被关闭后，再向该channel发送数据将导致panic异常。当一个被关闭的channel中已经发送的数据都被成功接收后，后续的接收操作将不再阻塞，它们会立即返回一个零值。关闭上面例子中的naturals变量对应的channel并不能终止循环，它依然会收到一个永无休止的零值序列，然后将它们发送给打印者goroutine。

没有办法直接测试一个channel是否被关闭，但是接收操作有一个变体形式：它多接收一个结果，多接收的第二个结果是一个布尔值ok，ture表示成功从channels接收到值，false表示channels已**经被关闭并且里面没有值可接收(注意这里是已经关闭并且没有可以接收的东西)**。使用这个特性，我们可以修改squarer函数中的循环代码，当naturals对应的channel被关闭并没有值可接收时跳出循环，并且也关闭squares对应的channel.

for {

x, ok := <-naturals

if !ok {

break // channel was closed and drained

}

squares <- x \* x

}

因为上面的语法是笨拙的，而且这种处理模式很常见，因此Go语言的range循环可直接在channels上面迭代。使用range循环是上面处理模式的简洁语法，它依次从channel接收数据，当channel被关闭并且没有值可接收时跳出循环。

func main() {

naturals := make(chan int)

squares := make(chan int)

// Counter

go func() {

for x := 0; x < 100; x++ {

naturals <- x

}

close(naturals)

}()

// Squarer

go func() {

for x := range naturals {

squares <- x \* x

}

close(squares)

}()

// Printer (in main goroutine)

for x := range squares {

fmt.Println(x)

}

}

最终版本

其实你并不需要关闭每一个channel。只有当需要告诉接收者goroutine，所有的数据已经全部发送时才需要关闭channel。不管一个channel是否被关闭，当它没有被引用时将会被Go语言的垃圾自动回收器回收。（不要将关闭一个打开文件的操作和关闭一个channel操作混淆。对于每个打开的文件，都需要在不使用的使用调用对应的Close方法来关闭文件。）

如果我们使用了无缓存的channel，那么两个慢的goroutines将会因为没有人接收而被永远卡住。这种情况，称为goroutines泄漏，这将是一个BUG。和垃圾变量不同，泄漏的goroutines并不会被自动回收，因此确保每个不再需要的goroutine能正常退出是重要的。

注意Add和Done方法的不对称。Add是为计数器加一，必须在worker goroutine开始之前调用，而不是在goroutine中；否则的话我们没办法确定Add是在"closer" goroutine调用Wait之前被调用。并且Add还有一个参数，但Done却没有任何参数；其实它和Add(-1)是等价的。我们使用defer来确保计数器即使是在出错的情况下依然能够正确地被减掉。

ch := make(chan int, 1)for i := 0; i < 10; i++ {

select {

case x := <-ch:

fmt.Println(x) // "0" "2" "4" "6" "8"

case ch <- i:

}

}

协程的退出.

<https://www.jianshu.com/p/2e48dacd56a0>

首先一个channel 被多个协程给for {select}了,其中的一个channel中的内容只会被一个协程消费.

还有就是如果一个channel被close了,那么无论多少个协程select了这个channel,都会瞬间收到0值,我们可以利用这个操作当作广播来用.

在Lock和Unlock之间的代码段中的内容goroutine可以随便读取或者修改，这个代码段叫做临界区

Var mu sync.Mutex

用这个结构体就可以去调用lock和unlock函数了,不过我们基本用不到,因为我们都是分布式锁.

Var mu sync.RWMutex

Mu.Rlock()

Mu.lock()

注意读写锁的写锁不是wlock,而是普通的lock

以前我们的java的时候单利模式的初始化搞的就很复杂

而go用once就可以了.

Var on sync.Once

On.Do(f)

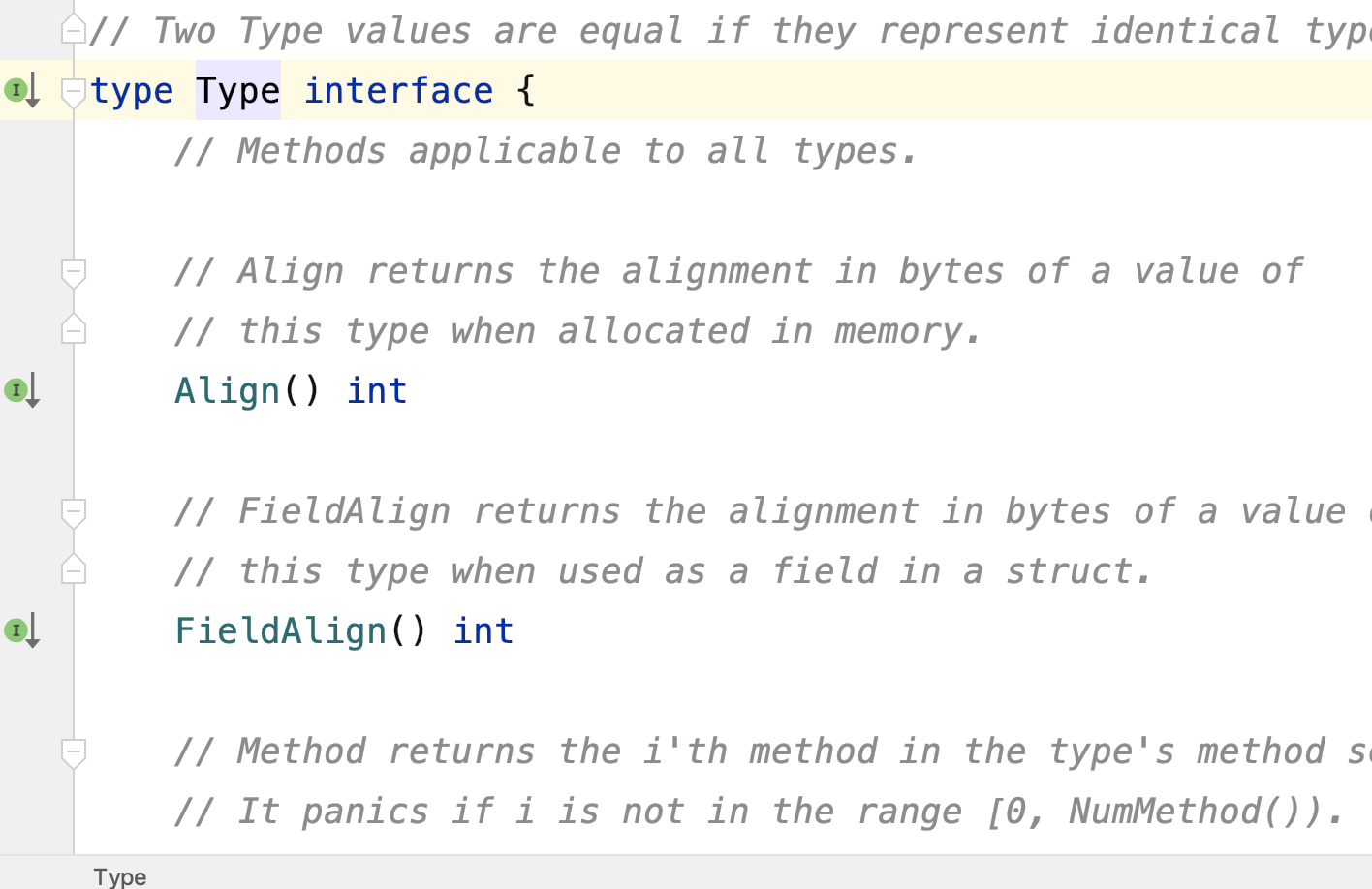
golang自带已经竞争检测器,你编译的时候加上-race就会打印报告,哪里出现锁竞争了都会记录下来.

### 反射

为啥要有放射 fmt.sprinf() 你可能传过来的是int string还可能是各种结构体和各种结构体的map和list,那么我该如果操作每一种类型呢.

t := reflect.TypeOf(3) // a reflect.Type

这个typeof返回的是个TYpe的一个东西,这是一个接口



因为 reflect.TypeOf 返回的是一个动态类型的接口值, 它总是返回具体的类型,即使

你是var xx 接口名 = 实现结构体

这个xx typeof出来的也是具体实现类的类型

reflect.ValueOf(3)

也是返回这个接口

reflect.ValueOf 的逆操作是 reflect.Value.Interface 方法. 它返回一个 interface{} 类型，装载着与 reflect.Value 相同的具体值:

v := reflect.ValueOf(3) // a reflect.Value

x := v.Interface() // an interface{}

i := x.(int) // an int

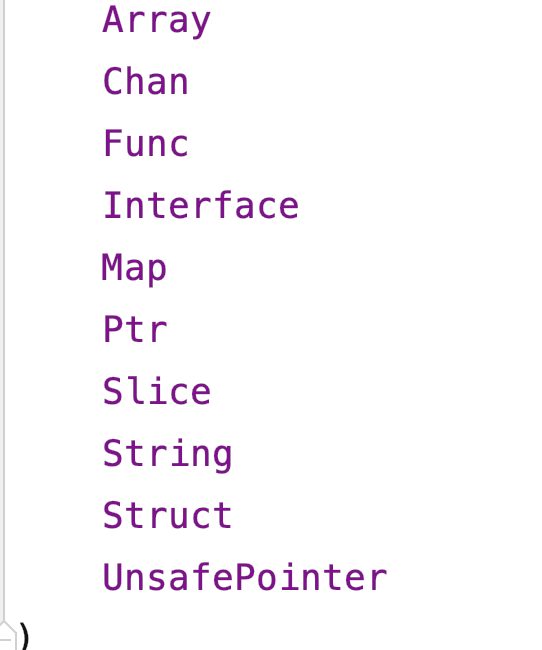
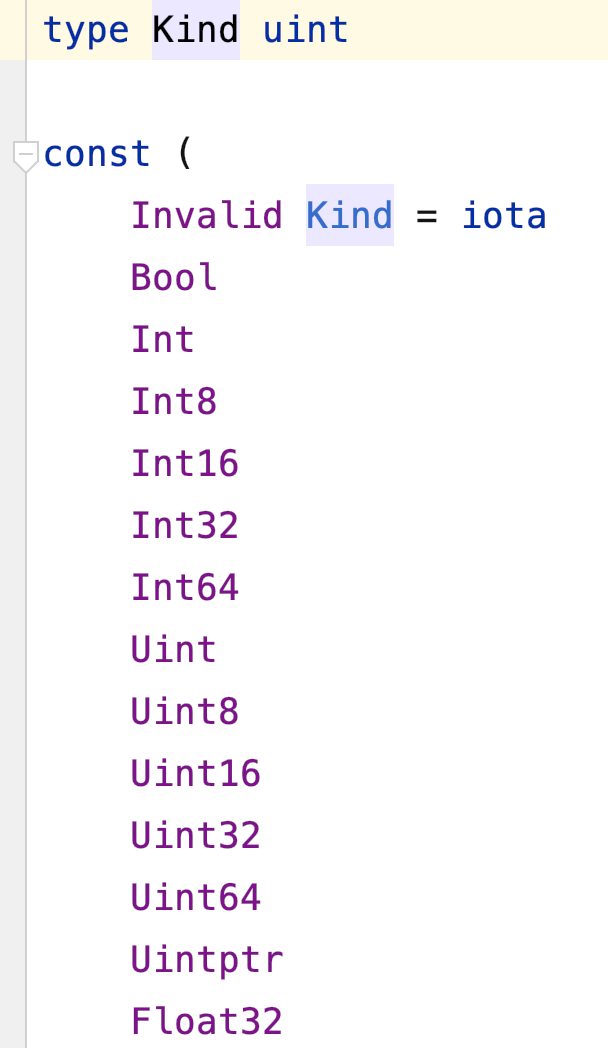
fmt.Printf("%d\n", i) // "3"

reflect.Value 和 interface{} 都能装载任意的值. 所不同的是, 一个空的接口隐藏了值内部的表示方式和所有方法, 因此只有我们知道具体的动态类型才能使用类型断言来访问内部的值(就像上面那样), 内部值我们没法访问. 相比之下, 一个 Value 则有很多方法来检查其内容, 无论它的具体类型是什么.

intercface就像是java的object,虽然本质不是一会事,但是都可以接住所有类型,但是这些类型的特殊的东西都会消失..只是一个传递的容器,类似void类型指针.

而value就类似与java的value对象,你是可以通过value自身的函数去获取到变量的特性的.

我们使用 reflect.Value 的 Kind 方法来替代之前的类型 switch. 虽然还是有无穷多的类型, 但是它们的kinds类型却是有限的: Bool, String 和 所有数字类型的基础类型; Array 和 Struct 对应的聚合类型; Chan, Func, Ptr, Slice, 和 Map 对应的引用类型; interface 类型; 还有表示空值的 Invalid 类型. (空的 reflect.Value 的 kind 即为 Invalid.)



func display(path string, v reflect.Value) {

switch v.Kind() {

case reflect.Invalid:

fmt.Printf("%s = invalid\n", path)

case reflect.Slice, reflect.Array:

for i := 0; i < v.Len(); i++ {

display(fmt.Sprintf("%s[%d]", path, i), v.Index(i))

}

case reflect.Struct:

for i := 0; i < v.NumField(); i++ {

fieldPath := fmt.Sprintf("%s.%s", path, v.Type().Field(i).Name)

display(fieldPath, v.Field(i))

}

case reflect.Map:

for \_, key := range v.MapKeys() {

display(fmt.Sprintf("%s[%s]", path,

formatAtom(key)), v.MapIndex(key))

}

case reflect.Ptr:

if v.IsNil() {

fmt.Printf("%s = nil\n", path)

} else {

display(fmt.Sprintf("(\*%s)", path), v.Elem())

}

case reflect.Interface:

if v.IsNil() {

fmt.Printf("%s = nil\n", path)

} else {

fmt.Printf("%s.type = %s\n", path, v.Elem().Type())

display(path+".value", v.Elem())

}

default: // basic types, channels, funcs

fmt.Printf("%s = %s\n", path, formatAtom(v))

}

}

func formatAtom(v reflect.Value) string {

switch v.Kind() {

case reflect.Invalid:

return "invalid"

case reflect.Int, reflect.Int8, reflect.Int16,

reflect.Int32, reflect.Int64:

return strconv.FormatInt(v.Int(), 10)

case reflect.Uint, reflect.Uint8, reflect.Uint16,

reflect.Uint32, reflect.Uint64, reflect.Uintptr:

return strconv.FormatUint(v.Uint(), 10)

// ...floating-point and complex cases omitted for brevity...

case reflect.Bool:

return strconv.FormatBool(v.Bool())

case reflect.String:

return strconv.Quote(v.String())

case reflect.Chan, reflect.Func, reflect.Ptr, reflect.Slice, reflect.Map:

return v.Type().String() + " 0x" +

strconv.FormatUint(uint64(v.Pointer()), 16)

default: // reflect.Array, reflect.Struct, reflect.Interface

return v.Type().String() + " value"

}

}

这里展示了value和java中的那个反射的各个相似之处,你可以通过value获取到结构体的filed的信息,slice的各个元素,map的key和value等各种信息.

Go语言reflect.Elem()——通过反射获取指针指向的元素类型

我们可以通过调用reflect.Value的CanAddr方法来判断其是否可以被取地址：

x := 2

d := reflect.ValueOf(&x).Elem() // d refers to the variable x

px := d.Addr().Interface().(\*int) // px := &x

\*px = 3 // x = 3

fmt.Println(x)

这个set只能对于那种指针类型的value进行set, 上面也是valueof的时候取了x的地址.

x := 1

rx := reflect.ValueOf(&x).Elem()

rx.SetInt(2) // OK, x = 2

rx.Set(reflect.ValueOf(3)) // OK, x = 3

rx.SetString("hello") // panic: string is not assignable to int

rx.Set(reflect.ValueOf("hello")) // panic: string is not assignable to int

var y interface{}

ry := reflect.ValueOf(&y).Elem()

ry.SetInt(2) // panic: SetInt called on interface Value

ry.Set(reflect.ValueOf(3)) // OK, y = int(3)

ry.SetString("hello") // panic: SetString called on interface Value

ry.Set(reflect.ValueOf("hello")) // OK, y = "hello"

一个可取地址的reflect.Value会记录一个结构体成员是否是未导出成员，如果是的话则拒绝修改操作。因此，CanAddr方法并不能正确反映一个变量是否是可以被修改的。另一个相关的方法CanSet是用于检查对应的reflect.Value是否是可取地址并可被修改的：

fmt.Println(fd.CanAddr(), fd.CanSet()) // "true false"

Unmarshal函数使用了反射机制类修改movie变量的每个成员，根据输入的内容为Movie成员创建对应的map、结构体和slice。unmarshal就是通过set设置的你输入的结构体的没个变量.还有就是你输入的结构体必须你指针,因为set只对指针生效,所以我们的jsonumarsharl就必须输入指针咯.

var data struct {

Labels []string `http:"l"`

MaxResults int `http:"max"`

Exact bool `http:"x"`

}

下面这个就是我要解析的结构体,把字符串解析程结构体.

func Unpack(req \*http.Request, ptr interface{}) error {

if err := req.ParseForm(); err != nil {

return err

}

// Build map of fields keyed by effective name.

fields := make(map[string]reflect.Value)

v := reflect.ValueOf(ptr).Elem() // the struct variable

for i := 0; i < v.NumField(); i++ {

fieldInfo := v.Type().Field(i) // a reflect.StructField

tag := fieldInfo.Tag // a reflect.StructTag

首先我们要解析出这个结构体的各个字段,还有平时后面在结构体后面加的什么gorm啊 json啊 pb那些东西到这里都叫做tag

name := tag.Get("http")

if name == "" {

name = strings.ToLower(fieldInfo.Name)

}

fields[name] = v.Field(i)

}

// Update struct field for each parameter in the request.

for name, values := range req.Form {

f := fields[name]

if !f.IsValid() {

continue // ignore unrecognized HTTP parameters

}

for \_, value := range values {

if f.Kind() == reflect.Slice {

elem := reflect.New(f.Type().Elem()).Elem()

if err := populate(elem, value); err != nil {

return fmt.Errorf("%s: %v", name, err)

}

f.Set(reflect.Append(f, elem))

} else {

if err := populate(f, value); err != nil {

return fmt.Errorf("%s: %v", name, err)

}

}

}

}

return nil

}

func populate(v reflect.Value, value string) error {

switch v.Kind() {

case reflect.String:

v.SetString(value)

case reflect.Int:

i, err := strconv.ParseInt(value, 10, 64)

if err != nil {

return err

}

v.SetInt(i)

case reflect.Bool:

b, err := strconv.ParseBool(value)

if err != nil {

return err

}

v.SetBool(b)

default:

return fmt.Errorf("unsupported kind %s", v.Type())

}

return nil

}

func Print(x interface{}) {

v := reflect.ValueOf(x)

t := v.Type()

fmt.Printf("type %s\n", t)

for i := 0; i < v.NumMethod(); i++ {

methType := v.Method(i).Type()

fmt.Printf("func (%s) %s%s\n", t, t.Method(i).Name,

strings.TrimPrefix(methType.String(), "func"))

}

}

我们也是可以通过反射来获取一个结构体的method的列表的

func hello() {

fmt.Println("Hello world!")

}

func main() {

hl := hello

fv := reflect.ValueOf(hl)

fmt.Println("fv is reflect.Func ?",fv.Kind() == reflect.Func)

fv.Call(nil)

}

普通的函数就可以通过反射轻易的去调用.

func prints(i int) string {

fmt.Println("i =",i)

return strconv.Itoa(i)

}

func main() {

fv := reflect.ValueOf(prints)

params := make([]reflect.Value,1) //参数

params[0] := reflect.ValueOf(20) //参数设置为20

rs := fv.Call(params) //rs作为结果接受函数的返回值

fmt.Println("result:",rs[0].Interface().(string)) //当然也可以直接是rs[0].Interface()

}

带有参数和返回值的func的调用.

Method也是一样的

func main() {  
 t := &TT{}  
  
  
 t1 := reflect.ValueOf(t)  
  
 for i:=0; i < t1.NumMethod(); i++ {  
 t1.Method(i).Call(nil)  
 }  
  
}

if !reflect.TypeOf(key).Comparable() {

panic("key is not comparable")

}

typeof之后有个comparable这个函数,可以判断出是否可以比较呢.

### 底层编程

go语言的实现刻意隐藏了很多底层的细节,这样的话程序刻意更加的稳定,可移植性也会更高.

unsafe包 黑魔法.

fmt.Println(unsafe.Sizeof(float64(0))) // "8"

unsafe.Sizeof函数返回操作数在内存中的字节大小，参数可以是任意类型的表达式，

由于地址对齐这个因素，一个聚合类型（结构体或数组）的大小至少是所有字段或元素大小的总和，或者更大因为可能存在内存空洞。内存空洞是编译器自动添加的没有被使用的内存空间，用于保证后面每个字段或元素的地址相对于结构或数组的开始地址能够合理地对齐（译注：内存空洞可能会存在一些随机数据，可能会对用unsafe包直接操作内存的处理产生影响）。

下面的三个结构体虽然有着相同的字段，但是第一种写法比另外的两个需要多50%的内存。

struct{ bool; float64; int16 } // 3 words 4words

struct{ float64; int16; bool } // 2 words 3words

struct{ bool; int16; float64 } // 2 words 3words

go语言调用c语言,有一个叫做cgo的方式.

#include <bzlib.h>

int bz2compress(bz\_stream \*s, int action,

char \*in, unsigned \*inlen, char \*out, unsigned \*outlen) {

s->next\_in = in;

s->avail\_in = \*inlen;

s->next\_out = out;

s->avail\_out = \*outlen;

int r = BZ2\_bzCompress(s, action);

\*inlen -= s->avail\_in;

\*outlen -= s->avail\_out;

s->next\_in = s->next\_out = NULL;

return r;

}

上面是一个.c的文件 我们这样用里面的函数

import "C"

import (

"io"

"unsafe"

)

type writer struct {

w io.Writer // underlying output stream

stream \*C.bz\_stream

outbuf [64 \* 1024]byte

}

// NewWriter returns a writer for bzip2-compressed streams.func NewWriter(out io.Writer) io.WriteCloser {

const blockSize = 9

const verbosity = 0

const workFactor = 30

w := &writer{w: out, stream: C.bz2alloc()}

C.BZ2\_bzCompressInit(w.stream, blockSize, verbosity, workFactor)

return w

}

Unsafe.pointer

类似于c语言的void类型的指针,他可以包含任意类型变量的地址,当然我们不可以直接通过\*p来获取unsafe.pointer指针指向的真实变量的值.

一个普通的\*T类型指针可以被转化为unsafe.Pointer类型指针，并且一个unsafe.Pointer类型指针也可以被转回普通的指针，被转回普通的指针类型并不需要和原始的\*T类型相同。通过将\*float64类型指针转化为\*uint64类型指针，我们可以查看一个浮点数变量的位模式。

func Float64bits(f float64) uint64 { return \*(\*uint64)(unsafe.Pointer(&f)) }

Unsafe.uinuptr

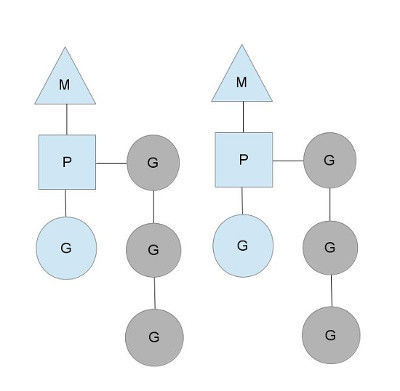
就是代表一个指针的值,是一个整型的变量,可以进行算数运算,那么我们就可以实现对指针进行偏移这种操作了.

### Runtime

num := runtime.NumCPU() //获取主机的逻辑CPU个数

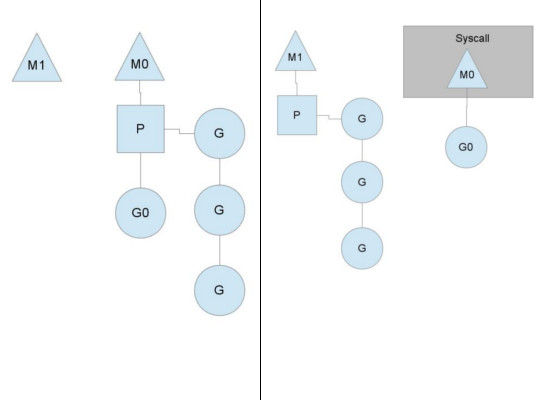
runtime.GOMAXPROCS(num) //设置可同时执行的最大CPU数

groutine能拥有强大的并发实现是通过GPM调度模型实现，下面就来解释下goroutine的调度模型。



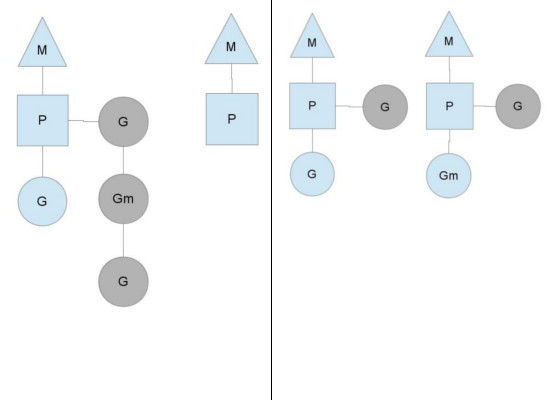
Go的调度器内部有四个重要的结构：M，P，S，Sched，如上图所示（Sched未给出）  
M:M代表内核级线程，一个M就是一个线程，goroutine就是跑在M之上的；M是一个很大的结构，里面维护小对象内存cache（mcache）、当前执行的goroutine、随机数发生器等等非常多的信息  
G:代表一个goroutine，它有自己的栈，instruction pointer和其他信息（正在等待的channel等等），用于调度。  
P:P全称是Processor，处理器，它的主要用途就是用来执行goroutine的，所以它也维护了一个goroutine队列，里面存储了所有需要它来执行的goroutine

Sched：代表调度器，它维护有存储M和G的队列以及调度器的一些状态信息等。



当MO返回时，它必须尝试取得一个P来运行goroutine，一般情况下，它会从其他的OS线程那里拿一个P过来，  
如果没有拿到的话，它就把goroutine放在一个global runqueue里，然后自己睡眠（放入线程缓存里）。所有的P也会周期性的检查global runqueue并运行其中的goroutine，否则global runqueue上的goroutine永远无法执行。

另一种情况是P所分配的任务G很快就执行完了（分配不均），这就导致了这个处理器P很忙，但是其他的P还有任务，此时如果global runqueue没有任务G了，那么P不得不从其他的P里拿一些G来执行。一般来说，如果P从其他的P那里要拿任务的话，一般就拿run queue的一半，这就确保了每个OS线程都能充分的使用，如下图：



1:1模型，1个内核空间线程运行一个用户空间线程。这种充分利用了多核系统的优势但是上下文切换非常慢，因为每一次调度都会在用户态和内核态之间切换。（POSIX线程模型(pthread)，Java）

goroutine的优势在于上下文切换在完全用户态进行，无需像线程一样频繁在用户态与内核态之间切换，节约了资源消耗。

p的数量如何确认

由启动时环境变量$GOMAXPROCS或者是由runtime的方法GOMAXPROCS()决定（默认是1）。这意味着在程序执行的任意时刻都只有$GOMAXPROCS个goroutine在同时运行。

M与P的数量没有绝对关系，一个M阻塞，P就会去创建或者切换另一个M，所以，即使P的默认数量是1，也有可能会创建很多个M出来。

P何时创建：在确定了P的最大数量n后，运行时系统会根据这个数量创建n个P。

M何时创建：没有足够的M来关联P并运行其中的可运行的G。比如所有的M此时都阻塞住了，而P中还有很多就绪任务，就会去寻找空闲的M，而没有空闲的，就会去创建新的M。

如果一个P的就绪队列所有任务都执行完了，那么P会尝试从其他P的就绪队列中取出一部分到自己的就绪队列中，以保证每个P的就绪队列都有任务可以执行。

### Sync.Pool

那么这个池子的目的是什么呢？其实一句话就可以说明白，就是为了复用已经使用过的对象，来达到优化内存使用和回收的目的。说白了，一开始这个池子会初始化一些对象供你使用，如果不够了呢，自己会通过new产生一些，当你放回去了之后这些对象会被别人进行复用，当对象特别大并且使用非常频繁的时候可以大大的减少对象的创建和回收的时间。

## Gc

golang的gc之路

1.1 Stw stop the world

1.3 mark stw sweep

1.5 三色标记法

1.8 hybrid write barrier 混合写屏障

引用计数法,这个对象被其他对象引用了这个对象维护的计数就加以,当引用该对象的对象被销毁时计数就减1,为0了就直接删.有很多脚本语言还是这样操作的,这样回收比较及时,但是整体性能不高,无法解决循环引用.

标记清除法 标记从根变量开始迭代得遍历所有被引用的对象，对能够通过应用遍历访问到的对象都进行标记为“被引用”；标记完成后进行清除操作. 但是从标记开始,就会暂停所有正常代码,直到清除完才会继续运行.

还有分代收集的就java那种.

gcTriggerAlways: 强制触发GC

gcTriggerHeap: 当前分配的内存达到一定阈值时触发，这个阈值在每次GC过后都会根据堆内存的增长情况和CPU占用率来调整

gcTriggerTime: 当一定时间没有执行过GC就触发GC（2分钟）

gcTriggerCycle: **runtime.GC()调用**

黑色：该对象已标记，且其后代已处理

灰色：该对象已标记，但其后代未处理

白色：未被标记

初始化阶段，所有的对象都是白色

从根对象遍历，扫描到可达对象后标记为灰色，加入灰色队列

扫描灰色队列，将其置为黑色，其引用对象标记为灰色

重复上一步，直到灰色队列为空，此时只存在黑色对象和白色对象，回收白色对象

Go的GC是并行GC，大部分是可以和用户代码同时运行的

主要分为四个阶段：

Mark Setup：初始化GC任务，开启写屏障和辅助GC，这个过程需要STW

Mark：从根对象扫描，进行标记，直到灰色队列为空，这个过程可以并行执行

Mark Termination：完成标记，re-scan在检查一下，因为在Mark过程中，很有可能有新的对象分配或赋值，这些都会通过写屏障记录下来，这个过程也会STW。

Sweep：根据标记结果，回收所有的白色对象,这个可以并行执行

目前整个GC流程会进行两次STW(Stop The World), 第一次是Mark阶段的开始, 第二次是Mark Termination阶段

第一次STW会准备根对象的扫描, 启动写屏障(Write Barrier)和辅助GC(mutator assist).

第二次STW会重新扫描部分根对象, 禁用写屏障(Write Barrier)和辅助GC(mutator assist).

可以看出，一个白色对象被黑色对象引用，是注定无法通过这个黑色对象来保证自身存活的，**与此同时，如果所有能到达它的灰色对象与它之间的可达关系全部遭到破坏，那么这个白色对象必然会被视为垃圾清除掉**。 故当上述两个条件同时满足时，就会出现对象丢失的问题。

插入屏障拦截将白色指针插入黑色对象的操作，标记其对应对象为灰色状态，这样就不存在黑色对象引用白色对象的情况了.....

被删除的对象，如果自身为灰色或者白色，那么被标记为灰色。

混合写屏障

1、GC开始将栈上的对象全部扫描并标记为黑色(之后不再进行第二次重复扫描，无需STW)，

2、GC期间，任何在栈上创建的新对象，均为黑色。

3、被删除的对象标记为灰色。

4、被添加的对象标记为灰色。

## 数据库

离线表 就是 binlog同步的那种表,就是离线表,不能和数据在线使用

扫离线表,不要扫在线表.

hot和iot iot遍历是有序的.

Redolog undolog binlog errlog slowquerylog

undolog保存了事务发生之前的数据的一个版本

get\_json\_object hive的

一个任务表,需要被扫 任务越来越多 扫表效率低,可以不停吧已经结束的任务迁移到别的记录表里

大的json用文档数据库 mongodb来存储

先set 再get可能会有主从延迟问题 解决的方法就是。后面的那个get你去读主库

如果你for update了这里 其他服务也就不能用了,要考虑好啊这个for update

存表情需要有的话需要这样设置db,编码集要设置成utf8mb4

kafka消费binlog同步到es

<https://www.cnblogs.com/Me1onRind/p/11565501.html>

es不要权量存es 存id就可以,因为默认es会给所有字段都加上索引,查询的时候会把索引都加载到内存中,如果你索引内容太多内存加载不上,就会走硬盘,所以最好es里面只存id,通过一些条件检索出来后再用唯一id去db或者hive等库里根据id查询内容就ok了.

为什么有内存缓存呢 因为redis的序列化很耗时,而且还要去访问外面的链接,但是内存缓存不会有序列化的问题的.

班课那边的缓存是1分钟 数据有变化会立马删除缓存.

用自增id标记数据, 如果分表的话就不行了 所以我们还是要自己搞个recordid.

如果1和非1差不多数量，加索引并不能有效优化。 如果1很少非1很多，而你只查询1，那么索引是有效的。但是增加和修改的索引效率会降低，如果修改操作出现了瓶颈那么应该去掉索引。

为啥叫这个名字呢,因为你前后两次读的同一个数据的值不一样,就叫不可重复读了.

### 语法

同一个事务中是读未提交的,前面的写,该事务中后面读的时候是可以读出来的



Non\_unique是0的话就表示这个是唯一的

如果对两个列加了联合唯一索引, 那么在对其中一个加唯一索引是不行的

数据库查出来的东西一定要判断两个返回值

db的分页是走的offset 通过偏移量就给完好了

default这个东西不是不传的时候才会生效, 传了0的话default就不会生效的

mysql分为独立部署和混合部署,混合部署就是这个库有很多业务方在用,不仅是你这个业务的几张表,只不过你们建了不同的database,但是数据库进程是同一个.

mysql带4个从,同步是没问题的

comment是个关键字啊

getwriteconn这就会走到主库那里了,这样可以避免一定的主从问题.

Throttled 数据库会限流,如果qpd过高.

如果是not null的话就必须要有默认值。而text没有默认值的,所以给text设置成not null就会出问题

如果你吧很多东西都弄成json当作字符串存在数据库里, 那么统计数据会非常蛋疼.

这是一个用lua脚本的地方,redis的eval语句就是用来执行lua脚本的,语句中的内容就是先get,如果get到了就去delete

var script string = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end"

 if result, err := redisClient.Eval(script, []string{lockKey(roomId)}, uuid).Result(); err != nil {

  logs.CtxWarn(ctx, "RedisUnLock Eval error: %v", err)

  return false, err

 } else if result.(int64) != 1 {

  logs.CtxWarn(ctx, "RedisUnLock result error: %v", result)

  return false, nil

 }

 return true, nil

public static void usePipeline(){

        ShardedJedis jedis = getShardedJedis();

        ShardedJedisPipeline pipeline = jedis.pipelined();

        long begin = System.currentTimeMillis();

        for(int i = 0;i<count;i++){

            pipeline.set("key\_"+i,"value\_"+i);

        }

        pipeline.sync();

        jedis.close();

pieline报错可能是因为pipeline的长度是0

幂等就是同一个消息来了重复了两次,不会重复执行两次,保持幂等就是执行前去查一下咯,有没有执行过,有没有用过.

opentsdb是基于Hbase的时序数据库[时间序列数据库]。不具备通用性，主要针对具有时间特性和需求的数据，如监控数据、温度变化数据等。opentsdb说是数据库，但并不能称作为数据库，他是在Hbase(HBase才是具有存储功能的)的基础上，进行数据结构的优化和处理，从而适合存储具有时间特性的数据，同时提供特定的工具进行查询等操作。

### 索引

explain select \* from paper\_question\_v2 where question\_id = 2 group by ea\_item\_id;

这个如果questionid和eamitemid是联合所以的话 就会完全的走联合索引,如果是单列索引 就会走quedstionid的单列索引.

单独对一个普通索引去做groupby是会走索引的,但是单独对一个唯一索引groupby就是不会走索引的, 所以说唯一索引和普通的index是有区别的.

而且如果你对普通的字段groupby的话除了不用索引还会有filesort,如果你对有索引的数据groupby是不会有filesort

explain select \* from examination where paper\_id >0 order by paper\_id;

这样排序查询就走索引还没有filesort

explain select \* from examination order by paper\_id;

这样排序不会走索引还会有filesort

max和count这种是函数,加不加是不会影响索引的,索引是查询的过程,函数只是对数据又操作了一波.

explain select paper\_id from examination where paper\_id =2;

这种情况肯定是会走索引的,而且后面的extra还是userindex,表示,直接用索引就可以查询出数据. userindex就是最叼的查询了

explain select \* from examination where paper\_id =2;

这种情况也是会走索引的,而且extra是null,因为这个会回表.

explain select examination\_id from examination where paper\_id =2 and name = '111';

这个paperid加了索引, name没有加, 这个也是会走索引的,extra是个userwhere,因为name没有加索引,所以还是要user where再过滤下. user where是这个意思.

explain select \* from examination where name = '22';

这种呢也是userwhereing 但是name不是索引 type就是all key是空的

explain select paper\_id from paper\_question\_v2 where paper\_id > 2;

这种用索引查询,查询的东西也是索引的, 肯定会走索引,extra是userindes和userwhere

explain select \* from paper\_question\_v2 where paper\_id > 2;

但是如果是这种 也会走索引,但是就是Using index condition

也就是范围查询,如果查的列就是范围就是userindex和userwhere,如果查的列就是where的值就是userindexcondition

alter table examination add index un\_test (paper\_id, create\_time, examination\_type);

explain select \* from examination where paper\_id = 6695587283409622792 and examination\_type = 1;

可以看到这个没有遵循最左原则,但是也不会完全不走索引,第一个还是会走索引的.

联合索引,对于字段的大于小于是会走的, 一个大于一个小于也会走的.

explain select \* from paper\_question\_v2 where paper\_id = 0 order by ea\_item\_id;

这两个是联合索引,一个where,一个order, 是完全走index,也不会有filesort

如果是两个单列索引的话,只会use where的index,后面的order是用的filesort,因为mysql只会用一个索引.联合索引就没有这个问题的.

Order by是可以走索引的,但是前面一般都有where条件,where条件那也是有索引的咯,由于mysql只能用一个索引,所以order的索引和where的索引一般都会用where的索引,但是如果你order by id的话就会导致mysql用了order后面的这个主键索引,但是不用where的索引,导致性能的低下了,一个索引的坑.

联合索引abc, a ab abc可以,ba. cba acb这些顺序乱了的也是可以的

explain select \* from eaitem\_update\_courseware\_record where keshi\_id = 11 or ea\_item\_id = 111 or teacher\_id = 11;

这种必须得是三个都是单列索引才会走索引,走的话是这个样子,

type是index merge,也就是根据各自的索引 最后把结果merge了,extra是user union。也就是meger的时候是吧三个结果结合在一起

explain select \* from eaitem\_update\_courseware\_record where keshi\_id = 11 and ea\_item\_id = 111 and history\_id = 11;

这种有一个是索引,2个 或者三个都是单列索引也只会用一个索引,具体用哪个得用mysql会选择,所以我们才需要用联合索引呀.

听说mysql查询只会认一个索引说得就是这个意思,三个and查询,难道我还要查三个b+树么,我只会查一个b+树, 联合索引不就只是一个b+树么.

create table employees2( 工号 varchar(4) primary key, 姓名 varchar(10) not null default "未知" , 年龄 int );

根据上文对not null 和 default的介绍，当想表插入数据时，如果给姓名字段具体值时不会报错；如果给了姓名字段空值，insert into employees value（"0001",null,22)，会触发not null约束，不会触发default约束，因为default只有在不给字段数据时才会使用，所以此条语句报错（填入了空值）；如果不给姓名字段数据，insert into employees （工号,年龄) value（"0002",22)，会触发default，给姓名默认值"未知"。

所以not null 和 default是两个独立的约束，可以用在一个字段上，但不会交互使用，是两个独立的约束条件。另外约束条件不仅是在插入数据时有用，当我们在对表数据进行修改时，也会受到约束条件的限制。

### 事务

对于innodb引擎来说,在rr级别的时候就通过nextkeylock解决了幻读问题,也就是innodb在rr级别下,就可以实现了序列化级别的功能呢. 但是你得自己去用lock in share model来操作,你不加这个s锁的话是不会解决幻读问题的.

**锁和事务的关系就是锁可以对你的某些操作特殊的提供一些隔离性的加持,但是事务不是通过锁实现的.**

什么叫undolog,其实就是没有被事务操作时候的数据的备份,注意undo的语句.

所以结论就是只有序列化的方式是用了锁的,其他的三个级别都是没有用锁的,锁还是需要我们自己去声明的,事务的保证并不是依赖的锁.

一个事务插入了一个数据内容是222, 未提交,另一个链接直接就之间创建了一个666, 之后223这个commit了, 最后查询出来222是在前面的.

Rc read commit 读已提交

Rr repeated read 可重复读

幻读的产生是由于,这边事务a准备比如计算和, 突然你事务b插入了一个,a读到了计算到和里了,然后你b给回滚了.

其实对于事务的顺序来说,前面的各种写操作,写完库里就有对应的事务id的数据了,也就是对库的影响是在写操作的时候就完成了,commit其实没有做什么,可能就是吧这个事务给标志成完成了.

平时普通的写操作也是有事务的,只不过平时的这饿操作的都是自动提教的.

儿我们的所谓的befign,也就是把自动提交给关了.

一条记录除了了事务id还会有一个删除id(就是哪个事务把这个记录给删除了,记录下)

一致性读就是读快照的意思

所谓的快照就是undolog

所谓的undolog就是增加了两个字段,叫做事务id,也可以叫做快照id, 其实我们的普通的一条记录, 他还有很多的快照记录的,只不过我们能看到的那条记录是最新的.

上面说的我们有一个事务id,并且无论是事务还是我们普通的curd都是有事务id的,那么这个事务id是如何生效的呢.

对于普通的读操作,只读取事务id比自己小的中那条最大的数据,如果这条数据的事务id比自己还大,说明这是在自己这条请求之后写进来的,那么就不要读,(但是如果我们现在的隔离界别是读未提交,也就是我们select的时候取的是最新的,这样就会读最新的,后面事务id大于自己的也会读出来,这样就会脏读了,如果我们读小于自己的就不会脏读了么? 如果事务是先写的,你后来读的,你还是会脏读,所以事务的情况还不能这么简单的考虑)

update操作就是新建一条记录,新记录的事务id就是自己的事务id,并且把老的记录的删除版本id设置为自己的shiwuid.

DELETE将当前事务的版本号保存至行的删除版本号.

所以我们可以看到,正常情况下只会有一条记录没有删除版本号,别的记录都是有删除版本号的.

Read Uncommited任何操作都不加锁 而且读的都是最新的

Read commited 就是读那些已经提交了的, 不会有脏读问题, 但是会有重复读问题 一个数据前后两次读,读出来的内容不一样

Repeated read 解决了重复读,就是通过mvcc实现的.

上面讲了 rc和rr都是通过非锁定一致性读实现的,只不过rc读的是最新的快照,而rr读的是小于自己版本的最新的快照,(注意,那种没有提交的数据是不叫快照的,只有提交了才叫快照)



### 锁

悲观锁

共享锁

共享锁又称为读锁，简称S锁，顾名思义，共享锁就是多个事务对于同一数据可以共享一把锁，都能访问到数据，但是只能读不能修改。

Select \* from for update in share mode

排他锁

排他锁又称为写锁，简称X锁，顾名思义，排他锁就是不能与其他所并存，如一个事务获取了一个数据行的排他锁，其他事务就不能再获取该行的其他锁，包括共享锁和排他锁，但是获取排他锁的事务是可以对数据就行读取和修改。

共享锁是都不能写的,大家都只能读,大家都可以获取,所有共享锁释放了,就可以写了,排他锁是只有一个人能获得,一旦你获得了这个锁就即可读也可以写

Select \* from xxx for update

乐观锁

悲观锁什么共享锁, 排他多 意向锁都是悲观锁 乐观锁就是mvcc

意向锁,意思就是如果你申请了一个行锁,除了这个行被你锁了,这个表的表锁其实别人也不能在申请了,所以mysql会有这么一个逻辑,就是你申请行锁的时候会自动让你把这表的意向锁也申请下来,这样别人申请表锁的时候一看这个表的意向锁已经没了就知道了.

共享意向锁is

排他意向锁ix 对应上面的,明白吧.

间隙锁 gap lock

目的是为了防止幻读,比如你要错误b列的值在15和20之间的数,事务开启的时候本来只有16 19,在你操作之前,有一个事务插入了17,然后这个事务又把17给回滚了,这样就出问题了.

幻读了.

间隙锁这种东西是mysql自己操作的,也不是就事务里面用,别的地方也会用的,与事务不事务没有关系.

间隙锁的意思就是15-20这些即使没有17,我也把17给锁了,后面你在想插入17就会被阻塞了.间隙锁就是在一个范围的操作就会生效的.

所以我们在更新和写操作的时候,尽量where的条件不要是范围,不然这个范围就会是间隔锁. 注意间隔锁就是为了解决幻读问题.

防插入锁（Next-Key Lock）：一个前开后闭区间内的多行，实际上是记录锁和间隔锁的结合

## 索引和锁和事务

clientA锁定了某一行，未提交事务，此时clientB也需要update/delete这一行，此时clientB就会进入等待状态，直到出现Deadlock 。

## Tips

  什么是“可打印字符”呢？为什么要用它来传输8Bit字节码呢？在回答这两个问题之前我们有必要来思考一下什么情况下需要使用到Base64？Base64一般用于在HTTP协议下传输二进制数据，由于HTTP协议是文本协议，所以在HTTP协议下传输二进制数据需要将二进制数据转换为字符数据。然而直接转换是不行的。因为网络传输只能传输可打印字符。什么是可打印字符？在ASCII码中规定，0~31、127这33个字符属于控制字符，32~126这95个字符属于可打印字符，也就是说网络传输只能传输这95个字符，不在这个范围内的字符无法传输。那么该怎么才能传输其他字符呢？其中一种方式就是使用Base64。

Echo $GOPATH

Echo $SHELL 这样可以打印一些生效的环境变量咯.

如果你要是用咯zsh,以后哪些环境变量相关的东西就挪到.zshrc里面咯

售卖流程都是会有createorder和confim order的.

Sd lookup 就是去consul走服务发现流程了.

Tlv type length value

Overload 限流

kerberos. 就是一个单点登录的协议,有一个票据服务,先去这个票据服务获取票据,然后你就可以,访问其他服务不用登陆了,kinit的k就是她

if len(os.Args) != 2 {

fmt.Fprintf(os.Stderr, "Usage: %s host:port", os.Args[0])

os.Exit(1)

}

service := os.Args[1]

这个os.Args os.Args[1]啥的乱七八糟的东西,就是你在执行这个bin文件的时候后面跟的参数

接口去抖就是防止用户手抖点击多次，导致瞬间调用接口两次。

大力这里用的也不是protobuf。如果真用的是protobuf的话，那么你的http请求响应体李的内容你就看不懂了，因为那里都是乱码了

centos yum mac brew debain apt

ssh的生成是由git登陆的邮箱生成的，每输入一次生成的都不一样，但是私钥文件是在你本地的，公钥配置在远端。

设置ssh key免密登陆服务器，把自己的公钥配置在服务器的.ssh文件夹下的一个文件里。

Reflect.TypeOf()这个是可以查看变量类型的,也可以看到go语言也是有指针的指针的,但是呢,对于json的unmarshal来说,这个你传指针,还是指针的指针都是能够处理的.

句柄（handle），有多种意义，第一种解释：句柄是一种特殊的[智能指针](https://baike.baidu.com/item/%E6%99%BA%E8%83%BD%E6%8C%87%E9%92%88/10784135" \t "/Users/yangwenshuo/Documents\\x/_blank) 。当一个应用程序要引用其他系统（如数据库、操作系统）所管理的内存块或对象时，就要使用句柄。 [1]  第二种解释：整个Windows编程的基础。一个句柄是指使用的一个唯一的整数值，即一个4字节(64位程序中为8字节)长的数值，来标识[应用程序](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F/5985445" \t "/Users/yangwenshuo/Documents\\x/_blank)中的不同[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1/17158" \t "/Users/yangwenshuo/Documents\\x/_blank)和同类中的不同的实例，诸如，一个窗口，按钮，图标，滚动条，输出设备，控件或者文件等。

句柄泄漏是进程在调用系统文件之后，没有释放已经打开的文件句柄。