04.07

有两场面试，一个是上海 ai lab（50/100），一个是上海一个小厂（80/100）

1. HTTP协议中的流式传输是用什么协议实现的

- 无论是传统的 HTTP 请求-响应，还是流式传输，本质上都是基于 TCP 这个传输层协议完成的数据传递。
- HTTP/2 流（stream）机制
  HTTP/2 引入了多路复用，多个逻辑 stream 复用一个 TCP 连接，每个 stream 可独立传输。
  底层依旧是 TCP，但数据以帧（frame）形式传输，更适合流式场景（比如 gRPC）。

2. 详细介绍一个aigc平台上传一个文件到返回用户信息整个过程和用到的中间件
   用户上传的文件通过网关接入（鉴权JWT、限流、路由转发Trafik），先存入对象存储(一般都是用云存储)，通过消息队列异步(kafka、rabbitmq)通知AIGC服务（GPU节点）处理结果，处理后的结果存入Redis/MySQL，最后前端轮询或推送结果展示给用户。
3. 固定窗口之间的区别

- 固定窗口
  将时间划分为一个个固定的区间（如每 1 秒一个窗口）。
  每个窗口单独统计请求数量，达到阈值就拒绝后续请求。
- 滑动窗口
  不是按固定时间段，而是按当前时间为参考点不断滑动窗口。每次请求的时候都检查时间点往前一段时间内的访问量。
  将一个大窗口划分成多个小窗口（如 1 秒划分为 10 个 100ms 的小格），实时更新统计。

4. 防止缓存击穿是怎么实现的

- 把每一个请求抽象为一个Call，相同的key请求用map映射到同一个Call
- 第一个key请求，查询map没有，新建一个Call存入map，并且使用wg.Add(1)
- 剩余的同一个key的请求查询map有对应的Call，但是被wg.Wait()阻塞，直到第一个key的请求把数据库中的数据更新到缓存并且使用wg.Done()，然后剩余的请求才能拿到缓存中数据
- 生成新的Call和销毁一个Call的过程需要用Mutex加锁

5. 仿GroupCache的流程

```
Group.Get(key)
    ├── 本地缓存命中？ → 是 → 返回
    │
    └── 否 → load(key)
          ├── 找到远程节点→ 是本身吗？ 否 → getFromPeer → 成功返回
          │                          ↓ 是 → GetLocally
          └── getLocally
                ├── 调用 Getter 拿原始数据
                └── 缓存到 mainCache
          ↓
        返回结果
```

6. HTTP缓存的实现方式

- 每个网络端口抽象为一个peer（节点）
- 用三个网络端口开启三个进程，每个进程都有自己的mainCache和peers，这个peers里面就包括三个端口
- 假设第一个进程对应的端口中的mainCache没有缓存数据，就利用一致性哈希选择一个peer节点，然后向这个peer节点发送HTTP请求，这个peer节点对应的进程的mainCache中如果也没有，但是再利用一致性哈希只能找到自己所以就从SlowDB中加载到自己的mainCache并返回数据
- 下一次访问同一个key，还是第一个节点没有缓存数据，用一致性哈希选择节点，从那个节点的mainCache中取数据

7. Redis常见数据类型与使用

Redis 数据类型与典型应用场景

数据类型	典型应用场景
String	- 缓存热点数据（用户 session、页面片段） - 计数器（浏览量、自增 ID） - 存储验证码、token、配置项
List	- 简易消息队列（任务推送、异步处理） - 时间线/评论系统（按时间顺序） - 日志系统收集
Hash	- 用户信息缓存（user:id -> name, email） - 商品或文章详情结构化存储

数据类型	典型应用场景
	- 用户购物车（商品 + 数量）
Set	- 标签系统（如文章标签、兴趣分类） - 社交系统好友/关注关系 - 签到、活跃用户去重
Sorted Set	- 排行榜系统（积分、热度） - 内容推荐排序（点击量、活跃度） - 延迟任务队列（score=时间戳）
HyperLogLog	- UV 统计（独立访客） - 活动参与人数估算 - 大规模数据去重统计
Bitmap	- 用户签到记录（按天位标记） - 是否看过某内容（位表示状态） - 用户权限或在线状态标记
Stream	- 高级消息队列（支持 ack、持久化） - 日志流系统 - 实时聊天、IoT 数据流

04-15

南京一个小厂（最快的一集，半小时不到）

1. goroutine底层，在golang中是什么地位。一个函数的多个G更倾向于分配到多个P还是一个P上（从全局队列拿G有没有什么倾向）
   goroutine的底层本质 是一个非常轻量级的用户态线程，其核心是 Go 运行时（runtime）自己维护的调度系统。在底层实现中，goroutine 被表示为一个结构体 G，由 Go 的 GMP 模型管理和调度。

- goroutine的id
- 保存上下文：寄存器信息、程序计数器（PC）、栈指针（SP）等，支持协程切换。
- 状态管理：每个 G 都有状态，如 _Grunnable, _Gwaiting, _Grunning。

当前 P 是你创建 G 的“发起者”，直接放入它的本地队列效率最高，不需要加锁（相比全局队列需要锁），也避免了繁琐的上下文切换。
2. 以前的是GM现在是GMP，P起到什么作用

功能项	说明
本地 G 队列（runq）	每个 P 拥有自己的 goroutine 队列，减少 G 的全局调度冲突。
调度上下文	P 保存运行时的调度状态、分配的资源（比如内存、缓存等）。
M 和 G 的粘合剂	M 只有在绑定 P 的情况下，才能执行 G。没有 P，M 无法调度。
调度逻辑的载体	Go 的调度循环 schedule() 是以 P 为中心进行的：找 G -> 找空闲 M -> 执行。

3. defer原理
   📦 一、编译期原理

- 在 编译阶段，defer 并不是立即生成调用指令，而是将其作为一个特殊的结构记录下来，最终统一在函数返回前处理。
- 每个函数在编译时会有一个 defer 栈（链表）。
- 遇到 defer，会将其封装为一个结构体 deferproc，加入当前函数的 defer 链表中。
- 编译器在函数 return 前会插入一段代码，调用这些 defer 的函数。
  🏃 二、运行期原理
- 在函数运行过程中：
- 执行到 defer 语句时：
- 会计算 defer 的函数参数值，并将函数指针 + 参数一起封装为一个结构体（称为 _defer）。
- 把这个 _defer 放入当前 goroutine 的 defer 链表中。
- 函数退出时（正常 return 或 panic）：会按**先进后出（LIFO）** 的顺序依次调用 _defer 链表中的函数。

4. HTTP和RPC区别

- HTTP是一个应用层协议，RPC是远程过程调用，只是一种函数（或方法）调用方式，与之对应的是本地调用，所谓的RPC协议其实只是基于TCP、UDP甚至是HTTP2改造之后的协议
- HTTP有很多适应浏览器的冗余字段，这些事内部服务用不到的，RPC可以摒弃很多HTTP Header中的字段（比如浏览器的各种行为）
- HTTP数据格式是JSON/XML（文本格式），但是RPC是Protobuf、Thrift（二进制格式），后者传输效率更高
- HTTP更适用于Web应用、前后端交互，而RPC更适用于微服务、分布式系内部通信。

RPC比HTTP出现的早，主要就是为了解决远程函数调用的问题。
一个单体服务内部的各个模块之间可以直接用本地函数（方法）调用，但是一个分布式微服务的系统内部各个模块的都是独立的服务，他们可能存储在不同的主机或者不同的容器中，那么他们就只能用远程

函数（方法）调用

5. 介绍一下MYSQL的索引、索引失效?

- 索引是数据库中用于加速查询效率的一种数据结构，就像书本的目录，可以快速定位数据而不用全表扫描。
- 加速查询、范围扫描、排序
- 常见的有主键索引、唯一索引、普通索引、组合索引
- InnoDB默认使用b+树
- 注意建立索引的时机和索引的使用
- Explain查询是否命中索引

6. docker编写dockerfile流程，RUN命令写一行还是多行好

- 指定基础镜像（FROM）、设置环境变量（ENV）、安装依赖（RUN）拷贝文件（COPY、ADD）、运行命令（CMD）
- RUN：在镜像构建阶段执行命令，并将执行结果提交为新的一层镜像。

写法风格	优点	缺点
一行一个 RUN	清晰易读、便于调试	每一条 RUN 会生成一个中间镜像层， 最终镜像变大，构建变慢
多条命令合并一行 RUN	减少镜像层数， 镜像更小，构建更快	可读性差，调试困难， 改动后容易导致缓存失效

7. 僵尸进程是什么，wait()/waitpid()有什么用？原理是?

- 子进程已经退出，但其父进程尚未调用 wait 或 waitpid 回收它的退出状态信息，导致其在系统中仍占据一个进程表项的特殊进程。
- wait() 阻塞等待任意一个子进程结束，并回收它的资源。
- waitpid() 可以精确指定等待某个子进程结束；支持非阻塞模式，更灵活。
- 子进程执行完毕后调用 exit()，进入退出状态。内核不会立即释放该进程的 PCB（进程控制块），而是将其标记为 Z（僵尸进程），并记录其退出状态。内核给父进程发送 SIGCHLD 信号，通知它有子进程退出。

8. 一个表里面只有姓名、性别和年龄三个字段，怎么设置索引比较好

字段	是否建议建索引	理由
name	✅ 是	查询常用字段，区分度高
age	✅ 是	可用于范围查询

字段	是否建议建索引	理由
gender	❌ 否	只有两种值，区分度低， 索引不生效
name和age也可以建立组合索引 （注意顺序）		

9. ceph的存储和使用，存大文件好还是小文件好，存静态文件好还是动态文件好

👉 RGW（前台）接收你上传的文件。
👉 RGW 把 example.txt 切成几个“页”大小的小块（对象）。
👉 RGW 查询 MON（管理员）：我要存这些小块，去哪放？
👉 MON 看当前 OSD 状态，结合 CRUSH 算法，告诉 RGW：
小块1 放 OSD3、OSD5、OSD7（副本）
小块2 放 OSD2、OSD4、OSD6
👉 RGW 把小块分别发到这些 OSD 存起来。
👉 RGW 在 omap 中记录：你这个 example.txt 是由哪几个小块组成的，每块的名字、大小等。
✅ 上传完成！

文件类型	是否适合使用 Ceph	理由
大文件（如视频、 备份镜像）	✅ 适合	更容易划分为对象，元数据压力小， 系统吞吐高
小文件（如图片、 日志、文档）	❌ 不太适合 （除非优化）	元数据数量庞大，造成性能瓶颈， 建议合并或使用 CephFS

文件类型	是否适合使用 Ceph	理由
静态文件（如图片、音频、HTML、 备份等）	✅ 非常适合	读取频繁、写入少， 适合对象存储特性，可高效缓存
动态文件 （如频繁更新的数据库文件）	❌ 不适合	高频更新会造成较高延迟，Ceph 不擅长高频小写入

南京全职公司电话面试（一个月两千多，没有住宿）：

1. 两个项目有没有遇到什么问题怎么解决的

- 第一个说了K8s没有部署成功

2. 除了protobuf优化节点数据序列化，还有没有什么方法
   Cap’n Proto（Captain proto）
   特点：比 Protobuf 更快、零拷贝、跨语言
   优点：支持直接访问内存结构（不需要 decode）。高吞吐、低延迟

04-16

武汉一个小厂（最逆天的一集）

- 因为第一道题我写过了，然后又让我写一道，总共写了40分钟
- 写完算法就问我，两个项目的代码量是多少（第一个不知道，第二个五六百，他表示不相信共享屏幕给他看第二个项目），然后问我gin的controller用了哪些（我的项目确实没有用过，然后我也没有学过）
- 但是唯一就是提醒我第一个项目有很多东西要补回来吧
- 第一个项目将近4000行，第二个项目将近五百行

04-17

南京小厂二面（面的很爽的一集，问的基本都是golang的，基本都会）

1. git如何放弃当前的修改
   git checkout\git restore\git reset
   放弃当前的修改（还没add）\git add的反向操作\撤回提交（可以跨好几个提交）
2. git如何合并分支，使得冲突比较少

- 确保每个分支都基于最新 main
  git rebase origin/main
- 设置合并顺序
  先把冲突小的（改动少、独立的）合并；
  再合并冲突大的；
  如果 feature-b 依赖于 feature-a，就先合 a，再合 b。
- 按顺序 merge 到主分支
  有冲突就手动解决，每次解决完都要 add + commit。

3. 项目的缓存击穿怎么实现？（略）
4. golang的协程相比较与线程的优点？

协程比较于线程：

- 轻量：1MB -> 2KB，轻轻松松创建成千上万个线程，这样也更好的利用系统的硬件资源
- 快速创建与销毁很快：协程的生命周期由Golang的调度器管理，并且线程的上下文切换也比较昂贵
- 无锁并发：协程之间的切换通常是无锁的，避免传统线程的锁竞争
- 更利于编写并发代码：Go的并发模型（channel+goroutine）编写的代码比传统的线程+锁的代码更加直观

5. 第一行写没有缓存的channel，第二行往里面写数据，能跑吗？
   直接过不了编译，all goroutines are sleep deadlock
6. map可以存哪些数据类型

- key必须放可以比较的数据类型
- 除了切片、map、func，其他的类型基本都可以比较

7. 调算法的API一般需要什么
   输入数据（如文本、图像、数值等）
   输入参数（如超参数、配置等）
   输出结果（如模型、预测值、处理结果等）
   错误处理（如无效输入、算法错误等）
   API 认证和授权（如 API 密钥、OAuth）
   请求方式和格式（如 GET/POST，JSON 格式等）
   性能需求和限制（如超时、数据大小限制等）
   版本控制（如 API 版本号）
8. channel的底层有哪些数据结构
   主要是就是三个环形队列和一个锁

- 一个缓冲区两个协程阻塞队列
- 一个是加锁的，防止读写时候出现并发问题

9. 函数退出前做一些处理，可以有什么方式
   defer、panic、recover
10. 各个数据类型到底是分配到堆上还是栈上。变量、常量放哪？

类型/情况	是否堆上分配
基本类型的常量	❌ 编译期内联，不分配
基本类型的局部变量	✅/❌ 逃逸则在堆上，否则栈上
函数返回变量的地址	✅ 一定逃逸
引用类型中的 make/map/slice	✅ 底层数据在堆上，头部逃逸则在堆上，否则在栈上

类型/情况	是否堆上分配
使用 new() 或取变量地址（&变量）	✅ 一般会导致逃逸，分配到堆上

- 准确地说，你并不需要知道。Golang 中的变量只要被引用就一直会存活，存储在堆上还是栈上由内部实现决定而和具体的语法没有关系。
- Golang 编译器会将函数的局部变量分配到函数栈帧上。然而，如果编译器不能确保变量在函数 return 之后不再被引用，编译器就会将变量分配到堆上。而且，如果一个局部变量非常大，那么它也应该被分配到堆上而不是栈上。
- 当前情况下，如果一个变量被取地址，那么它就有可能被分配到堆上,然而，还要对这些变量做逃逸分析，如果函数 return 之后，变量不再被引用，则将其分配到栈上。

11. 堆和栈的区别?

特性	栈（Stack）	堆（Heap）
生命周期	随函数调用自动创建和销毁	手动或由垃圾回收管理，生命周期长
分配速度	✅ 快（连续内存）	❌ 慢（离散空间，碎片多，需 GC 管理）
内存大小限制	通常较小（Go 会动态扩容）	较大
分配方式	编译器直接分配（无 GC 干预）	通常由 runtime/GC 分配与释放
释放方式	函数返回后自动释放	依赖垃圾回收机制
逃逸分析	没有逃逸 ⇒ 栈分配	有逃逸 ⇒ 堆分配
地址是否可取	✅ 可取，但函数结束失效	✅ 可取，地址长期有效

12. 哈希冲突的解决方式
    开放地址法、链地址法、再哈希、map
13. C/C++语言种数组指针+1代表什么

```
int arr[5] = {10, 20, 30, 40, 50};
int* p = arr;
std::cout << *(p) << std::endl;     // 输出 10
std::cout << *(p + 1) << std::endl; // 输出 20
std::cout << *(p + 2) << std::endl; // 输出 30
```

p + 1 实际上是 p 加上 sizeof(int) 个字节后的地址
所以 *(p + 1) 解引用后就是第二个元素：arr[1]

14. Restful API

内容：

- 资源（Resource）：被操作的对象，用 URL 表示。如 /users/1 表示 ID 为 1 的用户。
- 方法（Method）：使用 HTTP 的标准方法（GET、POST、PUT、DELETE 等）来操作资源。
- 表现层（Representation）：客户端与服务器通过 JSON、XML 等格式传输资源的表现。
- 无状态（Stateless）：每个请求都是独立的，服务端不会保存客户端状态。

设计规范：

- 使用 名词复数 表示资源：如 /users、/articles
- 避免动词（操作由 HTTP 方法表达）
- 子资源用嵌套表示：如 GET /users/1/orders 表示获取用户 1 的订单
- 使用状态码表示结果

4-21

小红书（问的都是没学过的）

1. linux启用一个进程，有哪些系统调用

- fork(): 创建当前进程的副本(子进程)
- exec()系列函数(execl, execle, execlp, execv, execvp, execvpe): 用新程序替换当前进程映像
- clone(): 类似于fork但更灵活，可以控制共享哪些资源

典型流程是先fork创建子进程，然后在子进程中调用exec加载新程序。

2. 查看ceph，IO负载、cpu负载
   IO负载：

- 查看整体IO负载
  ceph osd perf
- 查看每个OSD的IOPS和延迟
  ceph osd df
- 更详细的性能数据
  ceph osd pool stats

cpu负载：

- 查看monitor和OSD进程的CPU使用情况
  top -p $(pgrep -d',' -f "ceph-(mon|osd)")

- 或者使用ceph自带的性能插件
  ceph mgr module enable perf
  ceph perf

3. 怎么接收用户（进程）信号，用context退出,通知协程退出底层（业务）是谁做的
   协程需要不断的用select监听ctx.Done()，他其实就是一个channel, 如果在协程外使用cacel之后，这个channel会发送一个信号，协程收到这个信号之后才会主动退出

```
package main

import (
	"context"
	"fmt"
	"os"
	"os/signal"
	"syscall"
	"time"
)

func main() {
	// 创建可取消的 context
	ctx, cancel := context.WithCancel(context.Background())
	defer cancel() // 确保资源释放

	// 监听 SIGINT (Ctrl+C) 和 SIGTERM (kill)
	sigChan := make(chan os.Signal, 1)
	signal.Notify(sigChan, syscall.SIGINT, syscall.SIGTERM)

	// 启动工作协程
	go worker(ctx, "Worker 1")
	go worker(ctx, "Worker 2")

	// 阻塞等待信号
	sig := <-sigChan
	fmt.Printf("\nReceived signal: %v\n", sig)

	// 触发优雅退出：取消 context，通知所有协程
	cancel()

	// 等待协程退出（可选超时）
	time.Sleep(1 * time.Second) // 模拟清理耗时
	fmt.Println("All workers exited, shutting down.")
}

func worker(ctx context.Context, name string) {
	for {
		select {
		case <-ctx.Done():
			fmt.Printf("%s: Received shutdown signal, cleaning up...\n", name)
			time.Sleep(500 * time.Millisecond) // 模拟清理操作
			fmt.Printf("%s: Exited.\n", name)
```

```
                return
            default:
                fmt.Printf("%s: Working...\n", name)
                time.Sleep(1 * time.Second)
            }
        }
}
```

4. RGW一定会去查MON吗

- 在Ceph中，RGW操作不一定总是需要查询MON(Monitor)。
  客户端缓存状态：如果客户端有最新的OSDMap缓存，可能不需要查询MON
  操作类型：对于数据读取操作，通常只需要查询OSD，不需要MON
  集群状态稳定性：在稳定状态下，大部分操作可以直接与OSD交互
- MON主要在以下情况被查询：
  客户端首次连接集群
  OSDMap版本过期或失效
  异常场景（如OSD故障）可能触发MON查询以更新集群视图。
  执行需要集群全局状态的操作(如创建池、修改CRUSH规则等)

5. 怎么解决缓存击穿（写代码）（略）
6. 一致性哈希怎么实现的(略)
7. CRUSH算法是怎么实现的?

- CRUSH 是一种 去中心化的、伪随机的、可控的映射算法，根据一定的规则 直接计算出数据对象应该存在哪些 OSD 上，而不是通过中心节点查表。
- 输入：
  数据对象的 ID（如 object name 或 hash）
  CRUSH Map（集群拓扑 + 权重 + 规则）
- 输出：
  一组 OSD ID（表示将对象副本放在哪些 OSD 上）
- CRUSH Map（核心元数据结构）
  包含以下信息：
  OSD 列表：所有的存储节点
  层次结构（Hierarchy）：比如 data center → rack → host → osd
  权重：每个设备的相对容量
  规则（ruleset）：比如“从不同的 rack 中选 3 个 OSD”
  👉 这个 Map 是所有客户端都共享的副本，也就是说客户端自己就可以用它来算出数据存放在哪些 OSD。

- 选择流程（简化版）
  定位桶（Bucket）：根据规则选定在哪个 rack/host 下选择
  权重选择：优先选择空闲空间多的 OSD
  避开失败节点：自动跳过 down 状态的 OSD
  返回 OSD 列表：确定每个对象的副本放哪

度小满

1. map实现并发安全的方式有哪些

- Mutex、RWMutex
- sync.Map
- 用一些第三方库concurrent-map

2. map的底层怎么实现的（略）
3. MVCC中readView什么时候生成的

- ReadView 是在事务内第一次执行快照读（如 SELECT）时才生成的，用于确定当前事务能看到哪些版本的数据。
- 对于 当前读（加锁读，如 SELECT ... FOR UPDATE / UPDATE / DELETE），不使用 ReadView，而是直接读最新版本并加锁。

4. Golang GC的流程和劣势

- GC 仍然存在开销（STW 或 CPU 消耗）
  虽然 Go 1.5+ 已将 STW（Stop-The-World） 时间大大降低，但仍不可完全避免。
  在高频率、大对象创建的系统中，GC 扫描、标记、清理依然会占用 CPU 时间。
- 内存占用偏高
  Go 的 GC 倾向于 "以空间换时间"，为减少 GC 次数，往往不会急于释放内存。
  程序空闲时也可能保持较高的 RSS（常驻内存集），不利于部署在内存敏感的容器中（如 k8s）。
- 分配成本较高（频繁GC会降低性能）
  如果程序中频繁分配短生命周期的小对象，会增加 GC 频率，影响程序吞吐量。
  使用过多小 slice、map 等，容易触发更多 GC。

对于低延迟系统（如游戏服务器、交易系统）可能导致响应抖动。

5. 算法：三个协程循环打印
   三个协程都有一个for循环，三个chanel依次接受和发送
6. k8s的组件
   一个Control plane，多个Node
   用户通过 kubectl apply -f deployment.yaml 提交请求到 API Server。

API Server 将请求存储到 etcd，然后各个组件从 etcd 同步状态。
Controller Manager 看到 ReplicaSet 不足，创建新的 Pod。
Scheduler 监听到有新 Pod 无 Node，负责为其选择合适的节点，并更新绑定信息。
该 Node 上的 Kubelet 读取 Pod 信息，调用 Container Runtime 创建容器。
Kubelet 监控 Pod 运行状态并持续汇报给 API Server。
kube-proxy 根据 Service 定义设置网络规则，保证服务可访问。
Controller Manager 定期检查实际状态和预期状态是否一致（比如副本数），发现偏差就通过 API Server 调整。

04-22

百度

1. 单体服务是怎么拆成微服务的，有哪些微服务

- account：处理用户注册登录、查询用户信息、文件列表、文件重命名等等
- apigw：鉴权、将用户的用户注册登录的HTTP请求转换为服务之间的RPC调用
- dpproxy：数据库的连接、orm做一些业务处理
- upload：文件（分块）上传
- download：文件下载
- transfer：文件上传迁移

2. Rabbitmq做了什么，发送和接受什么数据？routing key、交换机、queue之间是怎么协同工作的

- 发送和接受文件元数据，比如文件名、文件hash值、当前路径和目标路径等等
- 首先创建一个交换机和多个queue，比如说采用direct模式，每一个消息和queue都会指定一个routing key，当生产者发送消息到交换机，交换机收到消息之后会根据消息的routing key转发到对应同样routing key的queue中

3. kafaka和Rabbitmq对比（略）
4. JWT的双Token是怎么样刷新Token的，怎么防止Token泄露？短时效Token怎么进行刷新的

- 短时效的Token放在客户端的内存或者cookie（持久性比较强）中
- 长时效的Token放在服务端的redis中
  每次登录的时候产生两个Token，然后转换页面的时候就会检查短时效的Token是否还在，如果还在就没事，如果不在就请求用服务端的长时效的Token刷新一个短时效的Token，同时长时效的Token也会刷新，如果长时效的Token也不存在了，那就需要重新登录

5. 文件的断点续传怎么实现？（略）
6. MYSQL主从复制怎么实现的，出现延迟怎么避免数据丢失

- 设置为半同步复制（semi-sync）：主库写数据时，必须等至少一个从库确认收到 binlog，才算 commit。
- GTID（全局事务标识）：从节点通过GTID定位未同步的事务，主节点恢复后继续同步。

7. 介绍仿做的GroupCache，为什么采用的protobuf实现的数据序列化，他有什么好处，有没有测试之前用json现在用protobuf的数据是什么样的

特性	JSON	Protobuf
可读性	人类可读（纯文本）	二进制格式（不可读）
编码/解码速度	较慢（需解析文本）	非常快（二进制解析）
消息大小	较大（字段名+结构冗余）	非常小（使用 tag 编码，无字段名）
网络传输效率	较低	高
使用便捷性	非常简单（无需额外工具）	较复杂（需编译器如 protoc）
多语言支持	广泛（几乎所有语言都支持）	广泛（支持 C++, Java, Python, Go 等）
调试方便性	直接查看内容	需借助工具反序列化

8. 算法：LRU（略）
9. 读golang程序

```
slice := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9} // 初始切片
s1 := slice[2:5]
s2 := s1[2:6:7] // 第二次切片
// 第一次切片
s2 = append(s2, 10) // 第一次追加
s2 = append(s2, 11) // 第二次追加
s1[2] = 20
fmt.Println("slice:", slice, "len = ", len(slice), "cap = ", cap(slice))
fmt.Println("s1:", s1, "len = ", len(s1), "cap = ", cap(s1))
fmt.Println("s2:", s2, "len = ", len(s2), "cap = ", cap(s2))
```

主要注意s1和s2底层都是slice，不要把s2底层当作s1

```
slice: [0 1 2 3 4 5 6 7 8 9] len =  10 cap =  10
s1: [2 3 4] len =  3 cap =  8
s2: [4 5 6 7] len =  4 cap =  5
slice: [0 1 2 3 4 5 6 7 10 9] len =  10 cap =  10
s1: [2 3 4] len =  3 cap =  8
s2: [4 5 6 7 10] len =  5 cap =  5
slice: [0 1 2 3 4 5 6 7 10 9] len =  10 cap =  10
s1: [2 3 4] len =  3 cap =  8
s2: [4 5 6 7 10 11] len =  6 cap =  10
slice: [0 1 2 3 20 5 6 7 10 9] len =  10 cap =  10
s1: [2 3 20] len =  3 cap =  8
s2: [4 5 6 7 10 11] len =  6 cap =  10
```