# Golang

## Todo

gc

https://my.oschina.net/u/4054187/blog/5491525

<http://thesecretlivesofdata.com/raft/>

以考虑字节的sonic库

https://github.com/codeyifei/dynamic-goroutine-pool/blob/master/pool.go

## Tips

> filename 清空文件



并发标记/扫描收集器

Stop-the-world（STW）标记/清除 （mark/sweep）是本科计算机科学类中最常见的 GC 算法

简单的STW 标记/扫描（mark/sweep）有非常严重的问题。随着你添加处理器或者堆增长，该算法无法良好工作。但是 -如果你的堆比较小，该算法就能够满足对停顿时间的要求！在这种情况下，你应该使用该算法，以保持你的GC开销足够低。

极端的情况下，也许你在一个拥有数十个核的机器上使用数百 GB 的堆。也许您的服务器正在运行金融市场交易，或搜索引擎，因此低暂停时间对您非常重要。这时候你可能愿意使用虽然降低程序运行速度但是可以并发执行的收集算法。

或者您也许有大批量作业。因为它们是非交互式，所以暂停时间根本不重要。在这种情况下，您最好使用吞吐量高于一切的算法，可以提高工作时间与执行收集时间的比率。

问题是没有单一的算法在所有方面都是完美的。语言运行时也不可能知道您的程序是批处理作业还是交互式延迟敏感型程序。这就是为什么“GC调优”存在的原因。它反映了我们计算机科学的基本限制

代际（generation）假说

自1984年以来，我们发现大多数对象都很“年轻”（在分配之后很快就变成垃圾）。这个情况被称为代际假说（generational hypothesis），是整个 PL 工程领域最强的经验之一。它在不同种类的编程语言中，以及在软件行业几十年的变化中一直是正确的：函数语言，命令式语言，没有值类型的语言和有值类型的语言都是如此。

发现这个事实是非常有用的，因为它意味着 GC 算法可以在设计时利用它。这些新一代垃圾收集器对旧的 SWT 垃圾收集器有很多改进：

GC吞吐量：他们可以更多更快的收集垃圾。

分配性能：分配新的内存不再需要搜索通过堆寻找可用内存，因此内存分配器变得更有效。

程序吞吐量：对象整齐地放在彼此相邻的内存中，这大大提高了缓存利用率。分代垃圾收集器确实需要程序在运行时做一些额外的工作，但是这种降低被改进的高速缓存效果所抵消。

暂停时间：大多数（但不是全部）暂停时间变得更低。

当然也引入一些缺点：

兼容性：实现一个分代垃圾收集器需要能够在内存中移动对象，并且在某些情况下，当程序使用指针时需要执行额外的工作。这意味着GC必须与编译器紧密集成。因此没有用于 C++ 的分代垃圾收集器。

堆开销：这些收集器通过在各种“空间”之间来回复制内存来工作。因为必须有空间来进行复制，这些垃圾收集器增加了一些堆开销。此外，它们需要维护各种指针映射，进一步增加内存开销。

暂停分配：虽然许多GC暂停现在非常短，但有些仍然需要对整个堆执行完全标记/扫描。

调优：代数收集器引入了“年轻代”或“eden空间”的概念，程序性能对这个空间的大小非常敏感。

预热时间：为了响应调优问题，一些收集器通过观测程序的运行以来动态地调整年轻代的大小，这种情况下暂停时间就取决于程序运行多长时间。

分代收集器的优势是如此诱人，因此基本上所有现代 GC 算法都是分代的。分代垃圾收集器可以通过各种其他功能进行增强，典型的现代 GC 将并发，并行，整理和分代集成在一起。

由于 Go 是一种命令式语言，它的值类型，内存访问模式和 C# （.NET 使用分代垃圾收集器）相当。

事实上，Go 程序通常是处理 request/response 任务（如 HTTP 服务器），这意味着 Go 程序表现出强烈的代际行为，Go 团队正在探索潜在的可以利用代际假说的算法，他们称之为“面向请求的垃圾收集器”。这本质上是一个可以策略调优的分代垃圾收集器。在处理请求/响应这种模式时，通过确保年轻代足够大以使通过处理请求产生的所有垃圾都在其中来优化 GC。（高可用架构译者注：指的是 Go下一代垃圾收集器 Transaction-Oriented Collector）

尽管如此，Go 的当前 GC 是不分代的。只是在后台运行标记/扫描。（高可用架构译者注：并发标记清除算法）

这样使暂停时间非常短 ，但使其他因素更糟糕。从我们的基本理论上面我们可以看到：

GC吞吐量：GC时间与堆大小同步增长。简单来说，你的程序使用的内存越多，内存释放速度就越慢，你的计算机花费的时间就越多。如果你的程序没有并行化，你可以不用考虑这个问题。

整理：因为没有整理，GC 过程会产生内存碎片。程序也不会受益于在缓存中整齐排列的内容。

程序吞吐量：因为GC必须在每个周期做很多工作，所以会消耗不少CPU时间。

暂停分布：与程序并发运行的任何垃圾收集器都可能遇到Java中“并发模式失败”的问题：您的程序创建垃圾的速度比GC线程可以清除它快。在这种情况下，runtime别无选择只能完全停止程序，等待GC完成垃圾收集。因此当Go团队声明GC暂停非常低时，该声明只能适用于GC具有足够的CPU时间和空间以完成垃圾回收的情况。另外，由于Go编译器缺乏确保线程可以被快速可靠暂停这一功能，会导致暂停时间是否很低取决于您运行的是什么类型的代码（例如，base64 解码单个 goroutine 中的大 blob 会导致暂停时间上升）。

堆开销：因为通过标记/扫描收集堆非常慢，您需要大量的空间以确保不会遇见“并发模式故障”。 Go默认使用100％的堆开销会让程序需要的内存量增加一倍。

我们可以看到这些权衡：

服务1分配内存多于服务2，因此STW暂停在服务1中较高。但STW暂停持续时间在两个服务上都下降了一个数量级。我们看到切换后，两个服务后在GC中花费的CPU使用率增加了约20％。

在这个特定的情况下，Go 以更慢的收集器为代价换取暂停时间的数量级下降。这是一个好的权衡吗？暂停时间已经足够低吗？

付出更多的硬件成本以获得较低的暂停时间，在一些情况下未必有意义。如果你的服务器暂停时间从 10msec 降低到 1msec，你的用户真的会注意到吗？如果你必须加倍你的机器数量才能达成这一目的呢？

Go 将暂停时间优化作为首要目标，以至于它似乎愿意将程序减慢至任何数量级，以获得较短暂停

与 Java 对比

HotSpot JVM 有几个 GC 算法，您可以在命令行中选择。因为他需要平衡其他各种因素，因此没有一个 GC 算法的目标能将暂停时间降低到 Go 水平。可以通过重新启动程序在 GC 之间切换，因为编译是在程序运行时完成（高可用架构译者注：这里指 JIT 编译器），所以不同算法所需的不同内存屏障可以根据需要编译和优化到代码中。

默认算法是吞吐量收集器（throughput collector）。这是为批处理作业设计的，默认情况下没有任何暂停时间目标。这种默认选择也是人们认为 Java GC 有点吸引力的一个原因：开箱即用，它试图使您的应用程序尽可能快地运行，并尽可能少的内存开销，而暂停时间不是该算法首要考虑的。

如果暂停时间对您更重要，那么您可能需要切换到并发标记/扫描收集器（ CMS concurrent mark / sweep collector）。这是和 Go 使用的 GC 算法最接近的垃圾收集器。但它也是分代的垃圾收集器，这也是为什么它的暂停时间比 Go 的更长的原因：年轻代需要整理并移动对象，而导致应用程序暂停。 CMS 中有两种类型的暂停。第一种，较为短暂可能持续大约 2-5 毫秒。第二种可能会持续 20 毫秒或者更久。 CMS 是自适应的：因为是并发的，所以它必须猜测什么时候可以开始运行 GC（就像 Go）。 CMS 将在运行时调整自己并尝试避免“并发模式故障”。因为堆的大部分是标记/扫描算法（高可用架构译者注：这里说的是老年代，使用 CMS 算法的时候，年轻代并非使用该算法而是使用基于标记/整理的 ParNew，所以严格来说把整理并整理内存的好处算在 CMS 算法头上是有问题的），可能会因为堆碎片而导致问题。

最新一代 Java GC 被称为“ G1”（ garbage first 垃圾优先）。它将在 Java 9 中成为默认算法。它旨在提供一个通用的算法。该算法是针对整个堆的并发的，分代的和整理的算法。 G1 在很大程度上也是自适应的，因为（像所有的 GC 算法）它不能知道你真正想要什么，但它允许你指定首选权衡：只需要告诉它你允许使用的 RAM 最大值和暂停时间目标（以毫秒为单位），它就会尽力满足暂停时间目标。除非你指定不同的目标，否则默认的暂停时间目标大约是 100 毫秒。 G1 会更倾向于让你的应用程序运行的速度快而非暂停少。其每次暂停时间并不完全一致，但大多数都非常快（少于一毫秒），有些暂停因为堆被整理而稍慢（ 50 毫秒）。 G1 的扩展性也非常好。有报告称，人们在 TB 级别堆规模的程序上使用 G1 算法。它还有一些其他功能，如重复数据删除堆中的字符串。

Red Hat 支持的一个项目组开发了一种新的 GC 算法，称为 Shenandoah。代码已经贡献给 OpenJDK，但不会出现在 Java 9 中（除非你使用红帽子的 Java 版本）。这一算法被设计为无论堆多大的情况下，都可以在提供整理的同时保证非常低的暂停时间。其成本是额外的堆开销和更多的内存屏障（高可用架构译者注：同时使用了读写屏障，而上述其他算法都只使用了写屏障）。在这个意义上，它类似于 Azul 的“无暂停”垃圾收集器（ArchNotes 译者注：指的是使用 C4 算法的垃圾收集器，严格来说也并非完全无停顿，只是保证停顿时间在任何情况都小于 10ms, 由于在软实时系统上 OS 带来的误差有可能超过 10ms，因此可以认为是无停顿垃圾收集器）。

结论

本文的重点不是说服你使用不同的编程语言或工具。 只是希望带来对垃圾收集器的正确的理解。垃圾收集是一个非常挑战的工作，很多计算机科学家在上面耗费了数十年，因此不太有可能一晚上就会有一个全新的别人没用过的 GC 算法问世，更有可能的是，声称的新的 GC 算法只是对老的 GC 算法做了一个不同的，而且成熟的 GC 算法不太会考虑的偏门 tradeoff 而已。

但是如果你仅希望减少程序暂停时间，那么请关注 Go GC

mongo也是有索引和 事务的

在MongoDB中，数据存储在文档中，文档中可以组合多个字段、嵌套文档和数组，因此可以存储更复杂的数据类型。MySQL支持结构化查询，能对事前定义好的数据表进行更高级的查询，例如聚合查询与多表查询，而MongoDB只支持简单的基本查询

MySQL的优点是它支持ACID数据完全性，“ACID”是Atomicity（原子性）、Consistency（一致性）、Isolation（隔离性）和Durability（持久性）的首字母组合，而MongoDB只支持ACID per collection（每一个集合）。MySQL也有大量的插件可用来保护数据，例如做数据备份、安全控制和存储引擎切换

MySQL 支持原始 SQL 语句，支持复杂的查询，因此能够处理复杂的数据；

• MySQL 支持强大的回滚管理，这样可以避免不需要的或):错误的更改；

• MySQL 支持外部键，这样可以有效地关联两个表

一般情况同等条件下，MongoDB会比MySQL快 ，主要原因如下：

a.Mongo使用的内存映射技术, 写入数据时候只要在内存里完成就可以返回给应用程序，这样并发量自然就很高。而保存到硬体的操作则在后台异步完成。

b.MongoDB的设计要求你常用的数据（working set)可以放到内存里。这样大部分操作只需要读内存，内存操作当然比较快，这也是MongoDB，经常会和redis比较的原因。前提就是要保证服务器有 足够用的内存，否则性能会严重下降。

c.数据集中存放，减少读写时磁盘寻道的时间，这也是MongoDB的基本思想之一。

并发能力，具体看过有网友测试过，MongoDB和MySQL的并发能力，借鉴一下

早期的版本是没有事务的，因为在MongoDB中，对于单条记录的一个操作是原子性的，一般来说，MongoDB将有关联的数据存储在一起，所以很多操作不像MySQL，需要做多表的操作。

从4.0版本开始，MongoDB支持副本集的事务，4.2支持切片的事务；

对于多表事物的需求场景，MongoDB也是支持的，可以在多个分片、库、表、文档之间实现分布式的事务。

分布式事务会严重影响性能，所以要谨慎使用，当需要事务时，优先考虑是否可通过合理使用MongoDB的内嵌文档和数组，降低使用事务的几率。

要了解MongoDB的事务，要先了解 一下MongoDB的集群

没有join ，连表操作能力弱，所以在复杂查询时，还是关系型数据库更胜一筹。

这是MySQL里的概念，在MongoDB事务提交前，事务外看不到本次修改的内容，隔离级别类似MySQL默认的隔离级别，“可重复读”的级别。

什么时候适合用MongoDB：

1. 数据结构不确定，可能发生改动的场景；

2.事务安全性要求不高，数据扩展要求较高的时候；

什么时候用MySQL：

1. 有事务要求，强一致性要求较高，涉及到金钱的时候；

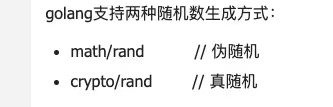
2. 逻辑复杂，有较多join的需求；

没有最好的，只有最合适的，适合自己业务的就是最好的



可以看到有ctx.Done方法和ct.x.Err()方法 确实是在关闭httpclient的时候, ctx变成了cancel 这个select也演示了 怎么做的超时+客户端关闭的中断

说明了go的包已经把链接的监控给做了, 当链接被断开的时候, 可以通过ctc的done感知到的





f, err := os.CreateTemp("", "sample")

check(err)

dname, err := os.MkdirTemp("", "sampledir")

check(err)

fmt.Println("Temp dir name:", dname)

在整个程序执行过程中，我们经常希望创建程序退出后不需要的数据。 临时文件和目录可用于此目的，因为它们不会随着时间的推移而污染文件系统

cRes = tx.CreateInBatches(&contents, 5)

order := WorkOrder{ID: orderID}

getRes := db.Take(&order)

在这种情况下，由于 Save 方法没有明确指定更新条件，它将根据结构体 order 或 order.Contents 中的主键（Primary Key）字段来确定更新的记录。GORM 会自动检测结构体中哪些字段被标记为主键，并使用这些字段作为更新记录的条件。

type WorkOrder struct {

CreatedAt time.Time `json:"created\_at"`

UpdatedAt time.Time `json:"updated\_at"`

ID uint64 `json:"id" gorm:"primaryKey;AUTO\_INCREMENT"`

PoiBid uint64 `json:"poi\_bid" gorm:"index"`

PoiAddr string `json:"poi\_addr" `

PoiImg string `json:"poi\_img"`

UserID uint64 `json:"user\_id" gorm:"index"`

UserExt string `json:"user\_ext"`

Source string `json:"source"`

Type string `json:"type" gorm:"size:32"` // 业务类别

ContentID uint64 `json:"content\_id"` // 业务内部id

ContentType string `json:"content\_type" gorm:"size:32"` // 业务内部类别

ContentMeta string `json:"content\_addr"` // 业务内容的源信息

Callback string `json:"callback"` // 回调信息

Status uint32 `json:"status"` // 审核调度状态

// todo:增加字段记录下游服务的id

Contents []Content `json:"-" gorm:"-:all"`

}

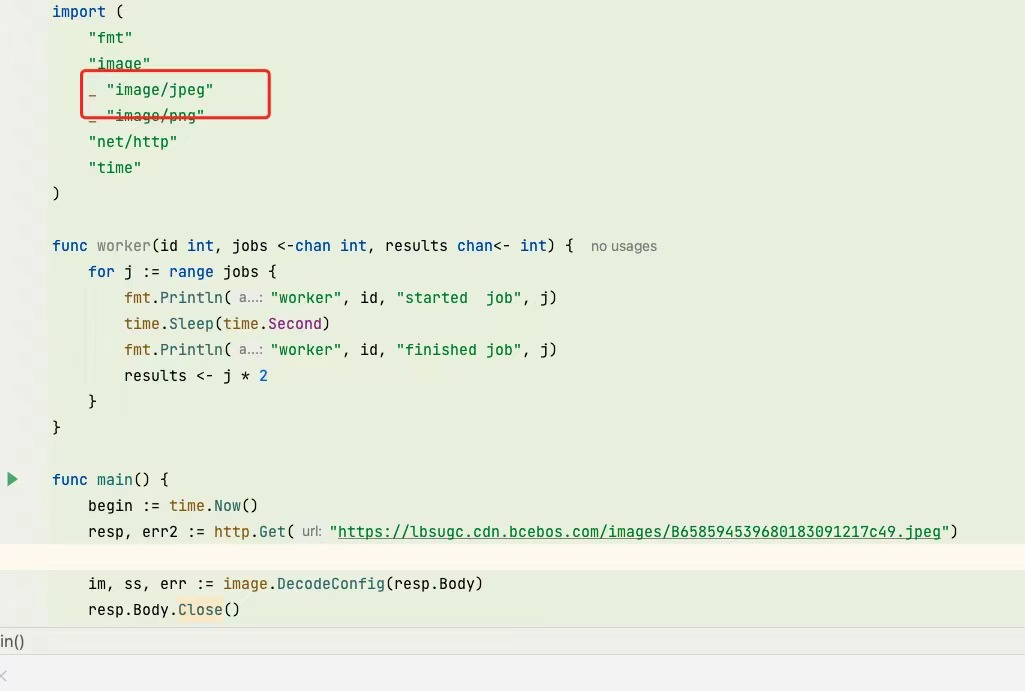
user\_id用的分片 那么每个接口都要有user\_id

sigs := make(chan os.Signal, 1)

signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)

cat /tmp/lines | go run line-filters.go

这也可以 算是stdin



redis事务实现cas

使用WATCH命令和事务可以解决这一问题：

exec(WATCH stock:1001);

if(exec(HGET stock:1001 state) == "in stock") {

exec(MULTI);

exec(HSET stock:1001 state "sold");

exec(EXEC);

}

If key in list:

这个在python里也是很慢的, list只能遍历, 会很慢

WATCH的机制是：在事务EXEC命令执行时，Redis会检查被WATCH的key，只有被WATCH的key从WATCH起始时至今没有发生过变更，EXEC才会被执行。如果WATCH的key在WATCH命令到EXEC命令之间发生过变化，则EXEC命令会返回失败。

前导模糊查询不能命中索引 非前导模糊查询则可以使用索引，可优化为使用非前导模糊查询

更新十分频繁的字段上不宜建立索引：因为更新操作会变更B+树，重建索引。这个过程是十分消耗数据库性能的。

区分度不大的字段上不宜建立索引：类似于性别这种区分度不大的字段，建立索引的意义不大。因为不能有效过滤数据，性能和全表扫描相当。另外返回数据的比例在30%以外的情况下，优化器不会选择使用索引。

业务上具有唯一特性的字段，即使是多个字段的组合，也必须建成唯一索引。虽然唯一索引会影响insert速度，但是对于查询的速度提升是非常明显的。另外，即使在应用层做了非常完善的校验控制，只要没有唯一索引，在并发的情况下，依然有脏数据产生

mysql嵌套子查询效率确实比较低

可以将其优化成连接查询

连接表时，可以先用where条件对表进行过滤，然后做表连接

（虽然mysql会对连表语句做优化）

建立合适的索引，必要时建立多列联合索引

学会分析sql执行计划，mysql会对sql进行优化，所以分析执行计划很重要

分表还是有必要的 不然一个表的索引会太高了 而且读写也会有行锁之类的 会互相堵塞 减少写入效率

hash 分表 均匀 但是扩展性差, 不好扩新的 range按照id数值分容易有热点key都在新的range问题 可以外面用range 每个range里面在分几个子hash

负向条件查询不能使用索引，可以优化为in查询。

负向条件有：!=、<>、not in、not exists、not like等

使用短索引，如果对长字符串列进行索引，应该指定一个前缀长度，这样能够节省大量索引空间



审核中台的二次审核能力 集成到一个函数中, 不要把每个rpc拆成一个步骤 而是把通用的能力拆成一个步骤 多个rpc合并成一个通用的步骤, 给多个service用 应该这样的

比如

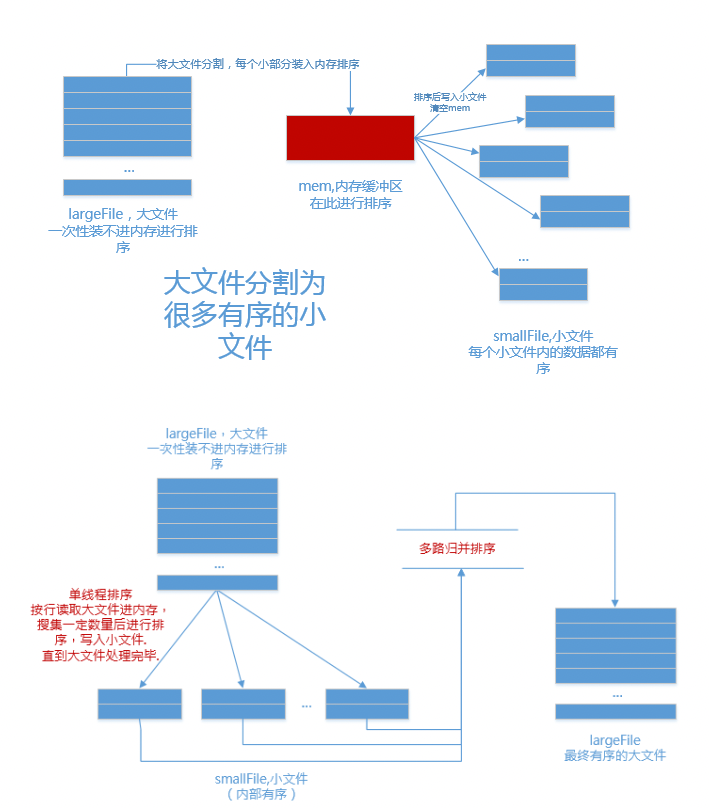
笔记写mysql和redis 这俩应该在一起叫做 savedb 而不是拆开单独怎么样

大问题 dao层暴露的太高了 没有对dao层聚合 dao层就不要暴露在service了

增删改成的聚合

linux自带的sort awk命令 为啥能不管内存就能跑出来 ,因为他们在运行的额时候会swap磁盘的空间 是这样的 聚合对象 service调用聚合对象

5 亿整数的大文件，来排个序 用位图也是可以的 还有就是mapredue 外部排序

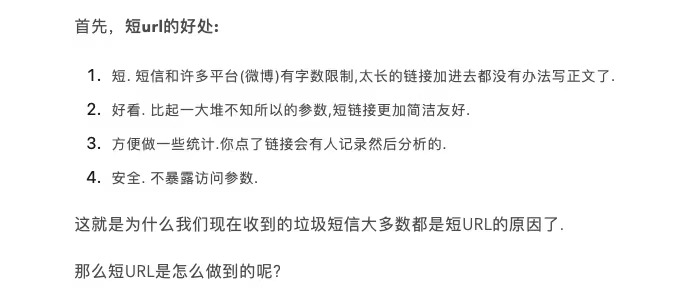


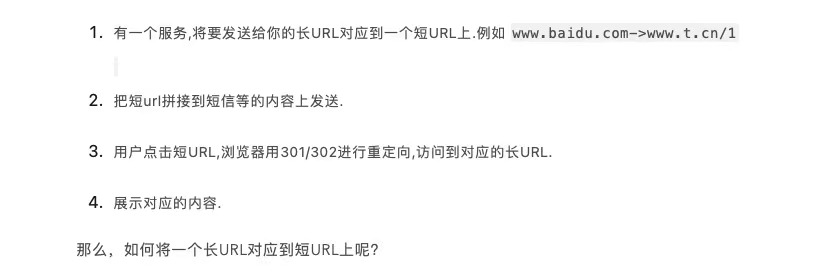
用内存可以承受的大小读取, 然后排序, 然后弄成多个小文件 小文件内部是排好序的

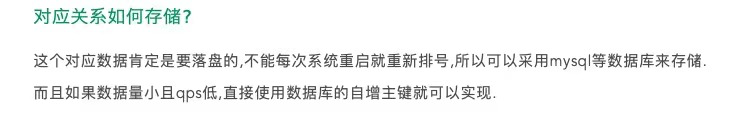
然后就是并归排序 多个小文件加载第一行, 看谁的最小 谁最小谁进去(输出到最终结果文件), 然后弹一个新的 这样循环下去

自动化关联那种 python服务 起了几十个进程反而拉夸 进程变成10个后倒是厉害了, cpu大部分都消耗在切换上了 可以发现, 即使那个自动化服务qps很低, qps也很高, 说明qps消耗不是在业务中而是在切换中

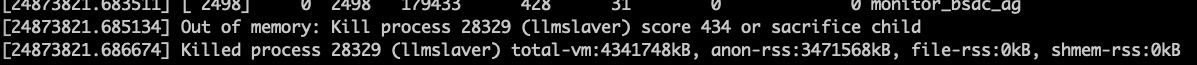
短链接是通过http重定向实现的







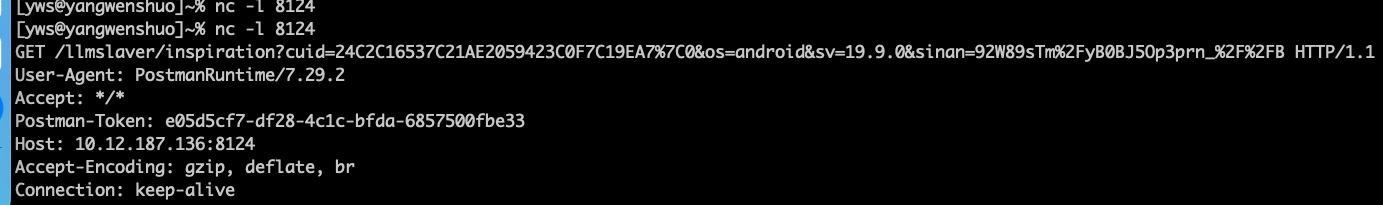




输入dmesg可以查看哪些被内核kill的进程, 中括号里的是系统启动多少秒后发生了这个kill时间, 输入update命令可以查看系统启动了多少秒就可以推断啥时候发生的了.一般都是内存满了才会触发这个kill

nc -l 8124 这个端口不能被别的监听

可以这样的话可以查看给这个端口发的信息 抓包用会打印



父ctx的cancel对子ctx一点影响也没有 除了调用done方法可以拿到信号,, 对于value ctx指针啥的一点影响也没有

select {

case <-ch1:

fmt.Println("Received from ch1")

case <-ch2:

fmt.Println("Received from ch2")

default:

fmt.Println("111")

}

有default才会不阻塞 没有default会阻塞的

基于Metrics收集的监控模式一般在服务内部借助CPU提供的CAS原子性指令实现指标统计，合理使用一般不影响服务性能

 一个月最多31天

Json反序列化成一个interface的话

里面的东西只能断言成 成[]interfcae map[string]interface 搞不成结构体 因为 json反序列的包人家就是这么弄的 同理 里面的整数也只能断言成float64 因为json包弄的时候就是搞的float64

代码里面给一个interface{}设置成int, 后面是可以成功断言int的, 因为json人家设置的时候就是float64

b, \_ := json.Marshal(routeInfo.Prefer)

if err := json.Unmarshal(b, &res); err != nil {

res = []\*vo.DaChePrefer{{}}

}

这个解析失败了 这个res居然长度不是0, 还得我手动复制为0才能清理

func statusLogFieldByErr(err error) logit.Field {

if errors.Is(err, context.Canceled) {

return status499Field

} else if errors.Is(err, context.DeadlineExceeded) {

return status504Field

}

return status500Field

}

地图的前端就是一个爱速搭 我们产出结构化的json 他们压根就不用开发, 什么样式什么的就是json的一个字段 直接页面就变了

g内存少 线程内存中等 进程内存多 所以切换起来 g快 m中 进程慢

大量密集计算一般是用多进程, 可以避免全局解释器锁 大量io是多线程

在编程中，使用多进程和多线程可以实现并发执行任务的目的。下面是一些常见的情况，可以帮助你决定何时使用多进程或多线程：

使用多进程：

当需要同时处理多个独立、相对较大的任务时，多进程是一个很好的选择。每个进程都有自己的独立内存空间，因此可以充分利用多核处理器来提高整体性能。

如果任务之间存在很少的数据共享需求，或者需要进行大量计算的密集型任务时，多进程可以避免全局解释器锁（Global Interpreter Lock，GIL）的限制，从而更好地利用系统资源。

使用多线程：

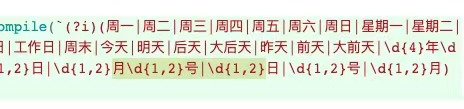
当需要处理多个I/O密集型任务时，多线程通常是比较适合的选择。因为在I/O操作中，线程可以通过切换执行的方式来提高整体的效率。

如果任务之间需要频繁地共享数据，或者需要同步操作（如锁）来避免竞争条件，多线程可以更方便地实现这些功能。

对于一些简单的并行操作，使用多线程可能更轻量级且易于管理。

需要注意的是，多进程和多线程都有各自的优缺点。多进程涉及到进程切换和进程间通信的开销较大，而多线程可能面临线程安全和共享资源的问题。因此，在选择使用多进程或多线程时，应根据具体的需求和场景来进行评估，并综合考虑性能、复杂度、可维护性等因素。

按月分片 分片也可以根据时间分片 一个月一个片



年月日 要在年月前面 不然就匹配不到年月日了

去掉时间的秒

date = date.Truncate(time.Minute)

Watch 监听 consul

package main

import (

"fmt"

consul "github.com/hashicorp/consul/api"

)

func main() {

// 创建Consul客户端

client, err := consul.NewClient(consul.DefaultConfig())

if err != nil {

fmt.Println("无法创建Consul客户端:", err)

return

}

// 创建Watch参数

params := &consul.QueryOptions{

WaitIndex: 0,

WaitTime: 0,

}

// 创建Watch

plan, err := client.KV().Watch("your-key", params)

if err != nil {

fmt.Println("无法创建Watch:", err)

return

}

// 处理Watch事件

go func() {

for {

select {

case <-plan.Stopped:

return

case index, ok := <-plan.KeysChannel():

if !ok {

return

}

// 处理配置变化

fmt.Println("收到配置变化，索引:", index)

// 在这里执行相应的操作，比如重新加载配置等

}

}

}()

// 等待程序退出

<-make(chan struct{})

data-access连接的是mongs，ip是不变的， 相当于mongo集群的代理

快速读文件全部内容, 一次比如读1000个字节, 但是这样可能并不是读的完整的一行, 然后需要再读一次 r.ReadBytes('\n') 这样就保证是完整全是一行行的了

r := bufio.NewReader(f)

var wg sync.WaitGroup //wait group to keep track off all threads

for {

buf := linesPool.Get().([]byte)

n, err := r.Read(buf)

buf = buf[:n]

if n == 0 {

if err == io.EOF {

break

}

if err != nil {

fmt.Println(err)

break

}

return err

}

nextUntillNewline, err := r.ReadBytes('\n')//read entire line

if err != io.EOF {

buf = append(buf, nextUntillNewline...)

}

wg.Add(1)

go func() {

golang也可以有模板方法模式 那种抽象类 就一个结构里有一个接口 那个接口就是不同的东西就OK了

select {

case <-ch1:

fmt.Println("Received from ch1")

case <-ch2:

fmt.Println("Received from ch2")

default:

fmt.Println("111")

}

有default才会不阻塞 没有default会阻塞的



一个月31天

b, \_ := json.Marshal(routeInfo.Prefer)

if err := json.Unmarshal(b, &res); err != nil {

res = []\*vo.DaChePrefer{{}}

}

往一个切片里面解json 解析失败了 切片长度居然不是0, 解析失败了还给我污染了

一个城市分一个表

Update on confict



通过ngixu网关一键转发替换接口 接口a转发到接口/xx/xx/b

窗口分为时间窗口(十分钟自动刷一次)和 次数窗口

异构计算是指在一个计算系统中同时使用多种不同类型的处理器和协处理器，以更高效地完成各种任务。这些处理器可以包括中央处理器（CPU）、图形处理器（GPU）、数字信号处理器（DSP）等。

大key会导致 四个9啥的不够

流式的时候 多个算子都不是空闲的 批量的那种 一个算子没执行完别的都是空闲的,资源也可以不同算子分别管理 人员开发也是



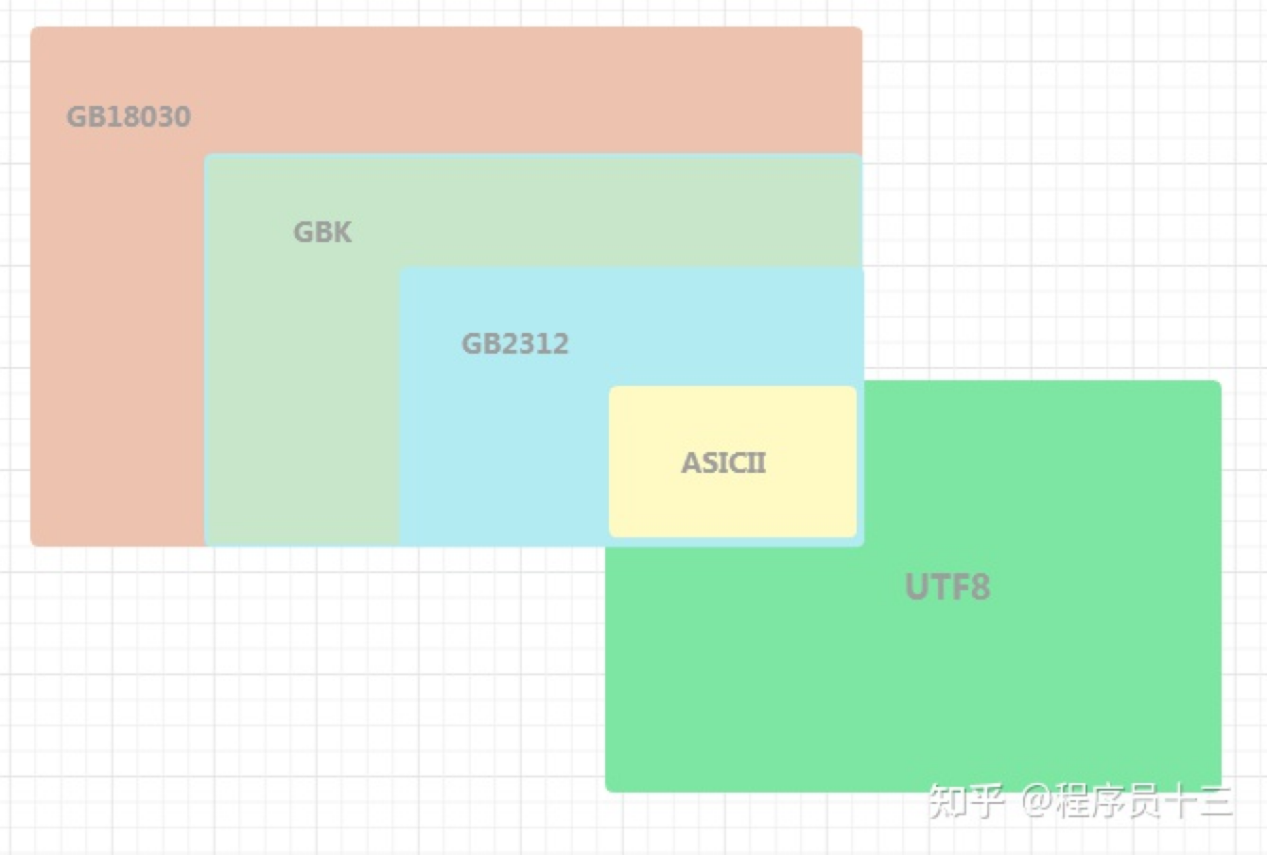
可以发现 cpu不太够 导致内存也高了 释放不及时, cpu够了内容使用率也下降了

对于系统资源如文件、数据库连接、socket 而言，应用程序打开这些资源并执行完业务逻辑之后，必须做的一件事就是要关闭（释放）该资源。

比如 Python 程序打开一个文件，往文件中写内容，写完之后，就要关闭该文件，如果不关闭会出现什么情况呢？极端情况下会出现 Too many open files 的错误，因为系统允许你打开的最大文件数量是有限的。

同样，对于数据库，如果连接数过多而没有及时关闭的话，就可能会出现 Can not connect to MySQL server Too many connections，因为数据库连接是一种非常昂贵的资源，不可能无限制的被创建

—————



gd18030即是字符集也是编码格式, unicode是个字符集 utf8是编码格式 所以gdk转utf8 是先吧gbk转unitcode 然后 unicode转utf8

import (

"fmt"

"time"

)

func worker(id int, jobs <-chan int, results chan<- int) {

for j := range jobs {

fmt.Println("worker", id, "started job", j)

time.Sleep(time.Second)

fmt.Println("worker", id, "finished job", j)

results <- j \* 2

}

}

func main() {

const numJobs = 5

jobs := make(chan int, numJobs)

results := make(chan int, numJobs)

for w := 1; w <= 3; w++ {

go worker(w, jobs, results)

}

for j := 1; j <= numJobs; j++ {

jobs <- j

}

close(jobs)

for a := 1; a <= numJobs; a++ {

<-results

}

}

1, 平时弄文件就可以这样 2, chan当参数传进去 3, 读出来的结果 就可以 通过result写进去 4, 然后再有个worker 从result里面读....

桶漏与令牌桶

package main

import (

"fmt"

"time"

)

func main() {

requests := make(chan int, 5)

for i := 1; i <= 5; i++ {

requests <- i

}

close(requests)

limiter := time.Tick(200 \* time.Millisecond)

for req := range requests {

<-limiter

fmt.Println("request", req, time.Now())

}

burstyLimiter := make(chan time.Time, 3)

for i := 0; i < 3; i++ {

burstyLimiter <- time.Now()

}

go func() {

for t := range time.Tick(200 \* time.Millisecond) {

burstyLimiter <- t

}

}()

burstyRequests := make(chan int, 5)

for i := 1; i <= 5; i++ {

burstyRequests <- i

}

close(burstyRequests)

for req := range burstyRequests {

<-burstyLimiter

fmt.Println("request", req, time.Now())

}

}

package main

import (

"fmt"

"math/rand"

"sync/atomic"

"time"

)

type readOp struct {

key int

resp chan int

}

type writeOp struct {

key int

val int

resp chan bool

}

func main() {

var readOps uint64

var writeOps uint64

reads := make(chan readOp)

writes := make(chan writeOp)

go func() {

var state = make(map[int]int)

for {

select {

case read := <-reads:

read.resp <- state[read.key]

case write := <-writes:

state[write.key] = write.val

write.resp <- true

}

}

}()

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

go func() {

for {

read := readOp{

key: rand.Intn(5),

resp: make(chan int)}

reads <- read

<-read.resp

atomic.AddUint64(&readOps, 1)

time.Sleep(time.Millisecond)

}

}()

}

for w := 0; w < 10; w++ {

go func() {

for {

write := writeOp{

key: rand.Intn(5),

val: rand.Intn(100),

resp: make(chan bool)}

writes <- write

<-write.resp

atomic.AddUint64(&writeOps, 1)

time.Sleep(time.Millisecond)

}

}()

}

time.Sleep(time.Second)

readOpsFinal := atomic.LoadUint64(&readOps)

fmt.Println("readOps:", readOpsFinal)

writeOpsFinal := atomic.LoadUint64(&writeOps)

fmt.Println("writeOps:", writeOpsFinal)

}

go func() {

OuterLoop:

for {

select {

case line, ok := <-linech:

if !ok {

break OuterLoop

} else {

这么一串和 queue := make(chan string, 2)

queue <- "one"

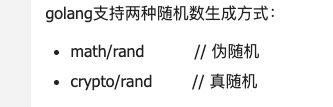
queue <- "two"

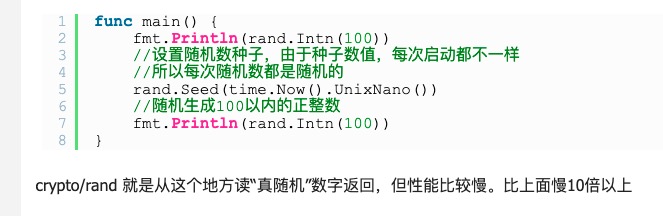
close(queue)

for elem := range queue {

fmt.Println(elem)

}一个意思 不close 也不会从for里面出来, 只有你需要监听多个 channel的时候 才需要for select





UrlParse net/url

package main

import (

"fmt"

"net"

"net/url"

)

func main() {

s := "postgres://user:pass@host.com:5432/path?k=v#f"

u, err := url.Parse(s)

if err != nil {

panic(err)

}

fmt.Println(u.Scheme)

fmt.Println(u.User)

fmt.Println(u.User.Username())

p, \_ := u.User.Password()

fmt.Println(p)

fmt.Println(u.Host)

host, port, \_ := net.SplitHostPort(u.Host)

fmt.Println(host)

fmt.Println(port)

fmt.Println(u.Path)

fmt.Println(u.Fragment)

fmt.Println(u.RawQuery)

m, \_ := url.ParseQuery(u.RawQuery)

fmt.Println(m)

fmt.Println(m["k"][0])

}

服务内存满了行为不可预期了, 明明没有报err但是却没有插入

【原因分析】

假如主库花费200s更新了一张大表，在主从库配置相近的情况下，从库也需要花几乎同样的时间更新这张大表，此时从库延迟开始堆积，后续的events无法更新。

意思就是大事务会导致主从延迟飙升

【解决思路】

拆分大事务，及时提交。

主库一个update语句执行200喵 执行完了开始同步从库 从库执行200s 从库执行这个200秒的时候 ，而从库SQL Thread为单线程应用日志 如果从库是个单线程的主从复制 ,那么这200s内她就废了

/usr/sbin/ss -nl 和netstat -anp都可以查看端口是否被监听 ss(server socker)

update A set number=number+ 5 where id=1;

简单一点理解就是锁的原理。这个时候第一个update会持有id=1这行记录的排它锁，第二个update需要持有这个记录的排它锁的才能对他进行修改，正常的话，第二个update会阻塞，直到第一个update提交成功，他才会获得这个锁，从而对数据进行修改。也就是说，按照关系型数据库的理论，这两个update都成功的话，id=1的number一定会被修改成22。如果不是22，那就是数据库实现的一个严重的bug。

MySQL采用的是边算边发的逻辑，因此对于数据量很大的查询结果来说，不会在server端保存完整的结果集。所以，如果客户端读结果不及时，会堵住MySQL的查询过程，但是不会把内存打爆。

而对于InnoDB引擎内部，由于有淘汰策略，大查询也不会导致内存暴涨。并且，由于InnoDB对LRU算法做了改进，冷数据的全表扫描，对Buffer Pool的影响也能做到可控。

全表扫描还是比较耗费IO资源的，所以业务高峰期还是不能直接在线上主库执行全表扫描的

首先 join on后面是可以跟多个条件的 其次 where 要和on分开

serviver的maxpoen就是指的对面的一个ip的最多的链接

自适应限流,无非就是本实例CPU负载到了就开始限流

maxConcurrency := minLatency \* maxQPS qps乘 接口耗时 就是并发度

访问量大: 水平复制 负载均衡

数据量大: 水平分区 单元化

流程场景多: 逻辑复杂 领域建模

smoke冒烟测试和happy path,如果happy path都没通，那就表示功能必定还没开发完

临时文件

f, err := os.CreateTemp("", "sample")

check(err)

dname, err := os.MkdirTemp("", "sampledir")

check(err)

fmt.Println("Temp dir name:", dname)

在整个程序执行过程中，我们经常希望创建程序退出后不需要的数据。 临时文件和目录可用于此目的，因为它们不会随着时间的推移而污染文件系统



可以看到有ctx.Done方法和ct.x.Err()方法 确实是在关闭httpclient的时候, ctx变成了cancel 这个select也演示了 怎么做的超时+客户端关闭的中断

说明了go的包已经把链接的监控给做了, 当链接被断开的时候, 可以通过ctx的done感知到的

io.ReadCloser 记得close



不引入这两个 是没办法解析图片的

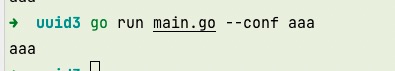
sigs := make(chan os.Signal, 1)

signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)

指定接受 某种信号

user\_id用的分片 那么每个接口都要有user\_id







go tool pprof <http://10.120.45.42:2012/debug/pprof/goroutine>

输入traces 可以查看栈 还可以输出top list

type Aa struct {

A string

}

func (a Aa) Name() {

fmt.Println(a.A)

}

func (a Aa) SetName() {

a.A = "33"

}

func main() {

a := &Aa{A: "22"}

a.Name()

a.SetName()

a.Name()

}

不变

type Aa struct {

A string

}

func (a \*Aa) Name() {

fmt.Println(a.A)

}

func (a \*Aa) SetName() {

a.A = "33"

}

func main() {

a := &Aa{A: "22"}

a.Name()

a.SetName()

a.Name()

}

会变

全看 method定义的是指针还是结构,那个只是自动的帮你加\*或者&符号, 不会改变method的属性

func intSeq() func() int {

i := 0

return func() int {

i++

return i

}

}

闭包可以是和返回匿名函数一起的

t := time.Now()

switch {

case t.Hour() < 12:

fmt.Println("It's before noon")

default:

fmt.Println("It's after noon")

}

func main() {  
 s := "睡个好觉送sfgdfg111"  
 ss := []rune(s)  
 for i, item := range ss {  
 fmt.Println(fmt.Sprintf("%v %c", i, item))  
 }  
}

Rune 配 %c 可以直接输出字符带中文 []rune就是具体字符数

make([]string, 3) 长度已经是3

copy只能copy切片, 且目的长度如果小于源长度会拷贝少,且是浅拷贝

go官方本身就提供了一个net包，然后底层也是用的epoll，但是，采用的是来一个链接就启动一个goroutine，对于短链接来说，可以通过goroutine池进行复用，可是对于长链接服务或者工具如rpc等，如果继续使用官方自带的net库就会有大量的goroutine进行常驻，熟悉go的调度模型mpg就会知道，goroutine并不是越多，速度就越快。当goroutine以几十万或者上百万个的时候程序就会非常缓慢。不仅仅如此，因为goroutine是属于有栈协程，所以会消耗大量的内存。所以对于高并发流量或者大量长连接服务场景来说，go的net包就不是特别的适合，所以需要单独开发调度epoll，只需要几十个goroutine就可以维护几十万上百万链接以及百万流量，可以大大节省内存资源以及延迟消耗

unicode是asccii的扩展 是更全的assccii,unicode是个字符集,utf8是对这个的变长优化 但是用的还是那个字符集,strconv.QuoteToASCII 可以把字符串转成uniode这个字符集的

go func() {  
 wg.Wait()  
 close(ach)  
}

正如我们所看到的，将 n 个元素添加到 map 中，然后删除所有元素意味着在内存中保留相同数量的 bucket。因此，我们必须记住，因为 Go map 只能增加大小，所以它的内存消耗也会增加。没有自动化的策略来缩小它。如果这导致高内存消耗，我们可以尝试不同的选项，例如强制 Go 重新创建 map 或使用指针检查是否可以优化。



把chan在里面创建, 返回出去, 外面监听

把chan当做一个返回值 把ctx传到里面 在里面监听done













子线程超时

通过链接的参数设置超时

设置超时时间有2种方式：context.WithTimeout 与 context.WithDeadline，WithTimeout 是设置一段时间后，WithDeadline 是设置一个截止时间点，WithTimeout 最终也会转换为 WithDeadline。

strings.Join()



要整体gbk编码

如果队列消费有问题, 可以让队列从指定重新消费一遍

go get icode.baidu.com/baidu/gdp/...

map是无序的 但是map生成的json的key是有序的

这个最后必须要加上个\n

command := exec.Command("sh")

in := bytes.NewBuffer(nil)

out := bytes.NewBuffer(nil)

command.Stdin = in

command.Stdout = out

command.Stderr = os.Stderr

in.WriteString(cmd)

in.WriteString("exit\n")

if err := command.Run();

这里这个err就是shell执行失败的那个err, 比如hdfs文件不存在就会返回err, 语义和shell的err一个意思

pthread -> bthread 就是线程切协程 qps升高 cpu下降 因为线程切换占太多了

map[string]any{

if poiBid, ok := query["bid"]; ok {

bid, \_ = strconv.ParseUint(poiBid.(json.Number).String(), 10, 64)

}

这种ok也可以写里面



结构体也可以搞commen

for k, v := range c.mlMap {

mtype := k

materials := v

newFunc := risker.RiskerFactory[c.param.Card][mtype]

if newFunc != nil {

wg.Add(1)

go func() {

defer wg.Done()

risker := newFunc(materials, c)

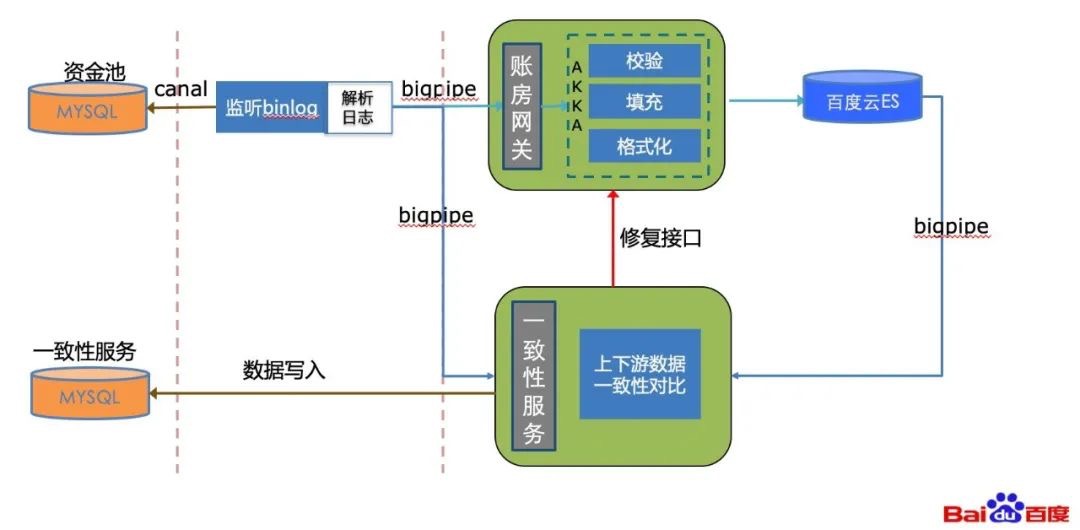
risker.Risk(ctx)

}()

}

}

注意这种for循环开协程的一定要注意 要把item k v复制一份



我们通过探针记录下请求相关的业务日志文件名、日志偏移量，并且存储数据库中。当用户在Jarvis管理端检索调用链相关的业务日志时，系统会先通过调用链 ID 去获取相关的虚拟容器地址、日志文件名、日志偏移量等元数据信息，然后通过这些元数据去具体的容器中取到完整的日志内容，最后展现给用户。

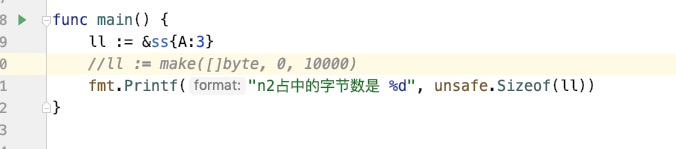
大量日志数据怎么存储!!!!!!!!!!!!!!!!!!!!!

支付还要搞个影子对账服务

Golang Present 是Golang 的 PPT 工具，Golang的分享很多是用这个工具来做的

安装：go get golang.org/x/tools/cmd/present

文档: <https://pkg.go.dev/golang.org/x/tools/present>



可以看到ll的sizeof 地址, 打印的是8, 可以看到我这个确实是64位的.

deadlock就是那种互斥资源导致的,就是行被锁了互相等待,或者有人锁表了.

bytes.HasPrefix(aa, bb)

var encodeStatePool sync.Pool 不错 我看很多包里面的都用了这个,加速对象的创建

延迟队列的set的拿取和delete,和塞到list里面可以写在lua里面, 搞成原子的

if c := len(p.elementRequests); c > 0 { 获取个c的变量

for reqKey, req = range p.elementRequests {

break

} 遍历获取map其中一个

type NewConnFunc func(ctx context.Context) (net.Conn, error)

// Trans 转换为原始的 NewElementFunc

func (nf NewConnFunc) Trans(p \*connPool) NewElementFunc {

return func(ctx context.Context, pool NewElementNeed) (Element, error) {

raw, err := nf(ctx)

if err != nil {

return nil, err

}

vc := newPConn(raw, p)

return vc, nil

}

}

这个把函数定义成一个类型, 然后给这个类型加上一个方法做转换.

对于connpool conn也有一个元素可以是pool

poi := PoiField{}

if err := poi.getData(ctx, req.Bid); err != nil {

最好这样写

filewatcher可以监控配置文件的变化

invalid connection 一般都是有事务死锁 导致的, 或者事务太耗时,链接被打满

所以说int64并不是线程安全的, 因为32位系统cpu一次只能操作32位.

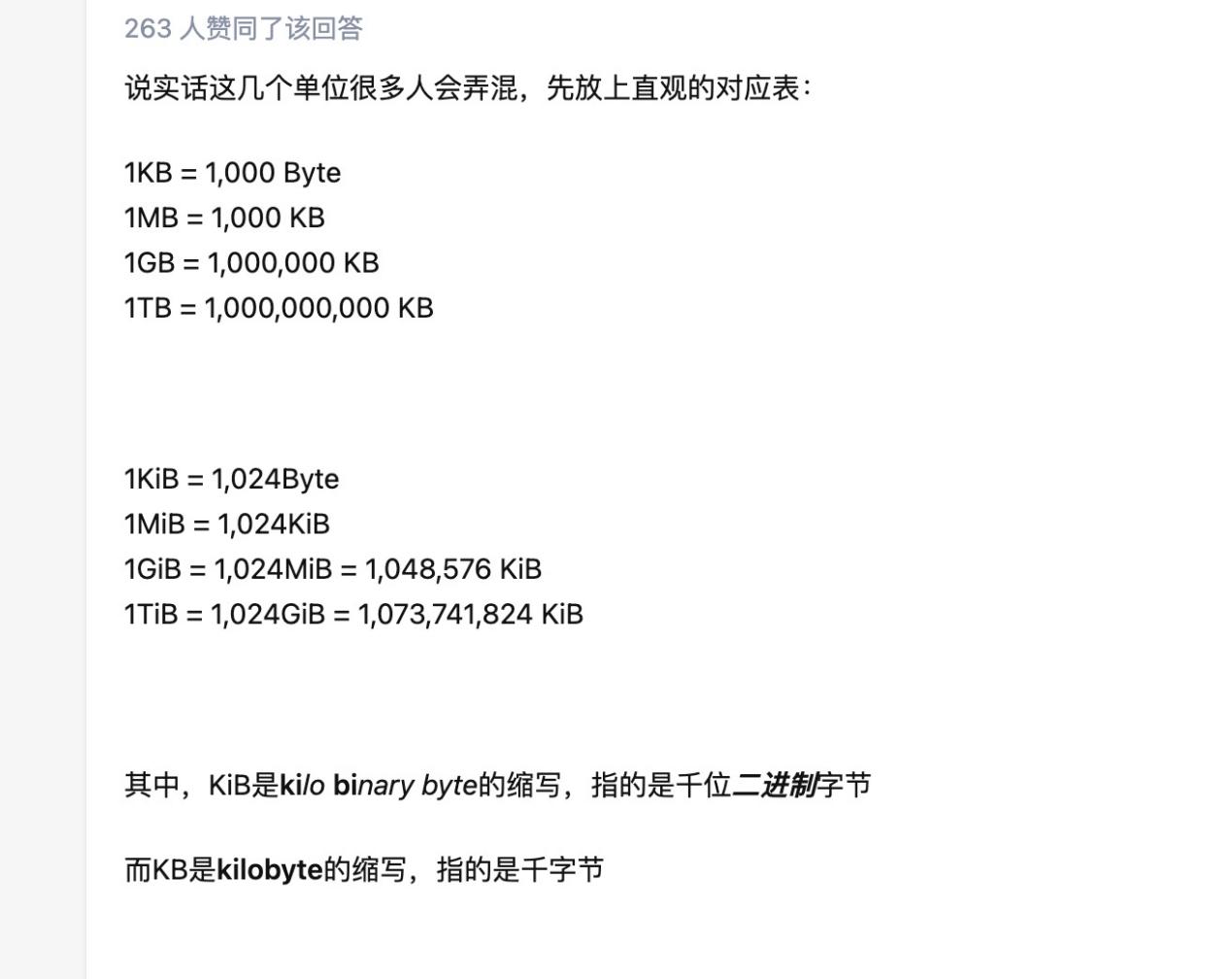
write timeout 设置的小了 你业务代码还没跑完呢, 服务端就吧链接断了, 可以理解为服务端执行业务的最长时间.

数据库网卡爆了 cpu爆了也会导致hang在conn步骤

errors.WithMessagef(err, "%s.plugin.finish", item.Name())



所以说在锁和waitgroup里面 信号量和cas操作是经常用到的.



信号量（[semaphore](https://so.csdn.net/so/search?q=semaphore&spm=1001.2101.3001.7020" \t "/Users/yangwenshuo/Documents\\x/_blank)）是操作系统用来解决并发中的互斥和同步问题的一种方法

数据模块和过程模块, 架构= 数据组件 + 过程组件

数据的封装 对数据的封装 有些东西要封装到数据里面

上面的例子中，我们定义结构体字段名首字母是小写的，这意味着这些字段在包外不可见,因而无法在其他包中被访问，只允许**包内访问**。

`{"service\_name":"%s"}` 字符串这样定义里面的双引号可以不用转义了

数组 也就是长度是固定的东西,是按照值来传递的!!!!!!!!!!!!但是切片不是

可以看到源码里面是没有什么defer释放锁的, 那样还是不太好

math.MaxFloat64

SetLogger() 和 Logger() 不要GetLogger()

var builder1 strings.Builder

builder1.WriteString("A Builder is used to efficiently build a string using Write methods.") 字符串拼接用strings.Builder

地图就是底图加路网加poi

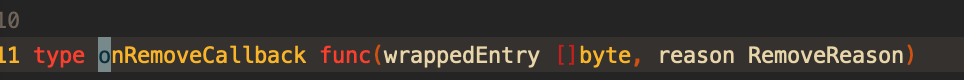
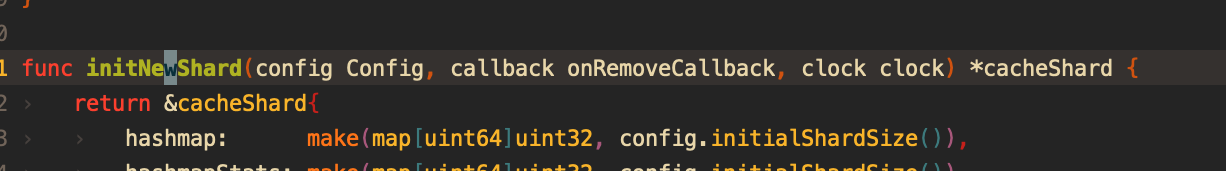
import "container/list"

这个原生包里封装了堆 前后双向链表和环

gdp那个toml文件生成就是用template去做

for finshnum < len(urls) {

这里这样写可以代替里面的判断长度然后break 这样的for是正确的方式  
 select {  
 case v := <- ch:  
 result[v.url] = v.result  
 finshnum++  
 case <- cc:  
 fmt.Println("怎么还不来啊")  
 return result  
 }  
}  
fmt.Println("chu l xunhuan")



当你想要把一个函数传进去的时候,参数不要直接写这个函数,而是要先把这个函数给type了,无论你创建参数还是函数定义都是比较好的

改过一个问题后,一定要想想还有没有其他地方要改.

尽量用>>和<<的位移操作来代替乘和除.

接收一个消息队列的数据 一定要保证幂等性 已经什么时候err 什么时候丢弃 是否主从等问题, 解决消息队列的主从可以在消息体里加一个字段就做消费时间,如果consume消费到的时间大于了消费时间就去消费.

大结构尽量用指针, 如果是结构会拷贝.

地图就是底图 加上路网 加上poi

内存逃逸就是,栈上的数据逃逸到堆上,栈退出了这个数据由于被别的线程用了,那么就会还得弄到堆上.

errors.WithMessage(err, "parse config")

errors.WithStack(err)

url编码 gbk编码 经过做为字符串类型的value, 序列化 再反序列化后,编码就乱了, 所以奥德赛那种用[]byte的方式 用base64传数据是最好的.

进程的定时任务是不靠谱的,因为进程死了,定时任务就没了.

删东西的时候要全局grep下,不然有可能有的地方用到

A, b:= 如果之前没有b,那么就是一个新建b,如果之前有了b但是没有a, 那么b就是重新赋值.

func main() {  
  
 var err error  
  
 defer func() {  
 if err != nil {  
 fmt.Println("来了")  
 }  
 }()  
  
 if err := ceshi(); err != nil {  
 fmt.Println(err)  
 }  
}  
  
func ceshi() error {  
 return errors.New("啊啊啊啊")  
}

这样是不会走到defer的

func main() {  
  
 var err error  
  
 defer func() {  
 if err != nil {  
 fmt.Println("来了")  
 }  
 }()  
  
 s, err := ceshi()  
 fmt.Println(s)  
}  
  
func ceshi() (string, error) {  
 return "aaa", errors.New("啊啊啊啊")  
}

这样是会走到defer的

func main() {  
 n := 0  
 f := func() int {  
 n += 1  
 return n  
 }  
 fmt.Println(f()) *// 别忘记括号，不加括号相当于地址*

闭包,就是匿名函数用了外面的变量 这个函数不仅是描述了执行操作,还记住了外面的变量的地址.

*//由 main 函数作为程序入口点启动*func main() {  
 x, y := 1,2  
  
 defer func(a int){  
 fmt.Println("defer x, y = ", a, y) *//y为闭包引用* }(x) *//x值拷贝 调用时传入参数* x += 100  
 y += 200  
 fmt.Println(x, y)  
}

这个的打印结果是 101 202 和 1 202

注意defer的匿名函数如果传参数的话,那么在这个defer注册的时候就已经把这拷贝了,也就是a是被当作参数传进来的,他就是1, 而y是做为闭包弄进来的,那么他就会实时感受到y的变化.

SLA：Service-Level Agreement的缩写，意思是服务等级协议

过去10秒内最慢的x%的请求的平均延迟.其中X是数字与100的差.

例如:p99 1.403 表示过去的10秒内最慢的1%请求的平均延时为1.403秒

p95 过去的10秒内最慢的5%的请求平均延时.

为什么我们换域名 前端都不用改代码,不用改域名,因为前端的那些js,html图片的资源其实就是通过域名先拉到的,那么代码里面肯定可以获取到当前域名,而app的的域名是写在自己的代码里面的,所以一旦域名改了老的app就用不了了.

10M带宽=1.25MB/s网速：

跨域 值得说的是虽然浏览器禁止用户对请求返回数据的显示和操作，但浏览器确实是去请求了，如果服务器没有做限制的话会返回数据的，在调试模式的network中可以看到返回状态为200，且可看到返回数据

客户端发送syn 自己进入synsended状态,

服务端收到后返回syn+ack 服务端进入syn-recved状态

此时就进入了半链接状态

如果这个时候客户端不回复响应,服务端还会重试,还会等超时,这样肯定会消耗服务端用于维护半链接的资源就阻塞服务端了.

浮点数用%v方式打印超过一百万会变成科学记数法.

浮点型数据的大小比较不能直接用等号,而是要用func isEqual(p1, p2, float64)

有race的代码不能要, 不然并发问题定位不到根本

go get [www.xxx.com/xxx/xxx/...](http://www.xxx.com/xxx/xxx/...)

type PoiIndex struct {

Rowkey string // table中存储的key

}

func NewPoiIndex(rowkey string) \*PoiIndex {

return &PoiIndex{Rowkey: rowkey}

}

func (p \*PoiIndex) Upsert(ctx context.Context, picId string) error {

items := []\*table.DataItem{{Name: fmt.Sprintf("%s:%s", "contain\_pic", picId), Value: ""}}

return dao.PicTableClient.Insert(ctx, p.Rowkey, items, constant.TablePoiPicPoiIndex)

}

对于hbase这种东西,是可以把参数拼到列里面的.

https的服务也是可以控制台看到明文的,因为在http层的数据已经被解码了,在具体传输的时候是加密的.

为啥bd内部起的各种服务都是http的服务,因为https的东西已经被bgw给做了.

http.HandleFunc("/hello", hello) http.ListenAndServeTLS(":8080", certPath, keyPath, nil)

还有就是mme的虚拟线程,和go的携程还是有点不一样的,因为mme的虚拟线程的运行哪个的调度完全是有我们的代码控制的,而go语言是由go的调度器控制的.

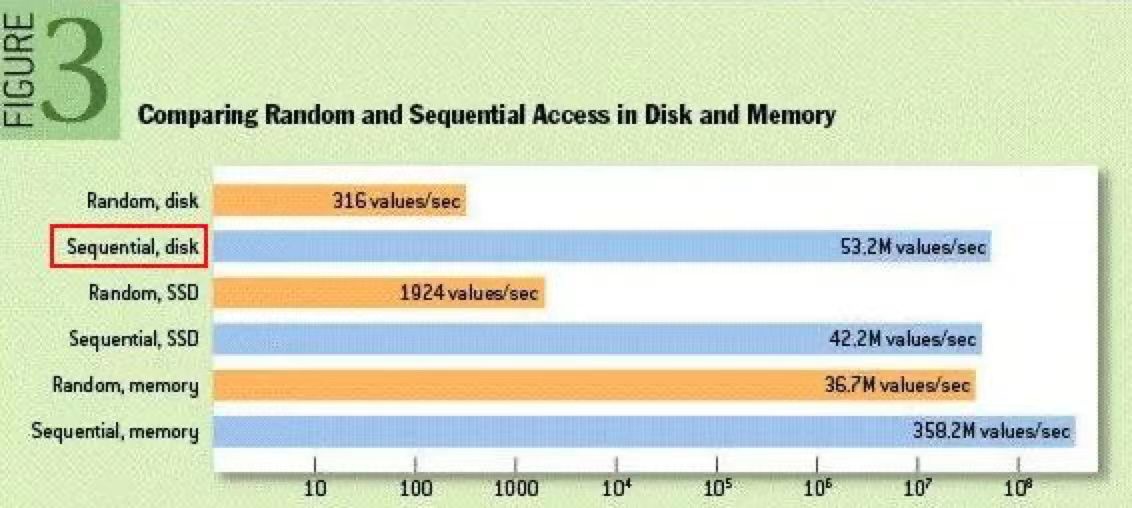


## 源码阅读

<https://github.com/cloudwego/kitex>

## 消息队列

顺序写文件的性能很高，接近于内存中的随机写，如下图示



我们给每一个消息文件定为固定的 1G 大小，如果文件满了的话再创建一个即可，我们把这些存储消息的文件集合称为 commitlog。这样的设计还有另一个好处：在删除过期文件的时候会很方便，直接把之前的文件整个删掉即可，最新的文件无需改动，而如果把所有消息都写到一个文件里，显然删除之前的过期消息会非常麻烦



即使解决了以上问题，也无法解决根据消费进度快速定位其所对应消息在文件的位置。假设 broker 重启了，然后读取消费进度（消费进度可以持久化到文件中），此时不得不从头读取文件来定位消息在文件的位置，这在效率上显然是不可接受的

那能否既能利用到数组的快速寻址，又能快速定位消费进度对应消息在文件中的位置呢，答案是可以的，我们可以新建一个索引文件（我们将其称为 consumeQueue 文件），每次写入 commitlog 文件后，都把此消息在 commitlog 文件中的 offset（我们将其称为 commit offset，8 字节） 及其大小（size，4 字节）还有一个 tag hashcode（8 字节，它的作用后文会提到）这三个字段顺序写入 consumeQueue 文件中

这样每次追加写入 consumeQueue 文件的大小就固定为 20 字节了，由于大小固定，根据数组的特性，就能迅速定位消费进度在索引文件中的位置，然后即可获取 commitlog offset 和 size，进而快速定位其在 commitlog 中消息

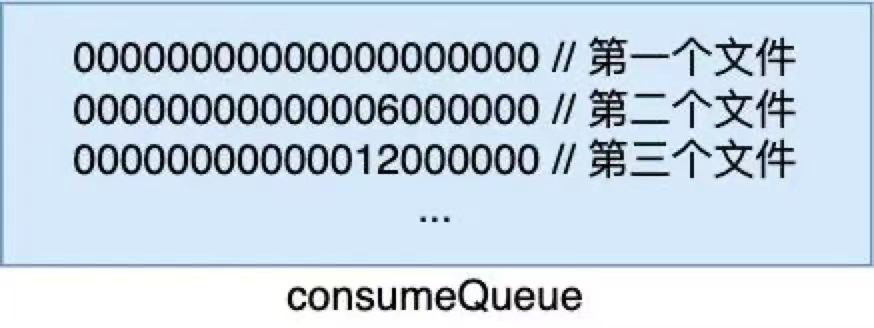
consumeQueue 文件也会写满，写满后也要新建一个文件再写入,我们规定 consumeQueue 可以保存 30w 条数据，也就是 30w \* 20 byte = 600w Byte = 5.72 M，为了便于定位消费进度是在哪个 consumeQueue文件中，每个文件的名称也是以偏移量来命名的

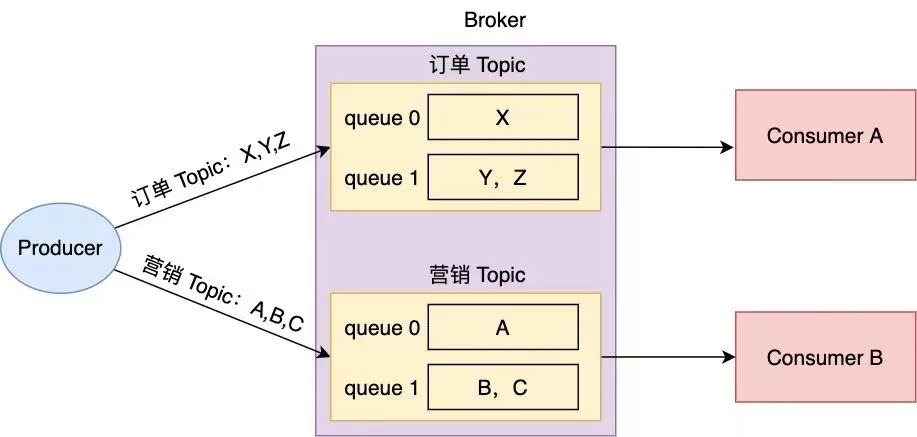
知道了文件的写入与命名规则，我们再来看下消息的写入与消费过程

消息写入：首先是消息被顺序写入 commitlog 文件中，写入后此消息在文件中的偏移（commitlog offset）和大小（size）会被顺序写入相应的 consumeQueue 文件中

消费消息：每个消费者都有一个消费进度，由于每个 consumeQueue 文件是根据偏移量来命名的，首先消费进度可根据二分查找快速定位到进度是在哪个 consumeQueue 文件，进一步定义到是在此文件的哪个位置，由此可以读取到消息的 commitlog offset 和 size，然后由于 commitlog 每个文件的命名都是按照偏移量命名的，那么根据 commitlog offset 显然可以根据二分查找快速定位到消息是在哪个 commitlog 文件，进而再获取到消息在文件中的具体位置从而读到消息

Broker 中只有一个 consumerQueue，显然没法满足集群模式下并行消费的需求，该怎么办呢，我们可以借鉴分库分表的设计理念：将数据分片存储，具体做法是创建多个 consumeQueue，然后将数据平均分配到这些 consumerQueue 中，这样的话每个 consumer 各自负责独立的 consumerQueue 即可做到并行消费





## 协程动态变化队列

import (  
 "context"  
 "fmt"  
 "sync"  
)  
  
type GPool struct {  
 ctx context.Context  
 count int32  
 reduceChan chan int32  
 wg \*sync.WaitGroup  
 f func(ctx context.Context) error  
}  
  
func NewGPool(ctx context.Context, count int32, f func(ctx context.Context) error) \*GPool {  
 return &GPool{  
 ctx: ctx,  
 count: count,  
 reduceChan: make(chan int32, 100),  
 wg: &sync.WaitGroup{},  
 f: f,  
 }  
}  
  
func (p \*GPool) Run() {  
 for i := 0; i < int(p.count); i++ {  
 p.Add()  
 }  
 p.wg.Wait()  
}  
  
func (p \*GPool) Add() {  
 var (  
 ctx = p.ctx  
 )  
 p.wg.Add(1)  
 go func() {  
 defer p.wg.Add(1)  
 for {  
 select {  
 case <-ctx.Done():  
 fmt.Println("ctx done exit")  
 return  
 case index := <-p.reduceChan:  
 fmt.Println("reduce exit", index)  
 return  
 default:  
 p.f(ctx)  
 }  
 }  
 }()  
}  
  
func (p \*GPool) Reduce(reduce int32) {  
 for i := 0; i < int(reduce); i++ {  
 p.reduceChan <- int32(i)  
 }  
}

## 语法

jsonresp有一个writeto的方法, 这个方法的参数是一个writer, 然后就可以把结构体的序列化封装在jsonresp中, 然后调用参数的writer的write方法.

这样什么protobufresp啥的都可以用了, 注意是把writer当做参数传进来.

正常的逻辑是writer去write data, 那么writer是主动的, 经常是data是参数.

这样封装一下把writer当做参数传进去, 那么writer就成参数了,就是被动的了.



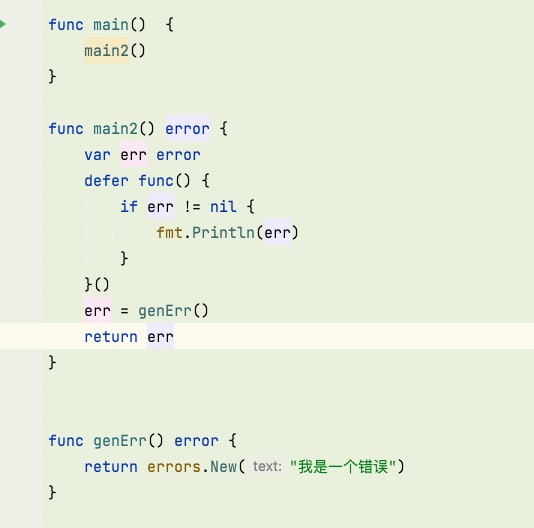
要多用继承 , 对于插件这种 他是都有name conf weight这个几个属性的,

那么这几个属性的操作可以抽象成一个基础接口的, 别的插件继承这个基础接口, before插件就这样



尽量不要搞那种大接口, 要接口之前的继承就好.

有接口的继承 拆接口的想法,或者多个接口的组合攒成这个大接口



看来必须要赋值一下, 而不能直接return.

父函数的小写资源是也会被继承到子函数的

单引号在go语言中表示golang中的rune(int32)类型，单引号里面是单个字符，对应的值为改字符的ASCII值。

func main() {

a := 'A'

fmt.Println(a) 这打印出来会是97啥的

单引号里面只能放字符

双引号表示是字符串

``这表里里面的东西不用转义了

go17后依赖变了 之前的依赖是不全的 ,你依赖的依赖在go.mod里面是看不到的, 在go.sum里面能看到,go17之后requeir会分两部分,一部分是你直接依赖的,一部分是依赖的依赖 也就是go.17后在go.mod里面能看到所有的依赖.

看了下大意是：

1. 如果逻辑需要修改Receiver则必须用\*T

2. 如果Receiver很大，使用\*T会更加cheap

3. 如果其他地方因上述原因使用了\*T，则所有Receiver都应该使用\*T

4. 如果是基础类型，使用T更为cheap

#4 这个应该类似于函数传值/地址的老问题了，一个足够小的struct可以通过栈、甚至是寄存器层面传递。

但回归最初的问题，按照文中讲的，XXXXResponse 虽然不大（但也不是寄存器层面能够解决的），并且NewXXXResponse返回的\*T，内存引用上已经属于逃逸了，这种肯定是分配在堆中。那么从NewXXXX和Receiver的类型统一性来看，都应该是使用\*T才是最佳做法吧。能否这么理解呢？

fmt.Sprintf("%v", float64(income)/100),这样超过百万会展示科学记数法的

data ,err := ioutil.ReadAll(resp.Body)

if err != nil {

panic(err)

}

但是你会发现，上面的操作方式会有一个小问题，那就是下载小文件还行，如果是大的文件的话，可能会出现内存不足的问题，因为它是需要先把请求内容全部读取到内存中，然后再写入到文件中的,制大文件也可以用 io.copy 这个，防止产生内存,它是将源复制到目标，并且是按默认的缓冲区32k循环操作的，不会将内容一次性全写入内存中,这样就能解决大文件的问题。

stream resp配合iocopy实现文件传输下载

## 协程池

github.com/panjf2000/ants

gopool

## IM

type client chan<- string *// send only channel*var (  
 entering = make(chan client)  
 leaving = make(chan client)  
 messages = make(chan string)  
)

func main() {  
 listener, err := net.Listen("tcp", "0.0.0.0:8888")  
 if err != nil {  
 log.Fatal(err)  
 }  
  
 go broadcaster()  
  
 for{  
 conn, err := listener.Accept()  
 if err != nil {  
 fmt.Fprintf(os.Stdout, "you got something wrong %v", err)  
 continue  
 }  
 go handleConn(conn)  
 }  
}

func broadcaster() {  
 clients := make(map[client]bool) *//all connected clients* for {  
 select {  
 case msg := <- messages:  
 *// Broadcast incoming message to all  
 // clients' outgoing message channels.* for cli := range clients{  
 cli <- msg  
 }  
 case cli := <- entering:  
 clients[cli] = true  
 case cli := <- leaving:  
 delete(clients, cli)  
 close(cli)  
 }  
 }  
}

im消息都是写模式,就是一个人发消息了 要把这个消息写到每个人的消息列表里.

而不是一个人上线了 现场去拉消息.微信啥的都是写模式的.

会有两个队列 一个是用户一次发消息纬度的,一个是写一个用户的消息列表纬度的.

也就是用户发一个群聊 会把消息内容, groupid 推到队列一, 队列一处理,看看队列一有那些用户, 推n个消息到队列2 队列2收到消息后就去写每个用户的消息列表

## 泛型

泛型就是 把类型当参数, 一个泛型函数, 参数是泛型,那么返回的时候也是泛型.

把类型的确定放到调用的时候, 就是你写调用代码的时候,而不是执行的时候

## 编译原理

比如进出队列的箭头 就是会被编译器转换成具体的runtime的代码 这是编译器干的

golang的源码是可以自己修改重新编译的。

抽象语法树 静态单赋值(SSA)

X86指令集 ARM指令集

编译器前端:承担着词法分析语法分析(生成抽象语法树),类型检查,中间码生成.

编译器后端:目标代码的生成和优化,翻译成中间码到机器码.

go源码包里面包含了不同指令集机器码生成所需的包,比如生成x86机器码的包, ARM机器码的包.

有了抽象语法树,那么类型检查的时候就会类似操作树一样什么left right node之类的了.

中间代码的存在是必要的,不能直接源码到机器码的.

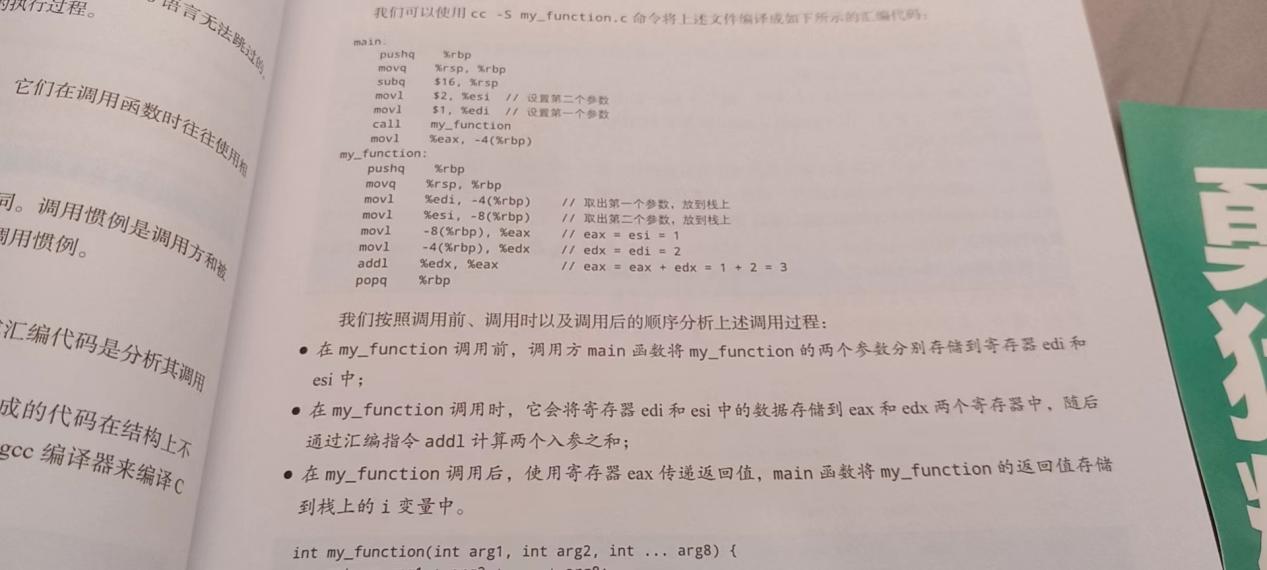
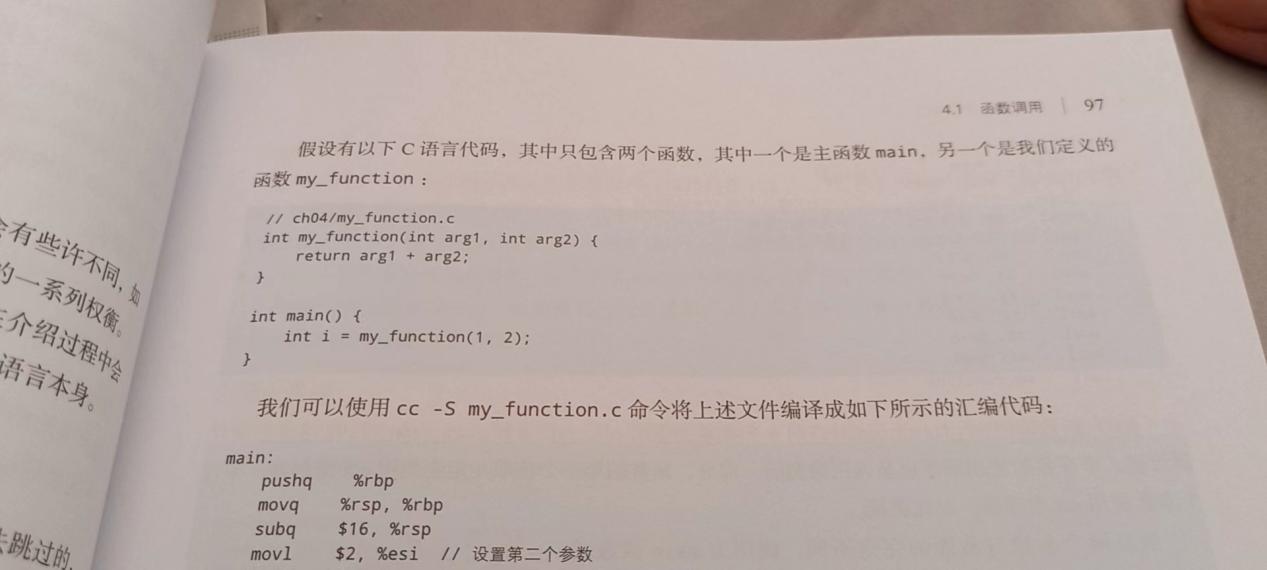
中间代码的生成过程就是抽象语法树到SSA中间代码的转换过程.

指令集是软硬件之间的桥梁,所以指令集架构是计算机的抽象模型,所以指令集架构也是计算机架构,包括x86和ARM.

X86就是复杂指令集,ARM是精简指令集.由于复杂指令集指令很多,指令长度也就不同,需要消耗额外的性能去判断指令长度,而简单指令集是用固定长度的基础指令组合使用,所以更好,性能更好.

中间代码会转成汇编代码, 汇编代码经过汇编器就会转成机器码

静态语法树 - SSA中间代码 - 汇编代码 - 机器码



可以看到上面的汇编,确实是值传递,里面的函数会复制一份外面传进来的参数使用.

是这样的,外面的1和2的实参,是main函数放到寄存器中的,myfunc用的时候会从寄存器里面复制数据放到自己的栈里,所以是值传递.

最后在用eax寄存器传递返回值,这个寄存器是专门用来传递返回值的,只有一个所以c语言只有一个返回值.

看来寄存器是有个数限制的,而且都有自己独特的名字.

c语言和go语言在设计函数调用选择不同的.

C语言使用寄存器和栈传递参数(如果参数数量小于6个全用的寄存器)

go使用栈传递参数和返回值.

c语言这样更快,cpu访问寄存器的速度比访问栈快几十倍.

go这样比较简单,不需要适配不同的寄存器,可以返回多个值,因为返回值也是在栈上,栈上多开空间不就可以返回多个了,但是寄存器却只有几个,就不能返回多个参数了.

for循环的实现基本都是用到了汇编的

JUMP命令跳转(跳到循环开始的地方) INCQ命令++ CMPQ命令比较.

## Chan

无缓冲 channel 的发送动作一直要到有一个接收者接收这个值才算完成，否则都是阻塞着的，也就是说，发送的数据需要被读取后，发送才会完成

带缓存的 channel 实际上是一个阻塞队列。队列满时写协程会阻塞，队列空时读协程阻塞。



一个chan同时只能被一个协程读或者写。

1. func makechan(t \*chantype, size int) \*hchan {

2. var c \*hchan

3. c = new(hchan)

4. c.buf = malloc(元素类型大小\*size)

5. c.elemsize = 元素类型大小

6. c.elemtype = 元素类型

7. c.dataqsiz = size

8.

9. return c

10. }

可以看出chan里面是有两个队列的，一个存的是想读的阻塞在这里的协程list,还有一个是想写的阻塞在这里的协程list, 这俩会根据情况来唤醒调度。

就是一个阻塞消息队列.

可以有多个consumer和多个producer

channel就是个带锁的阻塞队列 有锁 有size count 游标,有等待send携程列表 有等待recv携程列表

## Slice

7. func main() {

8. var array [10]int

9.

10. var slice = array[5:6]

11.

12. fmt.Println("lenth of slice: ", len(slice))

13. fmt.Println("capacity of slice: ", cap(slice))

14. fmt.Println(&slice[0] == &array[5])

15. }

这个len是1， 但是cap是5，后面的比较是true

注意，第二个是cap是5， 这个cap就是从起始位置，到真正的空间的结尾的位置。

append如果扩容了地址才会变的。

使用append向Slice追加元素时，如果Slice空间不足，将会触发Slice扩容，扩容实际上重新一配一块更大的内存，将原Slice数据拷贝进新Slice，然后返回新Slice，扩容后再将数据追加进去。

slice := array[start:end]

1. type slice struct {

2. array unsafe.Pointer

3. len int

4. cap int

5. }

使用make来创建Slice时，可以同时指定长度和容量，创建时底层会分配一个数组，数组的长度即容量。

那种通过下标创造的切片是一个新的切片,他也有自己的len cap属性,但是data还是指向原来的切片,所以原来的改了他也会改.

切片的扩容也会用到memove

## Map

map的实现是hash表。

1. type hmap struct {

2. count int // 当前保存的元素个数

3. ...

4. B uint8 // 指示bucket数组的大小

5. ...

6. buckets unsafe.Pointer // bucket数组指针，数组的大小为2^B

7. ...

8. }

buckets就是个bucket的数组，所以这里也是用的unsafe的指针。

1. type bmap struct {

2. tophash [8]uint8 //存储哈希值的高8位

3. data byte[1] //key value数据:key/key/key/.../value/value/value...

4. overflow \*bmap //溢出bucket的地址

5. }

这是其中一个的bucket的结构

可以看到，一个bucket并不是只存一个k-v，而是存8个，后面也是bucket做的拉链表。

注意放kv的时候也是key和value分开放，这样可以防止字节对不齐。

负载因子 = 键数量/bucket数量

哈希表需要将负载因子控制在合适的大小，超过其阀值需要进行rehash，也即键值对重新组织：

哈希因子过小，说明空间利用率低

哈希因子过大，说明冲突严重，存取效率低

每个哈希表的实现对负载因子容忍程度不同，比如Redis实现中负载因子大于1时就会触发rehash，而Go则在在负载因子达到6.5时才会触发rehash，因为Redis的每个bucket只能存1个键值对，而Go的bucket可能存8个键值对，所以Go可以容忍更高的负载子。

当负载因子过大时，就新建一个bucket，新的bucket长度是原来的2倍，然后旧bucket数据搬迁到新的bucket。

考虑到如果map存储了数以亿计的key-value，一次性搬迁将会造成比较大的延时，Go采用逐步搬迁策略，即每次访问map时都会触发一次搬迁，每次搬迁2个键值对。

如果查找不到，也不会返回空值，而是返回相应类型的0值。

如果当前处于搬迁过程，则优先从oldbuckets查找

## String

1. type stringStruct struct {

2. str unsafe.Pointer

3. len int

4. }

string数据结构跟切片有些类似，只不过切片还有一个表示容量的成员，事实上string和切片，准确的说是byte切片经常发生转换。

字符串构建过程是先跟据字符串构建stringStruct，再转换成string

string在runtime包中就是stringStruct，对外呈现叫做string。

1. func gostringnocopy(str \*byte) string { // 跟据字符串地址构建string

2. ss := stringStruct{str: unsafe.Pointer(str), len: findnull(str)} // 先构造stringStruct

3. s := \*(\*string)(unsafe.Pointer(&ss)) // 再将stringStruct转换成string

4. return s

5. }

这个unsafe.pointer是可以转换成各种类型的指针的。

字符串和[]byte 互相转换其实都是内存拷贝的。

频繁的字符串+拼接这样效率是很低的，如果要是正经拼接的话的是strings.builder。

像C++语言中的string，其本身拥有内存空间，修改string是支持的。但Go的实现中，string不包含内存空间，只有一个内存的指针，这样做的好处是string变得非常轻量，可以很方便的进行传递而不用担心内存拷贝。

因为string通常指向字符串字面量，而字符串字面量存储位置是只读段，而不是堆或栈上，所以才有了string不可修改的约定。

字符串的结构和切片的区别就是字符串没有cap字段.字符串的拼接也是需要用到copy函数的,而且新拼出来的字符串是个新的空间,不是追加, 所以不挺的拼接,那么就会不停copy越来越大的空间性能就会下降.

字符串是不可写的,转成[]byte是可写的,其实这个转换也是copy了一份,如果很大的话也是会损耗性能的.

byte切片转换成string的场景很多，为了性能上的考虑，有时候只是临时需要字符串的场景下，byte切片转换成string时并不会拷贝内存，而是直接返回一个string，这个string的指针(string.str)指向切片的内存。

比如，编译器会识别如下临时场景：

使用m[string(b)]来查找map（map是string为key，临时把切片b转成string）；

字符串拼接，如”<” + “string(b)” + “>”；

字符串比较：string(b) == “foo”

因为是临时把byte切片转换成string，也就避免了因byte切片同容改成而导致string引用失败的情况

## Defer

defer语句用于延迟函数的调用，每次defer都会把一个函数压入栈中，函数返回前再把延迟的函数取出并执行。 为了方便描述，我们把创建defer的函数称为主函数，defer语句后面的函数称为延迟函数。

1. func deferFuncParameter() {

2. var aInt = 1

3.

4. defer fmt.Println(aInt)

5.

6. aInt = 2

7. return

8. }

这个输出的是1，因为这个是走的参数，我们以前的那种走的是闭包defer引用了外面的变量，而不是参数传进来的。

1. func deferFuncReturn() (result int) {

2. i := 1

3.

4. defer func() {

5. result++

6. }()

7.

8. return i

9. }

这个最终会返回2.

return不是原子的，会先把i赋值给result，然后在执行defer里面的内容。

注意是这个流程。

这种函数带有具名返回值的就是这个流程。

1. func foo() int {

2. var i int

3.

4. defer func() {

5. i++

6. }()

7.

8. return i

9. }

注意这里最后是0，

流程还是一样的，先把i赋个一个匿名的result，defer又去把i给++了，最后返回的是匿名result，是这个流程的。

1. type \_defer struct {

2. sp uintptr //函数栈指针

3. pc uintptr //程序计数器

4. fn \*funcval //函数地址

5. link \*\_defer //指向自身结构的指针，用于链接多个defer

6. }

这是一个defer的链表

deferproc()： 在声明defer处调用，其将defer函数存入goroutine的链表中；

deferreturn()：在return指令，准确的讲是在ret指令前调用，其将defer从goroutine链表中取出并执行。

## Select

select {

case messages <- msg:

fmt.Println("sent message", msg)

default:

fmt.Println("no message sent")

}

意思是这个default 可以让一个channel seleft本来是阻塞的,现在有default 就不会阻塞等待了



也不一定非得for里面

select是Golang在语言层面提供的多路IO复用的机制

## Benchmark

<https://www.cnblogs.com/yahuian/p/go-benchmark.html>

## Test

func TestAll(t \*testing.T) {

t.Run("testPrint1", testPrint1)

t.Run("testPrint2", testPrint2)

}

func TestMain(m \*testing.M) {

fmt.Println("测试初始化开始.....")

m.Run()

}

TestMain就是每个test执行前执行的东西

## 并发

锁那些玩意本来就肯定会用到信号量的啊

1. type Mutex struct {

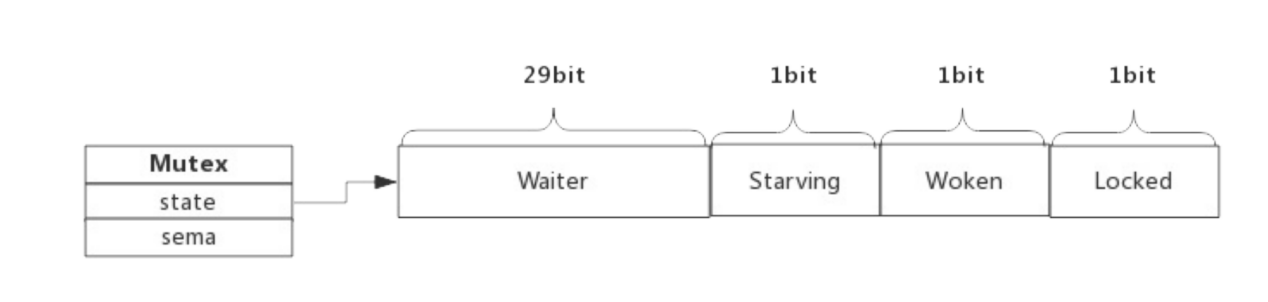
2. state int32

3. sema uint32

4. }

Mutex.state表示互斥锁的状态，比如是否被锁定等。

Mutex.sema表示信号量，协程阻塞等待该信号量，解锁的协程释放信号量从而唤醒等待信号量的协程



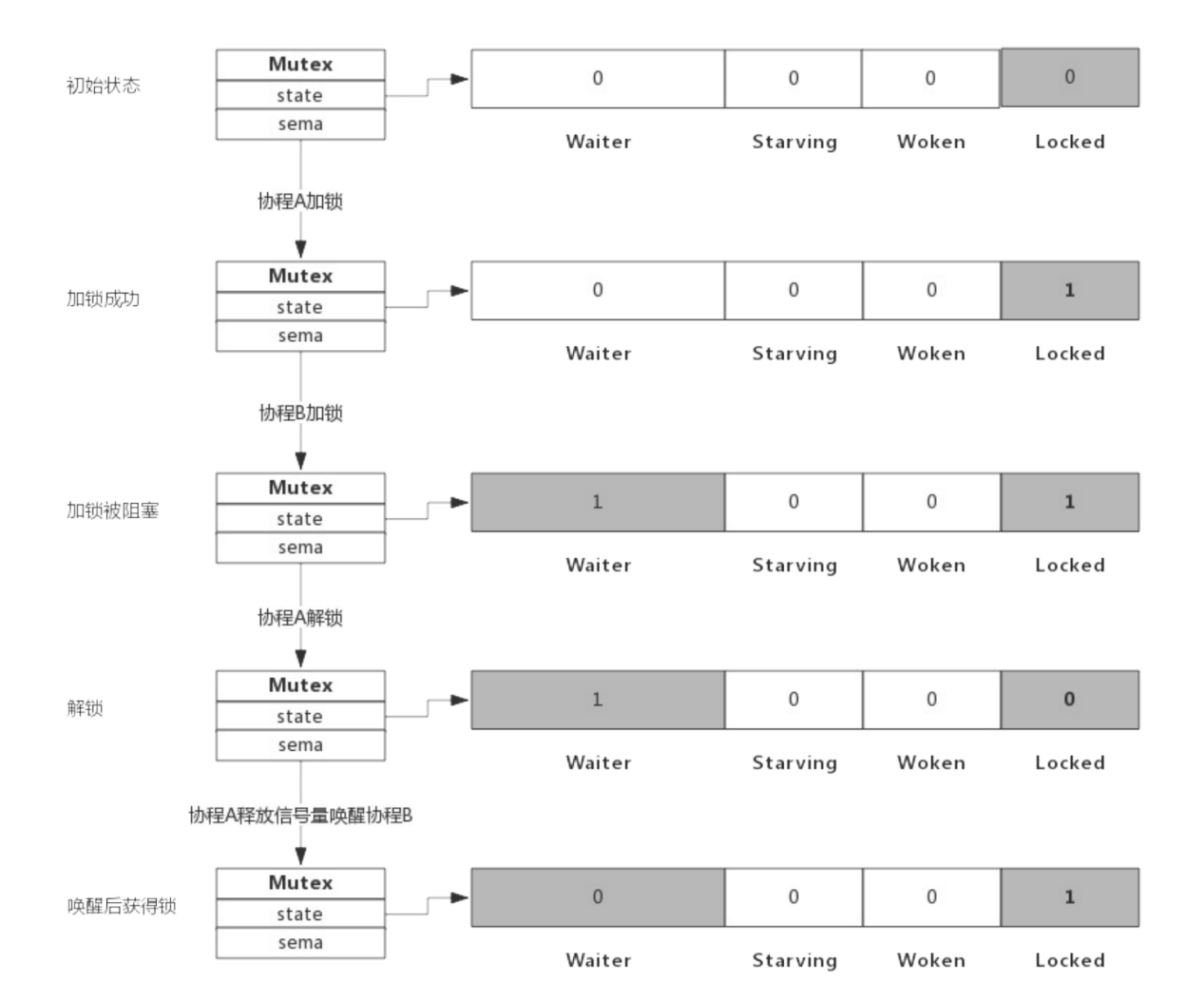
Locked: 表示该Mutex是否已被锁定，0：没有锁定 1：已被锁定。

Woken: 表示是否有协程已被唤醒，0：没有协程唤醒 1：已有协程唤醒，正在加锁过程中。

Starving：表示该Mutex是否处理饥饿状态， 0：没有饥饿 1：饥饿状态，说明有协程阻塞了超过1ms。 这里就可以统计有多少饥饿的协程。

Waiter: 表示阻塞等待锁的协程个数，协程解锁时根据此值来判断是否需要释放信号量。

协程之间抢锁实际上是抢给Locked赋值的权利，能给Locked域置1，就说明抢锁成功。抢不到的话就阻塞等待Mutex.sema信号量，一旦持有锁的协程解锁，等待的协程会依次被唤醒。



自旋对应于CPU的”PAUSE”指令，CPU对该指令什么都不做，相当于CPU空转，对程序而言相当于sleep了一小段时间，时间非常短，当前实现是30个时钟周期。

自旋过程中会持续探测Locked是否变为0，连续两次探测间隔就是执行这些PAUSE指令，它不同于sleep，不需要将协程转为睡眠状态。

Mutx lock方法

先是cas操作看看能不能直接锁上,不能的话就自旋,再看能不能用cas锁上.如果还不能就通过信号量的方获取锁了.

Unlock 先看能不能cas操作释放锁,不能的话就信号量.

所以golang也是有自旋锁的.

waitgroup也是用到了信号量和cas操作的

1. type RWMutex struct {

2. w Mutex //用于控制多个写锁，获得写锁首先要获取该锁，如果有一个写锁在进行，那么再到来的写锁将会阻塞于此

3. writerSem uint32 //写阻塞等待的信号量，最后一个读者释放锁时会释放信号量

4. readerSem uint32 //读阻塞的协程等待的信号量，持有写锁的协程释放锁后会释放信号量

5. readerCount int32 //记录读者个数

6. readerWait int32 //记录写阻塞时读者个数

7. }

可以看到，里面还是有一个mutex的。

读写锁不用担心读qps很高,导致写操作无法执行, 他肯定是会排队的,先来的写会先拿到锁的.

syncmap就是高级的读写锁, 里面用cas操作 用ditrymap等来优化,是比你用读写锁更好的.

sync.Pool 本质用途是增加临时对象的重用率，减少 GC 负担；

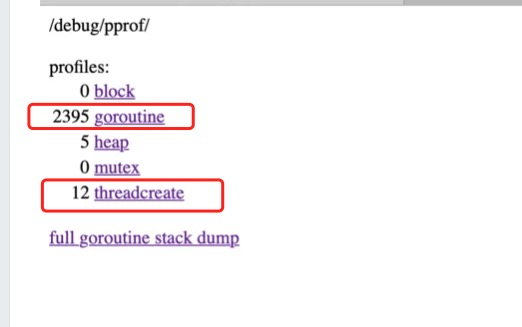
不能对 Pool.Get 出来的对象做预判，有可能是新的（新分配的），有可能是旧的（之前人用过，然后 Put 进去的）；

不能对 Pool 池里的元素个数做假定，你不能够；

sync.Pool 本身的 Get, Put 调用是并发安全的，sync.New 指向的初始化函数会并发调用，里面安不安全只有自己知道；

当用完一个从 Pool 取出的实例时候，一定要记得调用 Put，

## 调度器

可以看到 200个协程差不多对一个线程

线程最少占1m 协程几k

线程切换1us 协程切换0.2us 省80%

G有非常多个

M默认最大1000个可以通SETMAXTHREAD来设置

P就是你设置的GOMAXPROCS的数量.

一个P下面挂这他的runqueue,也就是G的队列.

他会绑定M去执行G, 如果一个M执行G阻塞, 那么P就会绑定别的M去执行G.

什么时候会触发P的调度行为呢?

1,系统调用 当你在 goroutine 进行一些 sleep 休眠、读取磁盘或者发送网络请求时，其实都会发生系统调用，进入操作系统内核.而一旦发生系统调用,就会直接触发runtime的调度，当前的P就会去找其他的M进行绑定，并取出 G 开始运行。

2,此外，在你的代码中，若因为锁或者通道导致代码阻塞了，也会触发调度。

3,在代码中直接调用 runtime.Gosched 方法，也可以手动触发。

调度器进化史

单线程调度器:就是有一个线程m,他会先执行g一会,然后保存栈寄存器和程序计数器,然后选择下一个g, 把执行器切到这个g上,再执行一会,整体就一个m.

多进程调度器:就是有多个m,每个m都是会类似于单进程那样,可以理解为有个g的队列,多个m消费这一个队列,就会需要给调度器加锁, 那么就会多个m竞争一个锁,性能不好.这个类似与多个M和G,但是只有一个P.从这个时候引用了GOMAXRPCS变量来帮我们控制m的数量.

任务窃取的调度器: 为每个M都加了一个P,这样就避免了全局锁.其次就是P发现自己的M上没有G了就回去窃取一个G.

以上的调度器都不是抢占调度器,都是需要G自己主动去让出执行权,才会触发后面的调度,如果一个G很自私,一直不让,那么其他G就会饥饿.

抢占式的就是如果一个G一直不放手,就会把他的执行权限给抢了,不然其他G就会饿死.

基于协作的抢占调度器:

如果 sysmon 监控线程发现有个协程 A 执行之间太长了（或者 gc 场景(可以看到gc太长也是会被抢占的)，或者 stw 场景），那么会友好的在这个 A 协程的某个字段设置一个抢占标记 ；

协程 A 在 call 一个函数的时候，会复用到扩容栈（morestack）的部分逻辑，检查到抢占标记之后，让出 cpu，切到调度主协程里；

这个有个问题就是,需要A去call函数的时候才会检查抢占标记,把cpu切出.这样的话比如A是个死循环,从来不call函数,那么他就永远不会调度了.



比如这个就是永远不会打印I got scheduled

基于信号的抢占调度

主要原理是 Go 程序在启动时，会在 runtime.sighandler 方法注册并且绑定 SIGURG 信号，绑定相应的 runtime.doSigPreempt 抢占方法。

同时在调度的 runtime.sysmon 方法会调用 retake 方法处理一下两种场景：

抢占阻塞在系统调用上的 P。

抢占运行时间过长的 G。

该方法会检测符合场景的 P，当满足上述两个场景之一时，就会发送信号给 M。M 收到信号后将会休眠正在阻塞的 Goroutine，调用绑定的信号方法，并进行重新调度。以此来解决这个问题。

注：在 Go 语言中，sysmon 会用于检测抢占。sysmon 是 Go 的 Runtime 的系统检测器，sysmon 可进行 forcegc、netpoll、retake 等一系列骚操作

这个的优点就是 给M发信号,M就会执行休眠操作, 而不用等

这个一个G执行时间不会超过20ms

g的结构很复杂, 有执行相关的栈信息,M的信息, 程序计数器, 状态, 标记抢占的, defer panic相关的.

默认最多能创建10000个m. linux系统一个线程是8m的栈内存 10000个m已经80多G

会创建GOMAXPROC个p, 这个基本和分配的核数差不多.

为什么要有p 1是减少那个大锁

2是可以减少内存

在还没有 P 组件的版本中，每个 M 结构体都有一个 mcache 字段，mcache 是一个内存分配池，小对象会直接从 mcache 中进行分配，M 在运行 G 时，G 需要申请小对象时会直接从 M 的 mcache 中进行分配，G 可以进行无锁访问，因为每个 M 同一时间只会运行一个 G，但 runtime 中每个时间只会有一部分活跃的 M 在运行 G，其他因系统调用等阻塞的 M 其实不需要 mcache 的，这部分 mcache 是被浪费的，每个 M 的 mcache 大概有 2M 大小的可用内存，当有上千个处于阻塞状态的 M 时，会有大量的内存被消耗。此外还有较差的数据局部性问题，这是指 M 在运行 G 时对 G 所需要的小对象进行了缓存，后面 G 如果再次调度到同一个 M 时那么可以加速访问，但在实际场景中 G 调度到同一个 M 的概率不高，所以数据局部性不太好。  
在引入了 P 组件后，mcache 从 M 转移到了 P ，P 保存了 mcache 也就意味着不必为每一个 M 都分配 一个 mcache，避免了过多的内存消耗。这样在高并发状态下，每个 G 只有在运行的时候才会使用到内存， 而每个 G 会绑定一个 P，所以只有当前运行的 G 只会占用一个 mcache，对于 mcache 的数量就是 P 的数 量，同时并发访问时也不会产生锁。

M

1、Golang 程序在启动时会创建主线程，主线程是第一个 M 即 M0；

2、当有新的 G 创建或者有 G 从 \_Gwaiting 进入 \_Grunning 且还有空闲的P，此时会调用 startm()，首先从全局队列（ sched.midle）获取一个 M 和空闲的 P 绑定执行 G，如果没有空闲的 M 则会通过newm() 创建 M；

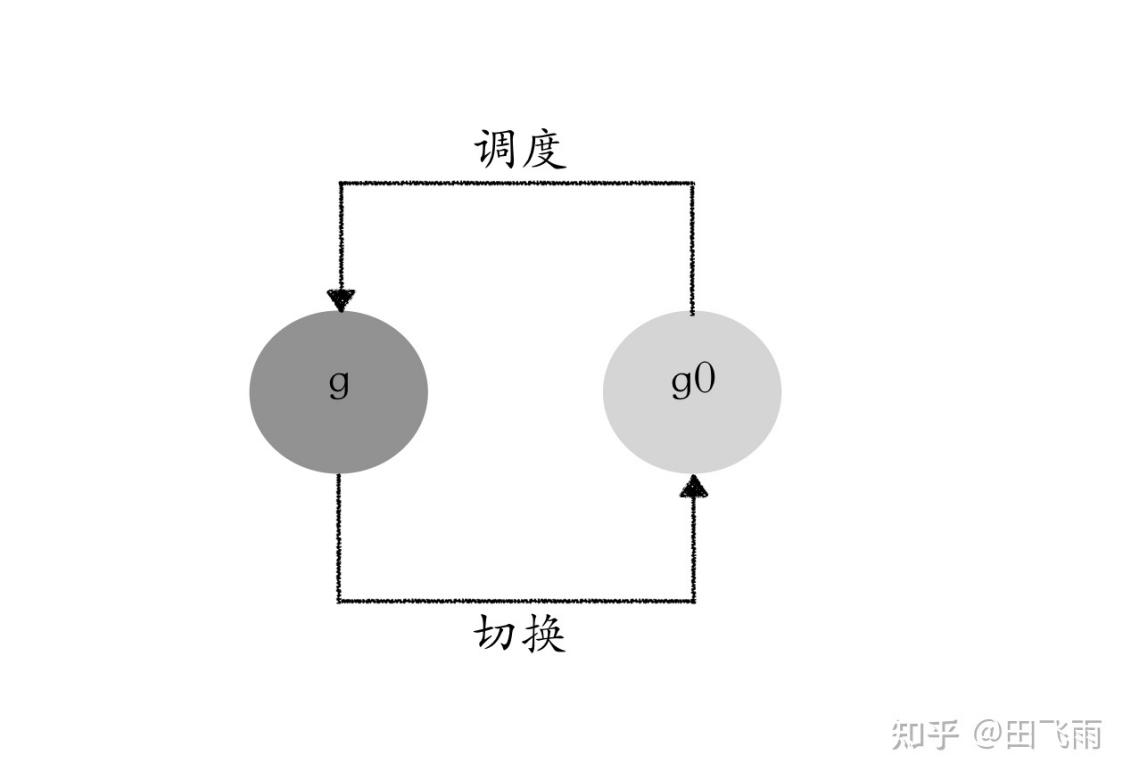
所以说startm是会先去检查m队列有没有空的, m也是有个队列的.

有空闲的p 就说明m少了 不然p会被m们抢光的.所以说m的数量是根据p的情况来的. 如果p都闲着那么m就会创建的.

如果p都忙着, 那么你的m再创建多少个也是不行的.

m有个curg 和 g0

curg就是用户态的g,要执行的g, g0是个特殊的g, 执行调度代码的时候会执行g0,包括g的创建, 内存的分配为 scheduler 执行调度循环提供了场地（栈）。对于一个线程来说，g0 总是它第一个创建的 goroutine。每个 M 都会有自己的 g0，g0 主要用来记录工作线程使用的栈信息，仅用于负责调度，在执行调度代码时需要使用这个栈。执行用户 goroutine 代码时，使用用户 goroutine 的栈，调度时会发生栈的切换。



全局队列：存放等待运行的 G。

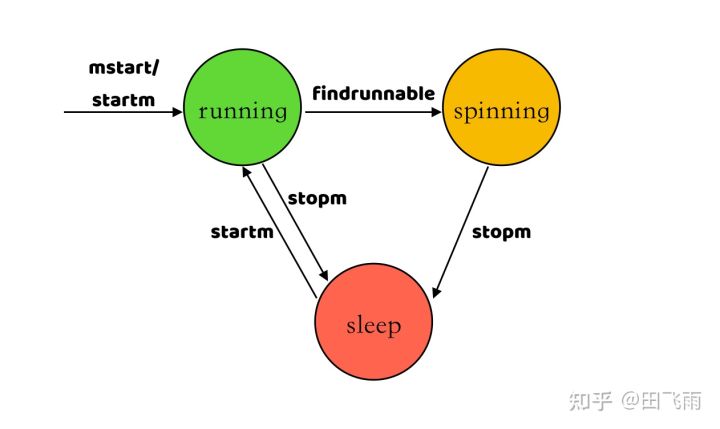
P 的本地队列：同全局队列类似，存放的也是等待运行的 G，存的数量有限。新建 G 时，G 优先加入到 P 的本地队列，如果队列满了，则会把本地队列中一部分 G 移动到全局队列。

P 列表：所有的 P 都在程序启动时创建，并保存在数组中，最多有 GOMAXPROCS(可配置) 个。

M：线程想运行任务就得获取 P，然后从 P 的本地队列获取 G，P 队列为空时，M 也会尝试从全局队列拿一批 G 放到 P 的本地队列，或从其他 P 的本地队列偷一半放到自己 P 的本地队列，M 运行 G，G 执行之后，M 会从 P 获取下一个 G，不断重复下去。

M的上面的这些逻辑 都是通过g0来控制的 也是runtime.schedule函数的逻辑.

Runtime.schedule里面会有个findrunable函数 , 在下图里面也可以看到, findrunable会去本地队列, 全局队列, 任务窃取去找G, 也***会去调用netpoll函数(也就是epoll\_wait)函数去找io已经好了的g来调度 这就和IO关联上了!!!!!!!!!!!!!!!!!!!!***



有个runtime.schdule的函数 就是m的g0来执行的.

M 需要与 P 关联才能运行，并且 M 与 P 有亲和性，比如在执行 entersyscall() 函数进入系统调用时，M 会主动与当前的 P 解绑，M 会将当前的 P 记录到 m.oldp 中，在执行 exitsyscall() 函数退出系统调用时，M 会优先绑定 m.oldp 中的 P。

M0 是一个全局变量，在 src/runtime/proc.go 定义，M0 不需要在堆上分配内存，其他 M 都是通过 new(m) 创建出来的对象，其内存是从堆上进行分配的，M0 负责执行初始化操作和启动第一个 G，Golang 程序启动时会首先启动 M0，M0 和主线程进行了绑定，当 M0 启动第一个 G 即 main goroutine 后功能就和其他的 M 一样了 。

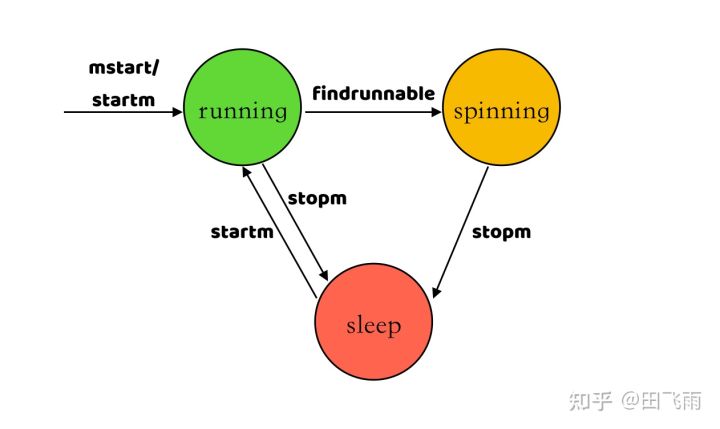
M 不会被销毁，当找不到要运行的 G 或者绑定不到空闲的 P 时，会通过执行 stopm() 函数进入到睡眠状态，在以下两种情况下会执行 stopm() 函数进入到睡眠状态：

1、当 M 绑定的 P 无可运行的 G 且无法从其它 P 窃取可运行的 G 时 M 会尝试先进入自旋状态 (spinning) ，只有部分 M 会进入自旋状态，处于自旋状态的 M 数量最多为非空闲状态的 P 数量的一半（sched.nmspinning < (procs- sched.npidle)/2），自旋状态的 M 会从其他 P 窃取可执行的 G，如果 M 在自旋状态未窃取到 G 或者未进入到自旋状态则会直接进入到睡眠转态；

2、当 M 关联的 G 进入系统调用时，M 会主动和关联的 P 解绑 ，当 M 关联的 G 执行 exitsyscall() 函数退出系统调用时，M 会找一个空闲的 P 进行绑定，如果找不到空闲的 P 此时 M 会调用 stopm() 进入到睡眠状态；

在 stopm() 函数中会将睡眠的 M 放到全局空闲队列（sched.midle）中。

这个m的状态迁移 的每个箭头的文字挺好的.



新建的 G 会通过 runqput() 函数优先被放入到当前 G(也就是ag新建了bg, 那么bg就会挂在ag所在的p是上) 关联 P 的 runnext 队列中，P 的 runnext 队列中只会保存一个 G，如果 runnext 队列中已经有 G，会用新建的 G 将其替换掉，然后将 runnext 中原来的 G 放到 P 的本地队列即 runq 中，如果 P 的本地队列满了，则将 P 本地队列一半的 G 移动到全局队列 sched.runq 中。此处将新建的 G 首先移动到 P 的 runnext 中主要是为了提高性能，runnext 是 P 完全私有的队列，如果将 G 放在 P 本地队列 runq 中， runq 队列中的 G 可能因其他 M 的窃取发生了变化，每一次从 P 本地队列获取 G 时都需要执行 atomic.LoadAcq 和 atomic.CasRel 原子操作，这会带来额外的开销。

G 与 M 绑定才能运行，而 M 需要与 P 绑定才能运行，所以理论上同一时间运行 G 的数量等于 P 的数量，M 不保留 G 的状态，G 会将状态保留在G的 gobuf 字段，因此 G 可以跨 M 进行调度。M 在找到需要运行的 G 后，会通过汇编函数 gogo() 从 g0 栈切换到用户 G 的栈运行。

P

type p struct {

id int32

status uint32 // P 的状态

link puintptr

schedtick uint32 // 被调度次数

syscalltick uint32 // 执行过系统调用的次数

sysmontick sysmontick // sysmon 最近一次运行的时间

m muintptr // P 关联的 M

mcache \*mcache // 小对象缓存,可以无锁访问

pcache pageCache // 页缓存,可以无锁访问

raceprocctx uintptr // race相关

// 与 defer 相关

deferpool [5][]\*\_defer

deferpoolbuf [5][32]\*\_defer

// goroutine ids 的缓存

goidcache uint64

goidcacheend uint64

// P 本地 G 队列,可以无锁访问

runqhead uint32 // 本地队列头

runqtail uint32 // 本地队尾

runq [256]guintptr // 本地 G 队列，使用数组实现的循环队列

runnext guintptr // 待运行的 G，优先级高于 runq

// 已运行结束的 G (状态为 Gdead)会被保存在 gFree 中，方便实现对 G 的复用

gFree struct {

gList

n int32

}

sudogcache []\*sudog

sudogbuf [128]\*sudog

mspancache struct {

len int

buf [128]\*mspan

}

tracebuf traceBufPtr

traceSweep bool

traceSwept, traceReclaimed uintptr

palloc persistentAlloc

\_ uint32

timer0When uint64

timerModifiedEarliest uint64

// 与 GC 相关的

gcAssistTime int64

gcFractionalMarkTime int64

gcMarkWorkerMode gcMarkWorkerMode

gcMarkWorkerStartTime int64

gcw gcWork

wbBuf wbBuf

......

// 抢占标记

preempt bool

}

M

type m struct {

// g0 记录工作线程（也就是内核线程）使用的栈信息，在执行调度代码时需要使用

g0 \*g

morebuf gobuf // 堆栈扩容使用

......

gsignal \*g // 用于信号处理

......

// 通过 tls (线程本地存储)结构体实现 m 与工作线程的绑定

tls [tlsSlots]uintptr

mstartfn func() // 表示m启动时立即执行的函数

curg \*g // 指向正在运行的 goroutine 对象

caughtsig guintptr

p puintptr // 当前 m 绑定的 P

nextp puintptr // 下次运行时的P

oldp puintptr // 在执行系统调用之前绑定的P

id int64 // m 的唯一id

mallocing int32

throwing int32

preemptoff string // 是否要保持 curg 始终在这个 m 上运行

locks int32

dying int32

profilehz int32

spinning bool // 为 true 时表示当前 m 处于自旋状态，正在从其他线程偷工作

blocked bool // m 正阻塞在 note 上

newSigstack bool

printlock int8

incgo bool // 是否在执行 cgo 调用

freeWait uint32

fastrand [2]uint32

needextram bool

traceback uint8

// cgo 调用计数

ncgocall uint64

ncgo int32

cgoCallersUse uint32

cgoCallers \*cgoCallers

// 没有 goroutine 需要运行时，工作线程睡眠在这个 park 成员上，

// 其它线程通过这个 park 唤醒该工作线程

doesPark bool

park note

alllink \*m // 记录所有工作线程的链表

......

startingtrace bool

syscalltick uint32 // 执行过系统调用的次数

freelink \*m

......

preemptGen uint32 // 完成的抢占信号数量

......

}

G

type g struct {

// 当前 Goroutine 的栈内存范围

stack stack

stackguard0 uintptr

stackguard1 uintptr

\_panic \*\_panic // 当前 g 中与 panic 相关的处理

\_defer \*\_defer // 当前 g 中与 defer 相关的处理

m \*m // 绑定的 m

// 存储当前 Goroutine 调度相关的数据，上下方切换时会把当前信息保存到这里

sched gobuf

......

param unsafe.Pointer // 唤醒G时传入的参数

atomicstatus uint32 // 当前 G 的状态

stackLock uint32

goid int64 // 当前 G 的 ID

schedlink guintptr

waitsince int64 // G 阻塞时长

waitreason waitReason // 阻塞原因

// 抢占标记

preempt bool

preemptStop bool

preemptShrink bool

asyncSafePoint bool

paniconfault bool

gcscandone bool

throwsplit bool

// 表示是否有未加锁定的channel指向到了g 栈

activeStackChans bool

// 表示g 是放在chansend 还是 chanrecv，用于栈的收缩

parkingOnChan uint8

raceignore int8 // ignore race detection events

sysblocktraced bool // StartTrace has emitted EvGoInSyscall about this goroutine

tracking bool // whether we're tracking this G for sched latency statistics

trackingSeq uint8 // used to decide whether to track this G

runnableStamp int64 // timestamp of when the G last became runnable, only used when tracking

runnableTime int64 // the amount of time spent runnable, cleared when running, only used when tracking

sysexitticks int64 // cputicks when syscall has returned (for tracing)

traceseq uint64 // trace event sequencer

tracelastp puintptr // last P emitted an event for this goroutine

lockedm muintptr

sig uint32

writebuf []byte

sigcode0 uintptr

sigcode1 uintptr

sigpc uintptr

gopc uintptr // goroutine 当前运行函数的 PC 值

ancestors \*[]ancestorInfo // ancestor information goroutine(s) that created this goroutine (only used if debug.tracebackancestors)

startpc uintptr // 触发这个 goroutine 的函数的 PC 值

racectx uintptr

waiting \*sudog // sudog structures this g is waiting on (that have a valid elem ptr); in lock order

cgoCtxt []uintptr // cgo traceback context

labels unsafe.Pointer // profiler labels

timer \*timer // cached timer for time.Sleep

selectDone uint32 // are we participating in a select and did someone win the race?

// GC 时存储当前 Goroutine 辅助标记的对象字节数

gcAssistBytes int64

}

## 网络轮训器

golang对linux的epoll和mac的kqueue等都封装成统一的接口 netpoll.

水平触发与边缘触发 <https://zhuanlan.zhihu.com/p/363353777>

就是水平触发就是 如果缓冲区有数据就发消息告诉你可读了

边缘触发是只有缓冲区数据满了才会告诉你可读, 所以如果你用边触发读数据要循环读直到读完. 边缘触发好处就是可以减少epoll\_wati的调用 go就是边缘触发.

//水平触发

ret = read(fd, buf, sizeof(buf));

//边缘触发

while(true) {

ret = read(fd, buf, sizeof(buf);

if (ret == EAGAIN) break;

}

### **go netpoller剖析**

可以先简单看个tcp协议的echo server demo，可以用nc 127.0.0.1 8990去连接测试

**package** main

**import** (

"log"

"net"

"strings")

**func** **handler**(conn net.Conn){

**defer** conn.**Close**()

**const** BUFF\_SIZE = 1024

**var** buff = make([]**byte**,BUFF\_SIZE)

**for** {

n,err **:=** conn.**Read**(buff)

**if** err **!=** **nil**{

log.**Println**(err)

**break**

}

**if** n > 0{

**if** strings.**Contains**(string(buff),"exist") {

log.**Println**("close connection",conn.**LocalAddr**())

**break**

}

log.**Println**(string(buff[:n**-**1]))

}

}}

**func** **startServer**(){

l,err **:=** net.**Listen**("tcp","127.0.0.1:8990")

**if** err **!=** **nil**{

log.**Println**("err",err)

**return**

}

**for** {

c,err **:=** l.**Accept**()

**if** err **!=** **nil**{

log.**Println**(err)

**continue**

}

log.**Println**("accept connection")

**go** **handler**(c)

}

}

**func** **main**() {

**startServer**()}

可以看到，核心的函数有三个，分别是net.Listen，l.Accept和conn.Read，接下来将会一一解析。

在解析之前，我们可以明确一些信息：

1. go net包中server的实现是基于epoll（这里指的是linux平台，其他平台有其他的多路复用的实现例如kqueue，go目前已经能支持跨平台了）
2. go net的连接模型会为每个连接都建立一个goroutinue，这个goroutinue在对应fd没有被就绪的时候会被park住（什么叫做被park住，简单来说就是这个goroutinue被打包了之后找到一个地方存，直到socket就绪的时候会被唤醒），合适的时候会被唤醒（唤醒是由runtime层做的，需要先了解下go的GMP）

**net.Listen**

我们初始化使用的是tcp的服务器，所以大概的调用路径如下

listenTCP->internetSocket->socket->newFD，newFD这个函数是比较重要的，这里引出了net包中两个重要的数据结构netFD和poll.FD，poll.FD下面还有一个关键的数据结构pollDesc（这三个数据结构可以理解为对操作系统接口调用的层层封装）。

**func** **newFD**(sysfd, family, sotype **int**, net **string**) (**\***netFD, **error**) {

ret **:=** **&**netFD{

pfd: poll.FD{

Sysfd: sysfd,

IsStream: sotype **==** syscall.SOCK\_STREAM,

ZeroReadIsEOF: sotype **!=** syscall.SOCK\_DGRAM **&&** sotype **!=** syscall.SOCK\_RAW,

},

family: family,

sotype: sotype,

net: net,

}

**return** ret, **nil**}

在pollDesc中有个init方法

**func** (pd **\***pollDesc) **init**(fd **\***FD) **error** {

serverInit.**Do**(runtime\_pollServerInit)

ctx, errno **:=** **runtime\_pollOpen**(uintptr(fd.Sysfd))

**if** errno **!=** 0 {

**return** **errnoErr**(syscall.**Errno**(errno))

}

pd.runtimeCtx = ctx

**return** **nil**}

其中有个核心函数runtime\_pollOpen，我们仔细看看它的实现

*//go:linkname poll\_runtime\_pollOpen internal/poll.runtime\_pollOpen***func** **poll\_runtime\_pollOpen**(fd **uintptr**) (**\***pollDesc, **int**) {

**...**

errno **:=** **netpollopen**(fd, pd)

**if** errno **!=** 0 {

pollcache.**free**(pd)

**return** **nil**, int(errno)

}

**return** pd, 0}

**func** **netpollopen**(fd **uintptr**, pd **\***pollDesc) **int32** {

**var** ev epollevent

ev.events = \_EPOLLIN | \_EPOLLOUT | \_EPOLLRDHUP | \_EPOLLET

**\***(**\*\***pollDesc)(unsafe.**Pointer**(**&**ev.data)) = pd

**return** **-epollctl**(epfd, \_EPOLL\_CTL\_ADD, int32(fd), **&**ev)}

在*poll\_runtime\_pollOpen* 函数中可以看到最终是调用了epollctl的操作系统接口把我们fd放入（epoll\_*create，epoll\_waite，epoll三个函数可以看上面的epoll demo*）

**l.Accept**

这个函数的作用是accept得到一个连接（**返回的连接初始化的时候会加入epoll中**）

沿着调用路径查看下去，最终可以看到poll.FD中的Accept函数，我们需要重点关注fd.pd.waitRead这个函数

**func** (fd **\***FD) **Accept**() (**int**, syscall.Sockaddr, **string**, **error**) {

**...**

**for** {

s, rsa, errcall, err **:=** **accept**(fd.Sysfd)

**if** err **==** **nil** {

**return** s, rsa, "", err

}

**switch** err {

**case** syscall.EINTR:

**continue**

**case** syscall.EAGAIN:

*// 重点看这里* **if** fd.pd.**pollable**() {

**if** err = fd.pd.**waitRead**(fd.isFile); err **==** **nil** {

**continue**

}

}

**case** syscall.ECONNABORTED:

*// This means that a socket on the listen* *// queue was closed before we Accept()ed it;* *// it's a silly error, so try again.* **continue**

}

**return** **-**1, **nil**, errcall, err

}}

**func** **netpollblock**(pd **\***pollDesc, mode **int32**, waitio **bool**) **bool** {

**if** waitio **||** **netpollcheckerr**(pd, mode) **==** 0 {

**gopark**(netpollblockcommit, unsafe.**Pointer**(gpp), waitReasonIOWait, traceEvGoBlockNet, 5)

}}

最终会调用netpollblock，当waitio为false的时候就会调用gopark暂停当前goroutinue，也就是如果当前socket没有处于就绪状态，会把goroutinue park住，使其不使用cpu资源进行空转之类的操作。

那这里被park住了，那么有个问题，**这个goroutinue又是怎么被重新调度起来的呢**？

对应epoll*wait的执行函数在netpoll\_epoll.go文件中*

**func** **netpoll**(delay **int64**) gList {**...**}

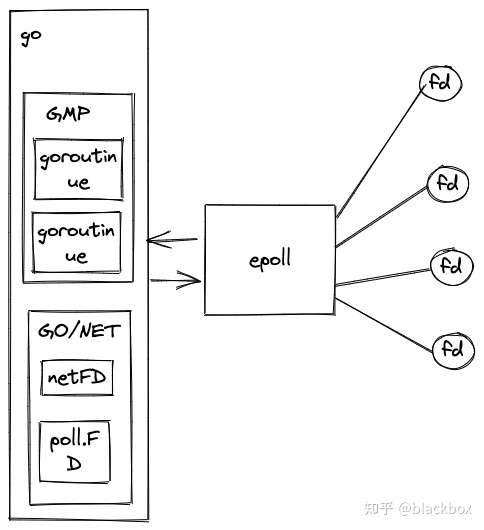
netpoll函数的作用是通过epoll\_wait去拿到就绪的goroutinue的列表，可以看到函数的返回是gList。

这个netpoll在findrunnable函数中被调用，findrunnable函数会被schedule函数调用，了解GMP模型的话可以知道，schedule是为了让可执行的goroutinue再次调度起来。除了schedule函数sysmon函数中也会调用netpoll，sysmon是一个golang runtime自带的监控任务，不需要额外的P就可以绑定运行，程序运行时会分配一个线程去执行sysmon任务，sysmon可以用来抢占一些陷入系统调用或者超时运行的协程，以及定时的gc。

**conn.Read**

conn.Read的逻辑和Accept类似，都是基于非阻塞IO+多路复用+GMP实现的，可以简单总结一下

1. 当调用conn.Read的时候先会执行fd的read方法，当fd没有数据可以读的时候会返回EAGAIN
2. 当返回EAGAIN的时候表示没有数据，需要重试
3. 这时候就会被gopark住
4. 当socket处于就绪状态的时候会把对应协程唤醒，执行业务逻辑，**总的来说就是event loop不需要我们去维护，runtime的sysmon以及schedule会帮我们维护event loop**

简易的go netpoller模型

### **存在的问题**

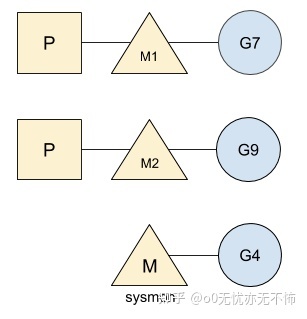
go netpoller存在的问题是每个连接都需要一个goroutinue，虽然goroutinue分配的栈空间和其他资源相比线程来说是很小的，但是依旧需要一定内存占用，当应用程序的连接数上百万的时候还是容易出现瓶颈。

### **其他高性能的网络库**

基于上面存在的问题，目前比较成熟的方案是多reactor模型，大概一个event loop去做多路复用，然后把就绪的连接交给后续的工作线程/协程或者是线程/协程池去处理，包括redis和nginx都用上了类似模型，关于go语言的话，netpoller和gnet都是多reactor模型的实现。

## Sysmon

Go 的标准库提供了一种监测应用程序的线程,并帮你 (找寻) 程序可能遇到的瓶颈. 该线程称为sysmon，即系统监视器 (system monitor).在GMP 模型中,这个 (特殊) 线程未链接任何的 P, 这意味着调度器 (scheduler) 没有将其考虑在内, 因此始终处于运行状态.



sysmon线程的作用很广, 主要涉及以下方面:

* 由应用程序创建的计时器 (timers). sysmon线程查看应该在运行却仍在等待执行时间的计时器. 在这种情况下, Go 将查看空闲的 M 和 P 列表, 以便尽可能快地运行它们.
* **网络轮询器和系统调用. 它将运行在网络操作中被阻塞的 goroutine.**
* 垃圾回收器（如果已经很长时间没有运行）. 如果垃圾回收器已经两分钟没有运行,则 sysmon 将强制执行一轮垃圾回收 (GC).
* 长时间运行的 goroutine 的抢占. 任何运行时间超过10 毫秒的 goroutine 都会被抢占, 将运行时间 (running time) 留给其他 goroutine.# io模型

## Timer

1. type timer struct {

2. tb \*timersBucket // the bucket the timer lives in // 当前定时器寄存于系统timer堆的地址

3. i int // heap index // 当前定时器寄存于系统timer堆的下标

4.

5. when int64 // 当前定时器下次触发时间

6.

period int64 // 当前定时器周期触发间隔（如果是Timer，间隔为0，表示不重复触

发）

7. f func(interface{}, uintptr) // 定时器触发时执行的函数

8. arg interface{} // 定时器触发时执行函数传递的参数一

9. seq uintptr // 定时器触发时执行函数传递的参数二(该参数只在网络收发场景下使用)

10. }

会有一个叫做timerbucket的结构，所有的timer都放在这里面，然后有专门的系统协程去调度这些timer。

这些timer会被存到一个小顶堆里面，然后距离执行时间最近的timer放在堆顶，这样去调度就可以了。

func cronMock() {

for {

//从堆中获取时间最近的定时器

t := getNearestTime()

//如果时间还没到，则continue

t.delta > 0 {

continue

}else{

//时间到了，将当前的定时器再加一个钟

t.when += t.duration

//将堆重新排序

siftdownTimer()

//执行当前定时器指定的函数，即sendTimer

t.sendTimer()

}

}

}

## GC

GOGC默认值是100，举个例子：你程序的上一次GC完，驻留内存是100MB，由于你GOGC设置的是100，所以下次你的内存达到200MB的时候就会触发一次GC，如果你GOGC设置的是200，那么下次你的内存达到300MB的时候就会触发GC。

https://blog.csdn.net/cjs68/article/details/103958681

## 反射

switch str := value.(type) {

case string:

return str

这样interface.(type) 配合switch case

## Context

Cancelctx cancel了父, 会一层层给子也发消息,那么cancelctx一定会有一个字段存的childlist.

valuectx会先查自己,自己查不到就去查父,那么一定会有一个字段是存的parent.

为了能cancel父的时候把子也cancel了,所以cancelctx肯定有个地方存了自己的childmap.那么valuectx肯定也存了自己的parent.

## GORM

CreateOrUpdatePoiCardRiskControl(d \*model.DbRiskCtrlCard, tx \*gorm.DB) error {

return tx.Table(obj.GetPoiRiskCtrlCardITableName()).Clauses(clause.OnConflict{

Columns: []clause.Column{{Name: "bid"}},

DoUpdates: clause.AssignmentColumns([]string{"content"}),

}).Create(&d).Error

类似于select \* from table where age > 20 limit 1000000,10这种查询其实也是有可以优化的余地的. 这条语句需要load1000000数据然后基本上全部丢弃,只取10条当然比较慢.

用游标的话,其实只load了10个 所以可以看到 游标比limit offset的性能要强非常多.

orm框架主要是链式调用,一个结构体 有where函数 有limit等函数 这些函数返回这个结构体自己 ,就可以实现链式调用了.

假设表中有一千万条记录，某个状态为0的记录总数大概会有100条，那么你想查询状态为0的记录时，有没有索引影响非常大，而查询状态为1的记录，则索引基本无用。如果两种状态的记录数相差无几的话，索引也基本无用。举个例子，假设表中有一千万条记录，某个状态为0的记录总数大概会有100条，那么你想查询状态为0的记录时，有没有索引影响非常大，而查询状态为1的记录，则索引基本无用。如果两种状态的记录数相差无几的话，索引也基本无用。

走索引一般都需要回表的, 回到聚集索引去捞真正的数据.

if err := Db.Model(xy).Where("id = ? ", id).Update("sign\_up\_num", gorm.Expr("sign\_up\_num+ ?", 1)).Error; err != nil {

return false

}

Save是会更新0值的,但是save的参数只能是个结构体,因为他会更新所有值.

mysql加锁

var res Result

 err = tx.Table("cron\_job").Raw("select exec\_state, exec\_time from cron\_job where id = ? for update", id).Scan(&res).Error

 if err != nil {

解锁

tx.Commit().Error

orm框架主要是链式调用.一个结构体 有where函数 有limit等函数 这些函数返回这个结构体自己 ,就可以实现链式调用了,因为我们这 ORM 的底层本质是 SQL 拼接，所以，我们需要把各种操作方法生成的数据，都保存到这个结构体的各个变量上，方便最后一步生成 SQL。

var expSum sql.NullInt64

if err := mdb.GormTJ.WithContext(ctx).Table(common.TableNameExpSum).Select("SUM(exp\_count)").

Where("sum\_time >= ?", timeStart.Unix()).

Where("sum\_time < ?", timeEnd.Unix()).

Where(where).Find(&expSum).Error; err != nil {

## Sql

括号里的数字叫数据的宽度，不同的数据类型对宽度的处理也不一样：

1、整数类型：这里显示的宽度和数据类型的取值范围是没有任何关系的，显示宽度只是指明Mysql最大可能显示的数字个数，数值的位数小于指定的宽度时会由空格填充；

如果插入了大于显示宽度的值，只要该值不超过该类型的取值范围，数值依然可以插入，而且能够显示出来。

如果你不设置宽度，系统将添加默认的宽度tinyint(4)、smallint(6)、mediumint(9)、int(11)、bigint(20)，这些默认的宽度是跟该类型的取值范围长度相关。

2、字符串类型：字符串类型这个宽度才真的用上了。不管是char还是varchar，宽度都定义了字符串的最大长度；例如上面的 password varchar(20)，如果你输入了一个21个字符的密码，那么保存和显示的只会是前20个字符，你将丢失一个字符信息，char同理。由于varchar是变长存储的，所以实际开发中我们一般都把varchar的宽度设为最长255，它会根据实际数据长度变化，反正你没用完它也不会浪费空间。char是定长存储，定义多长就是多长。

3、浮点和日期等数据类型：对数据的宽度没有要求，一般也不设置，默认是0。

order by id会大幅降低查询速度,其他索引就没问题,还是不要order by id

IF(process\_status = 2, 1, 0) as process\_status

select user\_name, count(DISTINCT tel) from user group by user\_name;

多个or条件要用括号扩起来

多个not in not like 要用or来链接.

INSERT INTO t\_stock\_chg(f\_market, f\_stockID, f\_name) VALUES('SH', '600000', '白云机场') ON DUPLICATE KEY UPDATE f\_market='SH', f\_name='浦发银行';

Insert on dunplicate key 就是有了就更新 没有就insert

那么到底怎么判断这个是不是已经存在呢,需要这个表有一个主键id或者uniquekey, 如果你的参数中包含了已经存在的主键id或者唯一key,那么就用这个找到了已有的数据进行更新.注意的是,如果最终是insert,effectrow是1, 最后是更新,effectrow是2, 如果你传的参数和表里的各个字段都一样,是不是做任何操作的,udpatetime也不会变的.

查看表的大小

select concat(round(sum(data\_length/1024/1024),2),'MB') as data from information\_schema.tables where table\_schema='ev\_courseware' and table\_name='user\_banke\_wrong\_item\_record' limit 10;

select record\_id, ea\_item\_id, count(\*), max(update\_time) from user\_question\_record where subject = 2 and user\_id = 1 group by ea\_item\_id;

mysqldump -h[47.92.202.123](/Users/yangwenshuo/Documents\\x/47.92.202.123) --skip-lock-tables --skip-add-locks -uhuamath\_dragon\_w -pebe00208299ad3bcb8  -d huamath\_dragon > dump.sql

只复制表结构

mysqldump -h[10.225.88.235](/Users/yangwenshuo/Documents\\x/10.225.88.235) -P3306 -uev\_boe\_w -p6abPHgXPTzxTB9P\_3NM9WeQvshLVKNLY --skip-lock-tables --skip-add-locks --databases ev\_boe --tables chat\_info --where "create\_time>\"2020\"" | mysql -h[10.225.124.243](/Users/yangwenshuo/Documents\\x/10.225.124.243) -P3306 -uev\_classroom\_w -plno3sGjYGd81TS2\_GJxHP66VQuHLBaWt

复制所有数据

数据库里面是null的话查出来的是空串 并不是null这个字符串的啊

update xxx= xx +1 这样是没有问题的

show index  from user\_examination;

drop index xxx on table name

Find(&userExamList).  必须要取地址

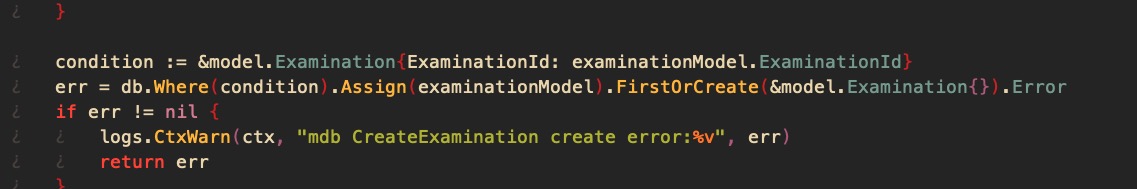
show create table banke;

tx := conn.Begin()

 var res Result

 err = tx.Table("cron\_job").Raw("select exec\_state, exec\_time from cron\_job where id = ? for update", id).Scan(&res).Error

加锁



没有就create,有就update, assgin和firstorcreate

select examination\_id, count(\*) count from examination group by examination\_id having count > 1;

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!!!!!

update xxx= xx +1 这样是原子的.

db的分页是走的offset,偏移量不会把所有的数据都捞出来.

但是如果这个操作在两个事务中还是会有问题

Left join xxxxx on a.user\_id = b.user\_id where b.user\_id is not null;

用on的话后面得是一个where.

SELECT \* FROM table WHERE zongbu NOT REGEXP'北京|上海|深圳|天津|香港|沈阳';

多个not like无论是or还是and都不好使，要用这个。

所以解决幻读问题也可以用lock in share model配合nextkeylock

唯一键冲突是导致自增主键id不连续的第一种原因

事务回滚是导致自增主键id不连续的第二种原因

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.

countid属实要比count(\*)快

text是可以变长存储的,就是容量比较大的varchar

读写分离 分了主从的话就一定会有那种双机房问题,写到了主里去从读就读不到,但是如果你setnx这是个写操作就会全到主里面.

那么这种双机房问题就咋办呢,电商公司一般都不会读写分离的,一是容易出问题,数据量大的话还是要走分库分表,头条这种feed流起家的一般都是走读写分离.

## 加密

MD5算法运1000次的平均时间为：226ms

SHA256算法运1000次的平均时间为：473ms

总而言之，md5和sha256都是密码散列函数，加密不可逆。虽然都不能防止碰撞，但是相对而言，md5比较容易碰撞，安全性没有sha256高

aes加密是可以解密的

md := md5.New()

md.Write([]byte("57b545da-cf8d-11ec-bd34-b8cef618cd4a"+ "https://taojin-pic-bj.cdn.bcebos.com/0dfe00d19e8eaaa5965252cf61b1c222.jpg"))

fmt.Println(fmt.Sprintf("pic\_%x", md.Sum(nil)))

md5的东西要用16进制打印

## 异步

1 异步回调的和核心是操作系统是知道啥时候 io结束的需要操作系统提供回调的工具

2,所谓回调地狱是有多个回调的函数,需要按照顺序先后执行



封装下就会变成这样



你有多个io回调 人家是会有个epoll的类似的东西的 吧你多个io回调注册上去的

反正异步就是这个鸟样子的.

## IDE

https://juejin.cn/post/6994695276468387848 左边栏

## 文章

<https://mp.weixin.qq.com/s/gVNKibDQ6UsX_q8_CHvg1A> 稳定性规范

<https://mp.weixin.qq.com/s/JQqex6kON6ixJUwodRZByA> es

<https://books.studygolang.com/The-Golang-Standard-Library-by-Example/> go语言标准库

<https://zhuanlan.zhihu.com/p/95056679?utm_source=wechatMessage_article_bottom&from=singlemessage>

<https://gocn.vip/topics/9611> go1.4

<https://cloud.tencent.com/developer/article/1434134> 限流算法

<https://mp.weixin.qq.com/s/0IKxbt8MDH6Yqu1f00cwSA> kv

<https://zhuanlan.zhihu.com/p/79772089?utm_source=wechat_session&utm_medium=social&utm_oi=54330396770304> 切换上下文开销

[https://zhuanlan.zhihu.com/p/22557362?utm\_source=wechat\_session&utm\_medium=social&utm\_oi=54330396770304&from=singlemessage&s\_s\_i=EuzWvW5ULdKasS50J8NyYlWM5Vs%2FPJi1BiU4UfbOeDg%3D&s\_r=1](https://zhuanlan.zhihu.com/p/22557362?utm_source=wechat_session&utm_medium=social&utm_oi=54330396770304&from=singlemessage&s_s_i=EuzWvW5ULdKasS50J8NyYlWM5Vs/PJi1BiU4UfbOeDg=&s_r=1) 无锁queue

<https://blog.csdn.net/hil2000/article/details/51638751?depth_1-utm_source=distribute.pc_relevant.none-task&utm_source=distribute.pc_relevant.none-task> 性能测试

<https://blog.csdn.net/xiaosongluo/article/details/80292894> 代码覆盖率

<https://blog.csdn.net/liumiaocn/article/details/54922487> 代码覆盖率

[https://item.jd.com/11152134.html#comment](https://item.jd.com/11152134.html" \l "comment) 量化研究方法

<https://mp.weixin.qq.com/s/o2oMMh0PF5ZSoYD0XOBY2Q> go gc

<https://mp.weixin.qq.com/s/24AvtcL8_cz_pQg2RYEaiQ> cpu如何识别代码

<https://www.xttblog.com/?p=2943> redis为何这么块

<https://blog.csdn.net/w372426096/article/details/88352833> rpc原理

<https://tech.meituan.com/2016/12/02/performance-tunning.html> 常见性能优化侧罗

<https://mp.weixin.qq.com/s/QMSQG-pPElPhI457g6OBqQ> 为服务中台架构

<https://maimai.cn/article/detail?fid=1372321387&efid=pm8WOnFAPtUMV4ivWMP4Tw&use_rn=1> epoll 本质

<https://juejin.im/post/5dce54ca51882510ba1cbee6> 编程规范[https://juejin.im/post/5c4f16dbe51d454f342fb7e7#heading-19](https://juejin.im/post/5c4f16dbe51d454f342fb7e7" \l "heading-19) flink

<https://mp.weixin.qq.com/s/VQGF1vNvv-3GjdS6viuMIA> shell