复习工具

https://javaguide.cn/home.html

https://labuladong.github.io/

https://pdai.tech/md/db/sql-mysql/sql-mysql-mvcc.html

面经

https://zhuanlan.zhihu.com/p/608484252

八股面经

https://segmentfault.com/a/1190000043759791

https://zhuanlan.zhihu.com/p/608484252

Java

AOP：面向切面编程，解耦

IOC控制反转：对象的创建交给容器控制，而不是代码

HashMap

Java 中 HashMap 的存储, 冲突, 扩容, 并发访问分别是怎么解决的

> Hash 表，拉链法（长度大于8变形为红黑树），并发访问不安全
>
> 扩容：找到不小于指定容量的2次幂，然后迁移数据。1.7迁数据是遍历桶重新hash，1.8通过判断最高位是0还是1选择原位置或者原位置+原容量
>
> 1.7和1.8的区别：
>
> - 数据结构上 1.7 数组+链表 1.8 数组+链表+红黑树
> - 插入的时候 1.7 **头插法** ，1.8 **尾插法** （节点的分布 符合泊松分布，超过8的概率很小）
> - 扩容的时候 1.7 是重新位运算计算hash 1.8是看二进制最高位0和1决定原位置还是原位置+原容量

HashMap 的并发不安全体现在哪?

> 拉链法解决冲突，插入链表时不安全，并发操作可能导致另一个插入失效

> - JDK1.7 HashMap线程不安全体现在：死循环、数据丢失
> - JDK1.8 HashMap线程不安全体现在：数据覆盖

Pre*pareStatement*为什么能避免注入?

> 是因为执行计划固定了，按照不带变量的SQL模板进行词法语法分析，生成执行计划。
>
> 执行计划不会因为值而变化，就不会出现类似于select 一条语句，后面加个分号就执行俩条SQL的情况

消息队列

什么时候用?

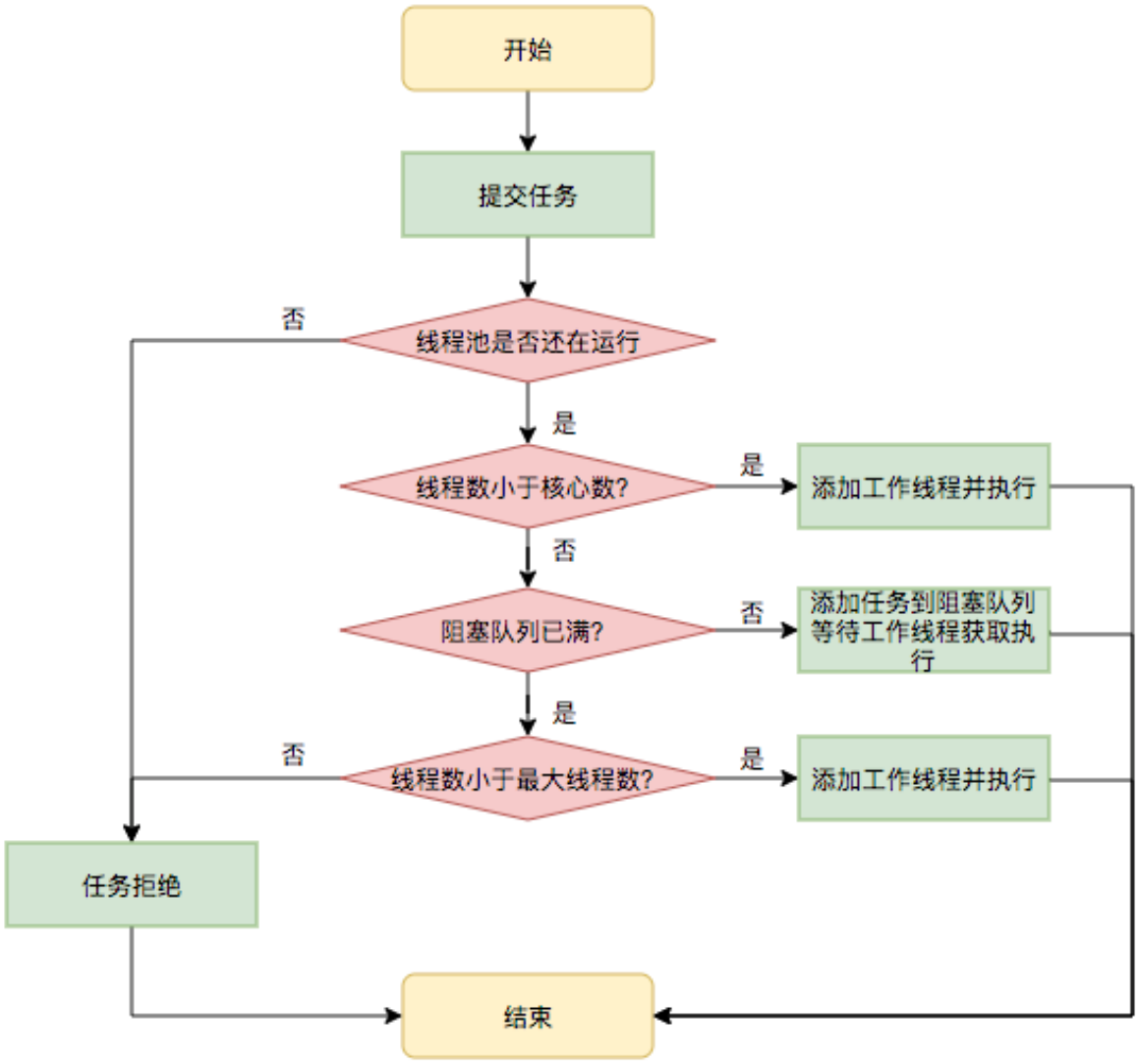
异步、削峰、解耦

Kafka 的消费者如何做消息去重?

> 幂等性可以用redis的setnx分布式锁来实现。比如操作订单消息，可以把订单id作为key，在消费消息时，通过setnx命令设置一下，offset提交完成后，在redis中删除订单id的key。setnx命令保证同样的订单消息，只有一个能被消费，可有效保证消费的幂等性!
> 要求不高的bloom过滤器也可以

线程池

核心线程数，最大线程数，存活时间，阻塞队列，线程池满的执行策略（抛弃，本线程执行，丢弃并抛异常）



其他

依赖倒转：高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象

面向接口而非实现

类加载器

> ### 类加载器
>
> 启动类加载器(BootStrapClassLoader)：用C++语言实现，是虚拟机自身的一部分，它负责将<JAVA_HOME>/lib路径下的核心类库，无法被Java程序直接引用
> 扩展类加载器(ExtClassLoader)：用Java语言实现，它负责加载<JAVA_HOME>/lib/ext目录下或者由系统变量-Djava.ext.dir指定位路径中的类库，开发者可以直接使用
> 系统类加载器(AppClassLoader)：用Java语言实现，它负责加载系统类路径ClassPath指定路径下的类库，开发者可以直接使用

双亲委派

定义：如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是**双亲委派模式**。

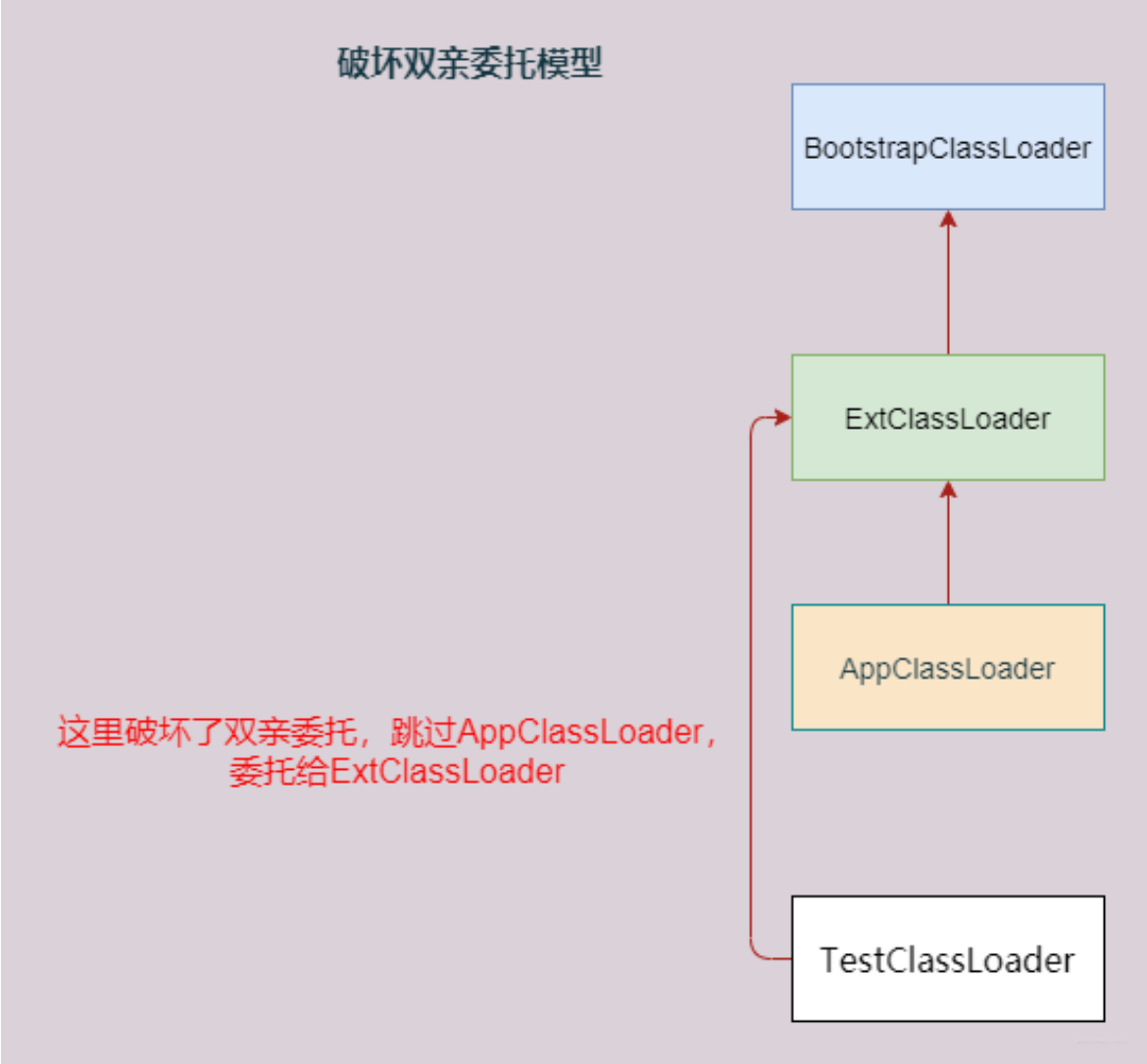
优点：采用双亲委派模式的是好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系，通过这种层级关可以避免类的重复加载，当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次。其次防止恶意覆盖Java核心API。

怎么破坏双亲委派（实现重复加载同一个类）：

自定义classLoader，并重新loadClass方法，可以绕过系统类加载器(AppClassLoader)，不过因为安全原因，扩展类和启动类加载器的双亲委派是不能破坏的。

Tomcat 和Mysql的JDBC就做了破坏委派

Tomcat 破坏是为了让多个web项目能加载不同版本的依赖jar



锁

volatile 解决变量在多个线程之间的可见性

volatile怎么保证可见性?

> 1）将当前CPU处理器缓存的数据写回到系统内存。
>
> 2）通过总线，让其他CPU里缓存该内存地址的数据无效。
>
> 紧张指令重排序

synchronied的底层原理?

> 在汇编里是monitorEnter、monitorExit指令，实际是靠 Java对象的对象头里 标记字段**(Mark Word)** 存的锁信息来判断当前线程能不能获取锁/资源。
>
> 【锁升级】：对象头里会存当前是什么级别的锁，哪些线程持有锁，最开始是同线程自动获取的偏向锁，在有其他线程竞争后升级为CAS的轻量级锁，再冲突就升重量级锁，抢锁失败的线程会阻塞等待锁释放。

代码加了synchronied不代表一定有冲突，看写法，有可能同一时间只有一个线程参与，所以才有锁升级这套

Mark Word 信息：

64位虚拟机

锁状态	56bit		1bit	4bit	1bit (是否偏向锁)	2bit (锁标志位)
	25bit	31bit				
无锁	unused	对象 hashCode	Cms_free	对象分代年龄	0	01
偏向锁	threadId(54bit)(偏向锁的线程ID)	Epoch(2bit)	Cms_free	对象分代年龄	1	01
轻量级锁	指向栈中锁的记录的指针					00
重量级锁	指向重量级锁的指针					10
GC 标志	空					11

相关好文：https://juejin.cn/post/6844903726545633287

- Java有哪些锁?

> 乐观、悲观锁、轻重量、自旋锁、公平不公平

> 乐观锁在Java里基本靠CAS，比如AtomicInteger的getAndIncrement()，在mysql里靠update时的where value=x
>
> 乐观锁的ABA问题靠 加版本号解决
>
> 共享锁：就是读锁，多个线程共用一个锁（ReadWriteLock）

ConcurrentHashMap如何保证线程安全的?

> 1.7是分段加锁，锁的是segment（继承了ReentrantLock）
>
> 1.8用 synchronized，synchronized锁的是table数组头结点
>
> 他们都是拉链，但1.7是头插法，1.8是尾插法而且用了红黑树

ConcurrentHashMap 1.7和1.8分别是怎么扩容的?

> 1.7 ConcurrentHashMap中的扩容是仅限于本Segment，也就是对应的小型HashMap进行扩容，所以是可以多线程扩容的。
>
> 1.8 在需要扩容时，首先会生成一个双倍大小的数组，生成完数组后，线程就会开始转移元素

四大引用

> 强软弱虚
>
> 软引用在内存不够时回收，弱引用在下次GC时只要没有被强引用就可以回收

> 虚引用不影响生命周期，唯一作用是：能在这个对象被回收时收到一个系统通知。

https://learn.lianglianglee.com/%e4%b8%93%e6%a0%8f/Java%20%e5%b9%b6%e5%8f%91%e7%bc%96%e7%a8%8b%2078%20%e8%ae%b2-%e5%ae%8c/47%20%e5%86%85%e5%ad%98%e6%b3%84%e6%bc%8f%e2%80%94%e2%80%94%e4%b8%ba%e4%bd%95%e6%af%8f%e6%ac%a1%e7%94%a8%e5%ae%8c%20ThreadLocal%20%e9%83%bd%e8%a6%81%e8%b0%83%e7%94%a8%20remove%28%29%ef%bc%9f.md

ThreadLocalMap的key 是啥?

> ThreadLocalMap的key 是 ThreadLocal实例（也就是每次代码new的threalocal），而非当前线程号。ThreadLocalMap 是在 Thread 类里的，每次ThreadLocal.get就是获取当前Thread里的ThreadLocalMap，然后get(当前ThreadLocal实例)的值。

ThreadLocalMap的内存泄露问题：手动调 remove 方法

> `ThreadLocalMap` 中使用的 key 为 `ThreadLocal` 的弱引用，而 value 是强引用，value没法被回收，所以用完 `ThreadLocal`后要手动调用`remove()`方法，它的逻辑是获取ThreadLocalMap去remove掉ThreadLocal实例
>
> **public void remove**() { ThreadLocalMap m = getMap(Thread.currentThread()); **if** (m != null) m.remove(this); }
>
> 原文链接：https://javaguide.cn/java/concurrent/java-concurrent-questions-03.html#threadlocal-%E5%8E%9F%E7%90%86%E4%BA%86%E8%A7%A3%E5%90%97

synchronized和ReentrantLock 都是可重入的，有什么区别?

- JVM层 锁和API层锁。
- ReentrantLock多了一些功能：可以选择公平锁（默认跟synchronized一样是非公平的），可以中断而非死等，能判断当前线程是否获取到了锁

AQS

AQS 底层的*数据结构*是 **双向链表**

AQS 的原理是什么?

https://javaguide.cn/java/concurrent/java-concurrent-questions-03.html#aqs-%E6%98%AF%E4%BB%80%E4%B9%88

CAS有什么问题

> ABA、浪费CPU资源

CAS

> Compare and Swap，乐观锁，比较旧值与预期的旧值相同才修改为新值，预计CPU提供能力，跟MySQL的乐观锁 update value = ... where value = 100 一个道理
>
> > 避免ABA问题：加个版本号，1A，2B，3A

问题排查

问题排查调试命令：

栈分析：

```
jstack 2815（线程号，ps得来）
```

堆dump：

```
jmap -dump:format=b,file=/tmp/heap3.bin 2815
```

JVM

Java内存模型

简单讲讲 Java内存模型

内存里包含 虚拟机栈、本地方法栈、方法区、程序计数器、堆，其中堆中又分新生代老年代，新生代里分Eden和俩个Survivor区。

新创建的对象会在Eden区，GC后进Survivor区，每次GC把Eden和Survivor区活着的对象复制到另外一个Survivor区，然后清空，俩个Survivor区来回倒腾。

Eden:Survivor = 8:1:1

需要俩个Survivor的原因：

> GC的时候，Eden区活着的对象复制到Survivor区，如果只有一个Survivor区，会导致多次GC后，那个Survivor区内存碎片很多，因为Survivor区的对象也会有死掉的时候，物理空间就不连续了。

空间分配担保
　YGC前，看老年代有没足够连续空间接受全部新生代，如果有，正常YGC
　如果没有，再看空间够不够以往YGC到老年代的平均值，够就YGC，不够就FGC，给新生代腾空间
空间分配担保的目的是 让老年代有足够空间容纳YGC过来的对象

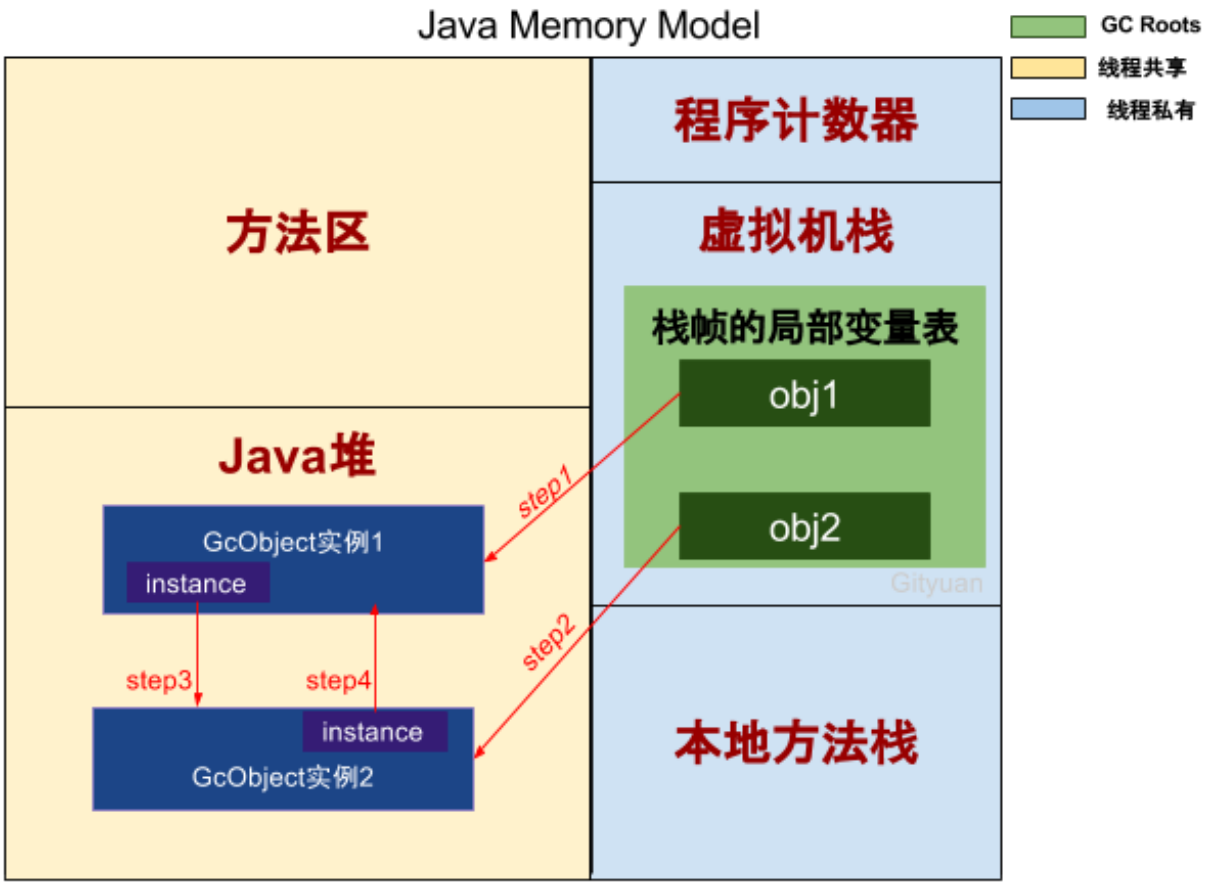
什么情况会从新生代变成老年代

- 大对象（默认3M，可配)直接进
- 新生代GC 15次
- 大于等于某个年龄相同的对象占一半以上，就这个年龄和比它大的都迁老年代

垃圾回收

- 判断对象是否可回收：

> 引用计数法
>
> **可达性分析**：从Gc-Root开始找无法被找到的对象就是可回收的
>
> > GC ROOT都有哪些?
> >
> > 1.[虚拟机] 栈中变量引用的对象
> > 2.[方法区中的] 静态变量和常量引用的对象
> >
> > 3.native方法引用的对象



垃圾回收算法

- 标记-清除
- 标记-整理（标记活着的移动在一边，清掉其他）
- 复制（活的复制到另外一区，大的Eden区和2块小的Survivor区）

分代清理：

- **新生代使用: 复制算法（死得多活的少）**
- **老年代使用: 标记 - 清除 或者 标记 - 整理 算法**

垃圾收集器

CMS 收集器:CMS(Concurrent Mark Sweep)，Mark Sweep 指的是标记 - 清除算法，老年代算法

G1（garbage first） 收集器（JDK 9到现在JDK 16的默认垃圾收集器）： 把堆空间分成多个小块，进行标记-整理，详细看这里

ZGC（在JDK 11 为实验阶段，在JDK 15转正）

Parallel Scavenge收集器其实**就是Serial收集器的多线程版本**，新生代采用复制算法 ;老年代采用标记-整理算法

三色标记法-TODO

设计模式

平常能用上的基本就 单例、工厂、策略、建造者，其他很少见。

工厂：一个父类多种实现，在工厂内写if else选择具体实现。

```
if(shapeType.equalsIgnoreCase("CIRCLE")){
   return new Circle();
} else if(shapeType.equalsIgnoreCase("RECTANGLE")){
   return new Rectangle();
}
```

策略模式：一个父类多种实现，外部传入

```
      Context context = new Context(new OperationAdd());
```

工厂和策略的区别： 工厂里写If,else选择具体实现，策略是外部传入，工厂相当于黑盒子，策略相当于白盒子；

```
public static SingleTon{
  private volatile static SingleTon ton;
  private SingleTon(){}
  public  SingleTon getInstance(){
    if(ton == null){
      synchronized(SingleTon.class){
        if(ton == null){
          ton = new SingleTon();
        }
      }
    }
    return ton;
  }
}
```

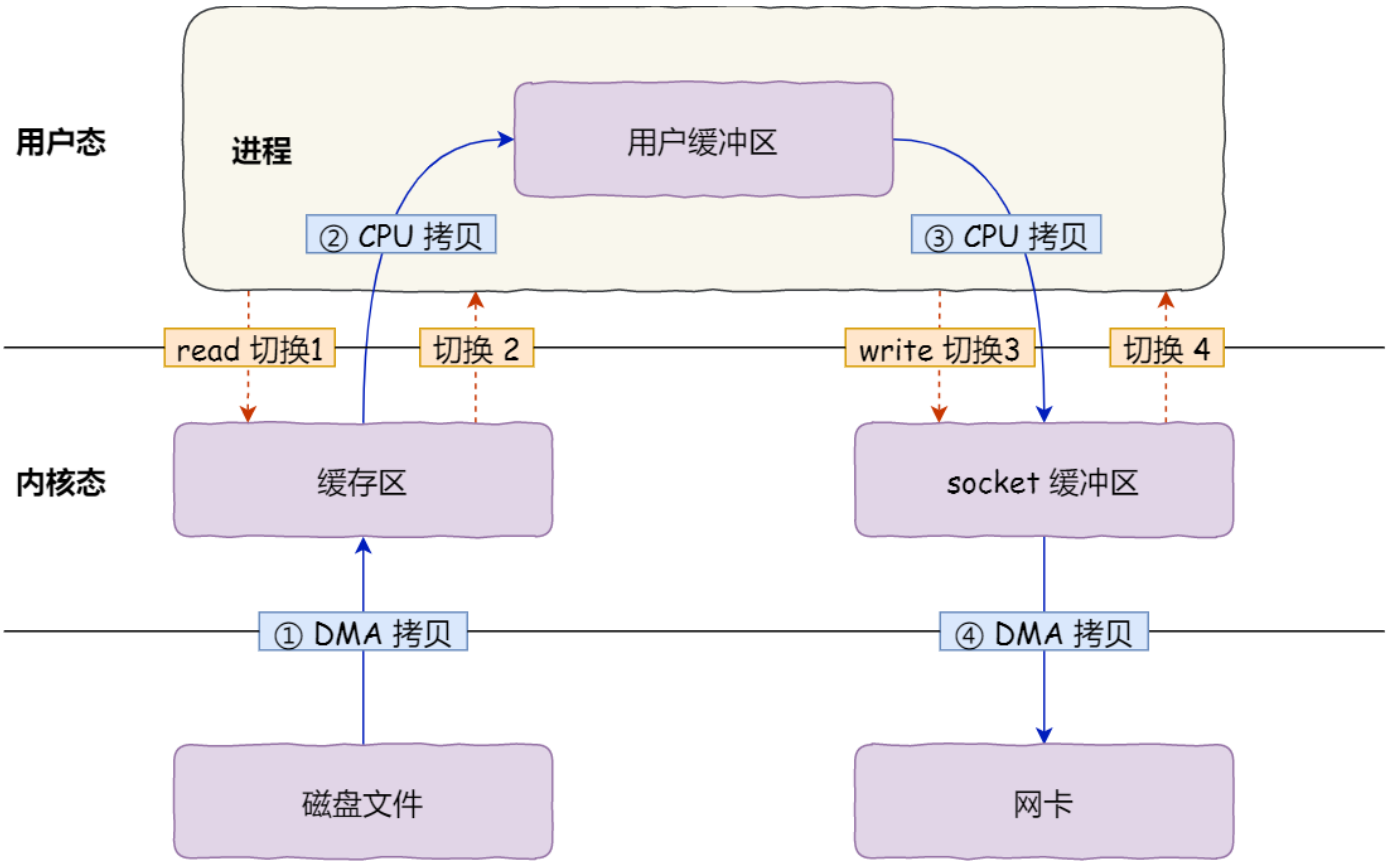
建造者模式 就是Builder

软件开发中的原则 - SOLID

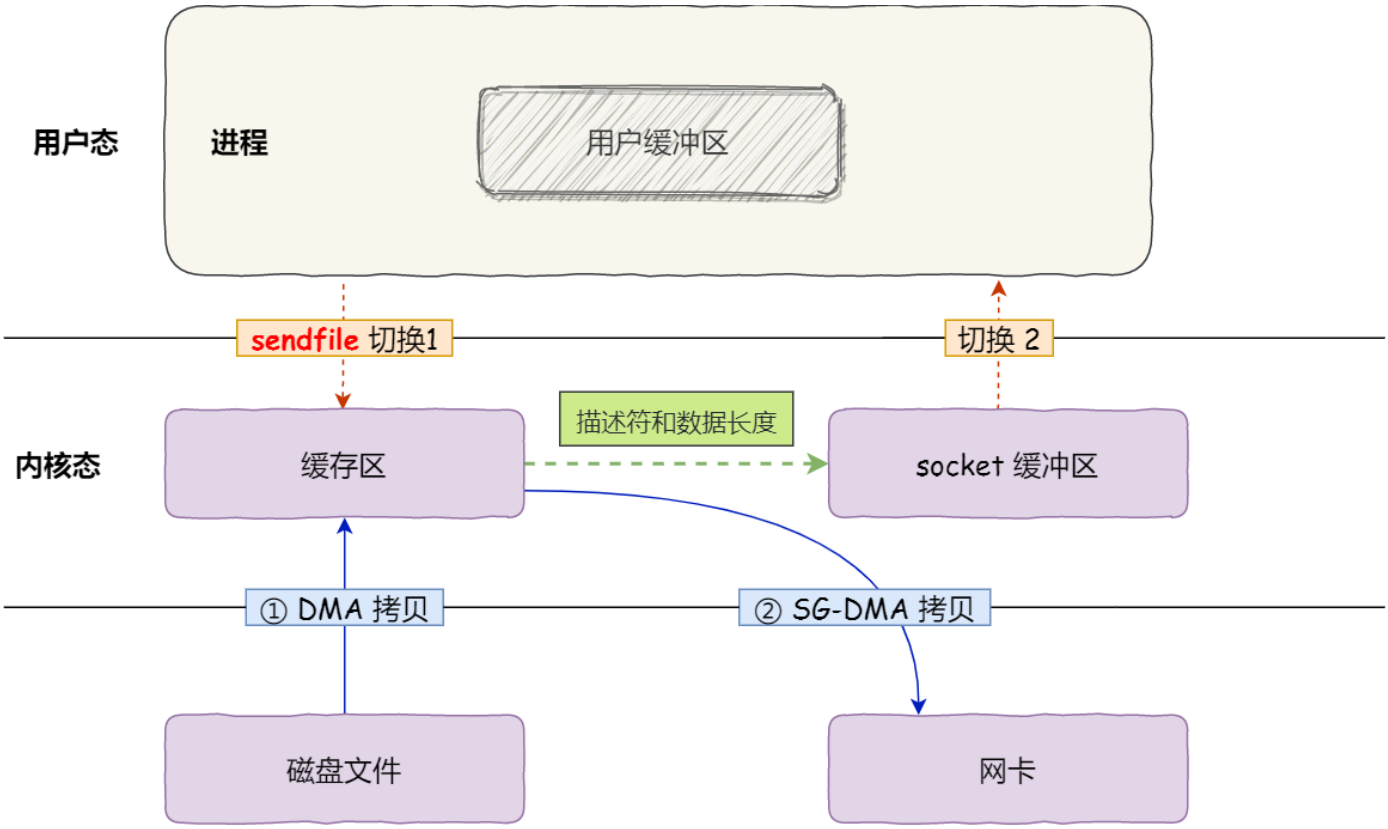
S单一职责SRP
O开放封闭原则OCP：对扩展开放，对修改关闭
　案例：Collections.sort() 方法，这就是基于策略模式，遵循开闭原则的，你不需为新的对象修改 sort() 方法，你需要做的仅仅是实现你自己的 Comparator 接口。
L里氏替换原则LSP：任何基类可以出现的地方,子类也可以出现。
I接口隔离法则：客户端不应该依赖那些它不需要的接口
D依赖倒置原则DIP：要依赖抽象,而不要依赖具体的实现

零拷贝

有零拷贝前：数据从磁盘读取到写入网卡要经过4次拷贝（由DMA复制到内核态的页缓冲区，再复制到用户态的用户缓冲，再复制到内核态的socket缓冲区，再到网卡）



有零拷贝后：通过sendfile函数把数据从磁盘复制到内核态的页缓冲区后，直接copy到网卡缓冲区，全程在内核态由DMA操作，CPU不干预



好文:https://www.xiaolincoding.com/os/8_network_system/zero_copy.html

MQ-kafka

kafka 高性能原因?

- sendfile的零拷贝技术
- 磁盘操作时用的 页高速缓冲区，减少IO。
  - 页高速缓冲区 在磁盘进行标准IO操作时，操作系统内核会先把数据写入到PageCache，这样我读取数据时会直接从Cache中读取同时减少了IO的次数，达到了提升读写效率。
  - 如果 Producer和Consumer 速度差不多，就可以只通过缓冲区解决不用落盘 https://blog.csdn.net/Cecilia3333/article/details/103786654
- 顺序写，不修改已有数
- 批量接收、发送操作
- 数据压缩，增加网络传输效率

Kafka怎么做的高可用性?

【冗余副本+ISR列表+Controller】每个partition的数据冗余到多个borker上，并且leader维护一个10秒内有同步数据的follower列表，叫ISR，并同步到ZK，如果leader挂了，就由担当控制器的borker从follower列表/ISR 里遍历选第一个在列表里的副本当新leader。

而选哪个节点当控制器是靠 集群borker启动的时候，会在zk创建一个临时节点让自己成为控制器，先抢到的是控制器

展开:如果控制器挂了怎么办

如果Controller挂掉或者网络出现问题，ZooKeeper上的临时节点就会消失。其他的Broker通过Watch监听到Controller下线的消息后，继续按照先到先得的原则竞选Controller。这个Controller就相当于选举委员会的主席。

生产者ack机制：acks值为0 代表发了就代表成功，1为leader写入磁盘就成功，all为同步所有follower成功。如果想

Metaq跟kafka区别

metaq优点是解决堆积问题

- 消费消息，kafka是拉模型，metaq默认拉，同时也支持了推模式
- 在选主上，kafaka用的zk，允许从节点变成主，metaq自己搞了个命名服务(NameServer)，主节点做冗余，从节点不能升级为主节点。

MQ推拉模式优缺点

- 推模式快，但不好控制速度
- 拉模式，速度由消费方控制，MQ中间件在设计上不依赖消费方

MQ如何保证消息不丢失

- 生产者开启confirm机制，提交成功会拿到ACK
- MQ本身开持久化
- 消费者使用手动ack

Mysql

基础问题

binlog和redolog的区别

> binlog是逻辑日志，记录增删改查，是MySQL服务器层的
>
> redolog是物理日志，记录页的变更，用于崩溃恢复到最后一次状态，是Innodb特有

MYSQL的JOIN原理

> 俩表JOIN，如果关联key是索引字段，就扫小表，去跟大表的索引进行匹配；
>
> 如果没有，就接近于俩层循环，在扫外层表的时候，批量拉取到缓冲区，让内循环的每一条都能跟外循环的N条进行匹配，减少IO

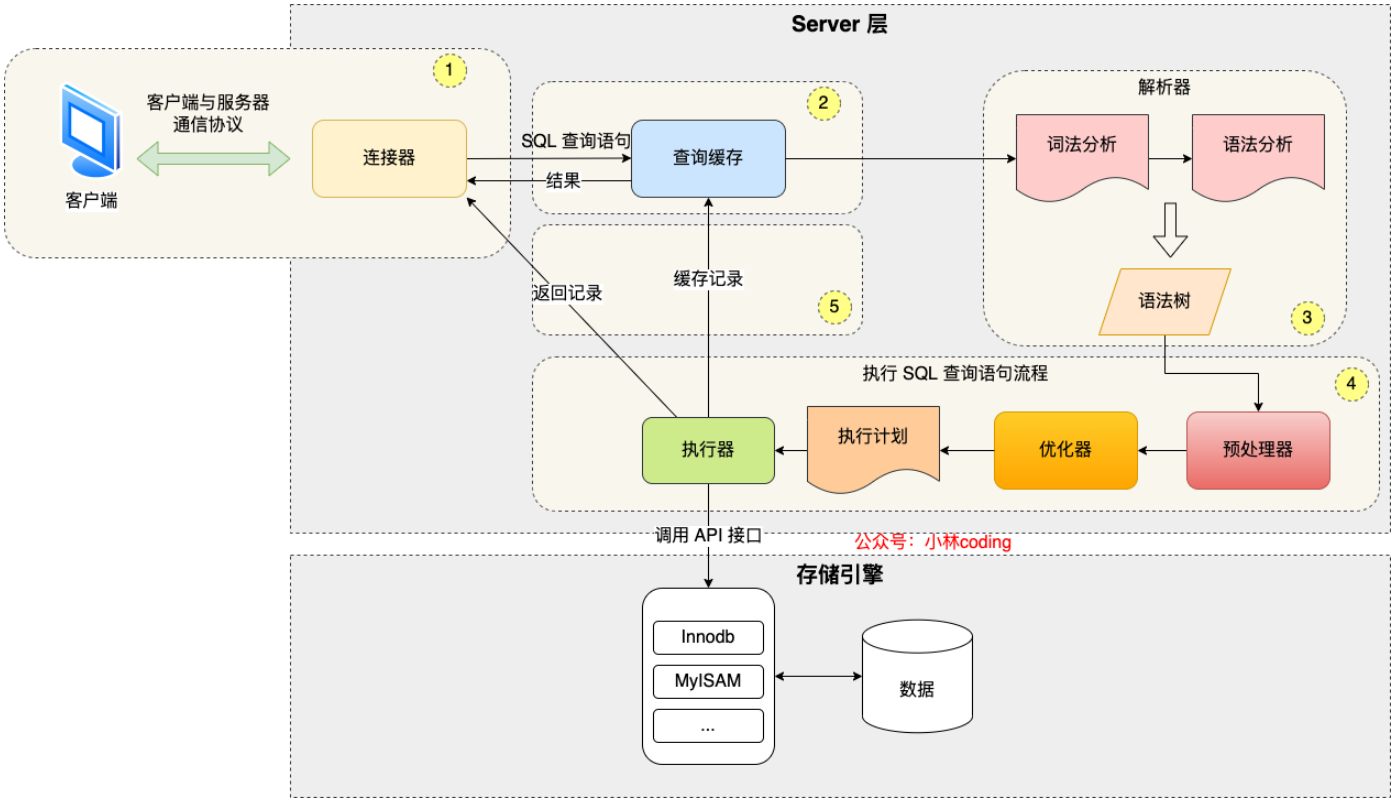
为什么互联网公司选择使用 RC

> Read Repeeatted RR级别有间隙锁，容易死锁

Mysql查询过程

先是「连接器」验证权限，然后查询缓存（8.0后被删除了）没命中就到「解析器」做词法、语法的分析，再是SQL优化并产出执行计划，最后是「执行器」与存储引擎进行交互，查询数据并写缓存

如果是Update，没有查缓存操作，但会清除这个表的全部缓存，同时多了记录binlog(server层)和redolog（inndo)的操作



索引

[MySQL索引背后的数据结构及算法原理]

MySQL使用B+树，而非B树

> 主要区别是非叶子节点存不存数据，B 树数据在各节点上，b+树只在叶子节点
>
> 带来的区别：
>
> B+树每次都查到叶子节点，时间复杂度固定logn， B-树的查询速度不稳定，跟位置有关，可能是O(1)
>
> B+树可以利用空间局部性加上前后指针做区间查找
>
> B+树可以把索引和数据分开存储

MySQL不用跳表而用B+树

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

Mysql不用二分查找树

> 二分查找树是二叉树，相同节点数下树高更高，也就是磁盘IO更多，并且还有退化成链表的可能

MyISAM 和InnoDB 引擎区别

> InnoDB 行级锁，支持事务，底层数据结构不同是 InnoDB 主索引的叶子节点存放数据，而MyISAM存数据地址，也导致了InnoDB在文件存储上，数据和索引存在同一个文件，而MyISAM是俩个文件

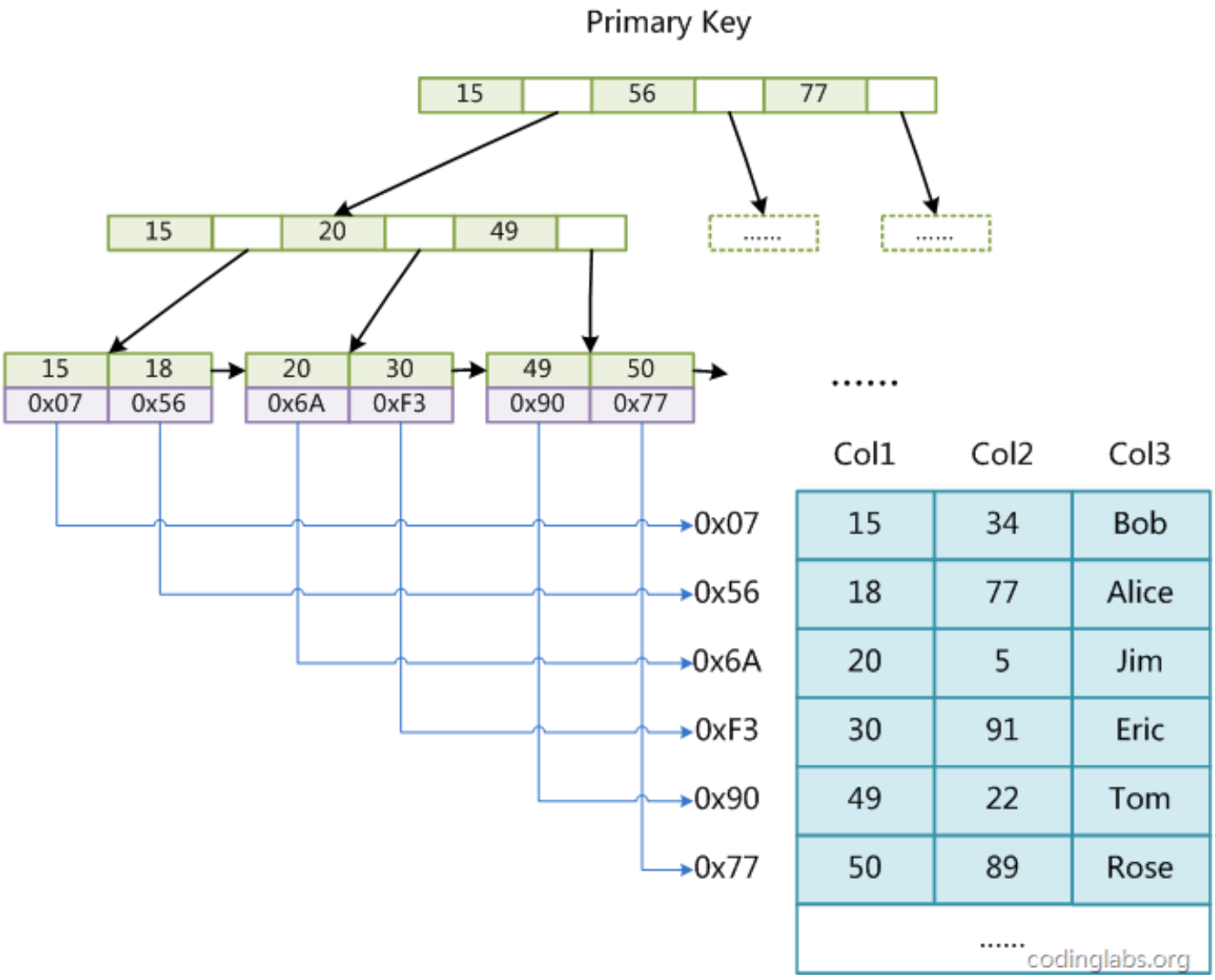
聚集索引和非聚集索引：

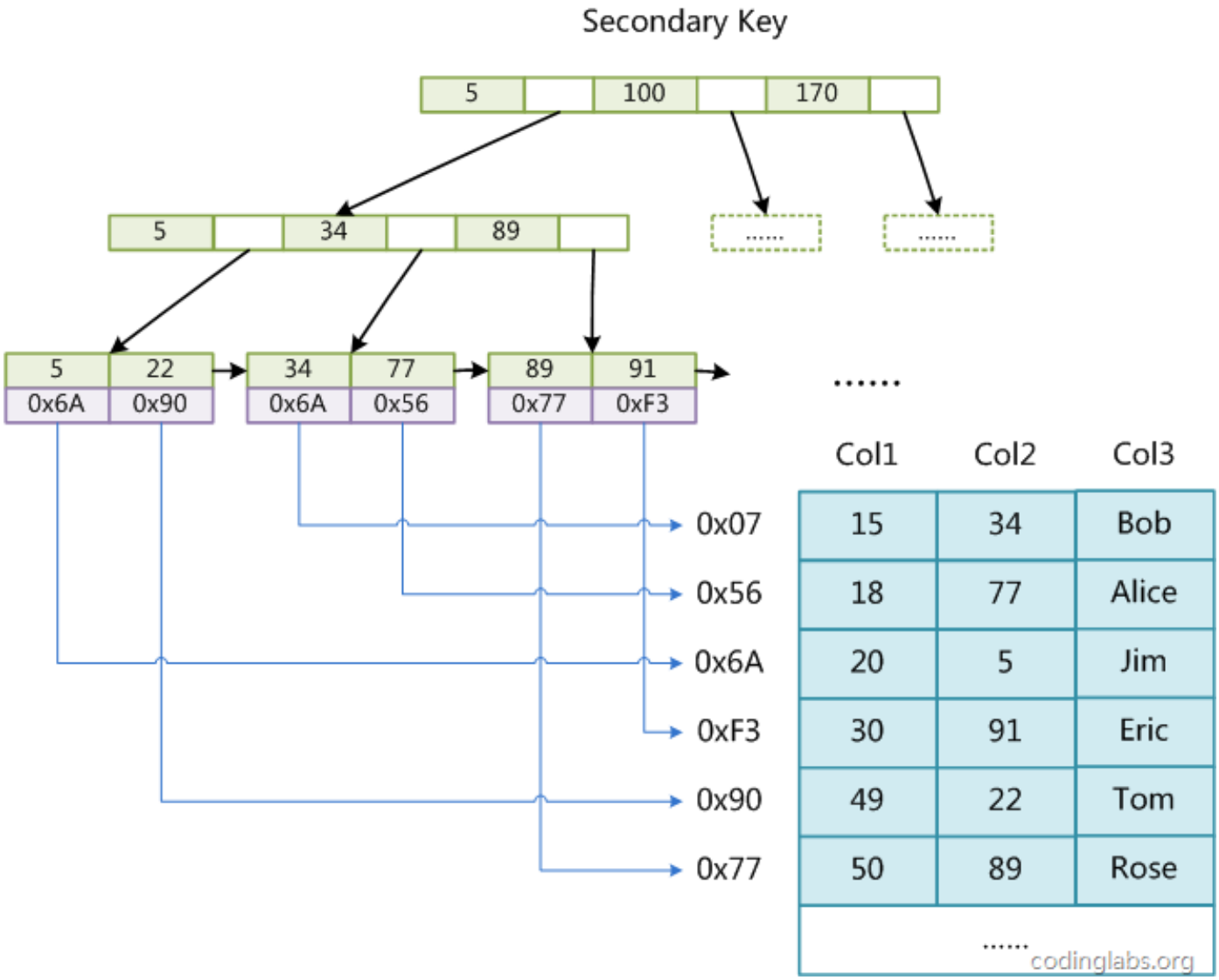
> 物理存储顺序和逻辑顺序一样叫聚集索引，方便区间查询，比如inndo，不一样叫非聚集索引，比如myasim叶子节点存的物理地址，不要求顺序

数据结构

MyISAM：

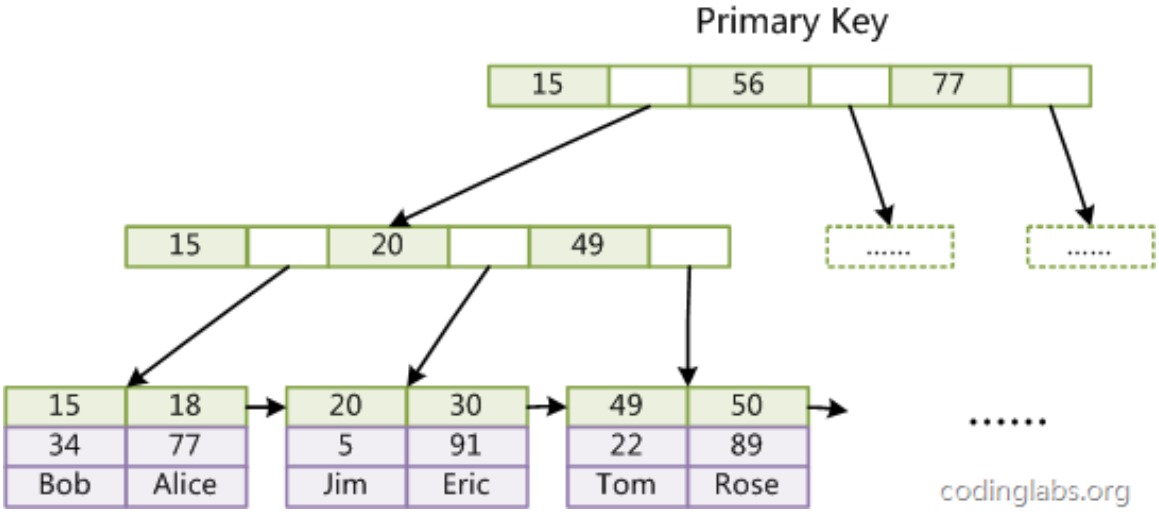
主索引和辅助索引都是叶子节点存数据所在的地址



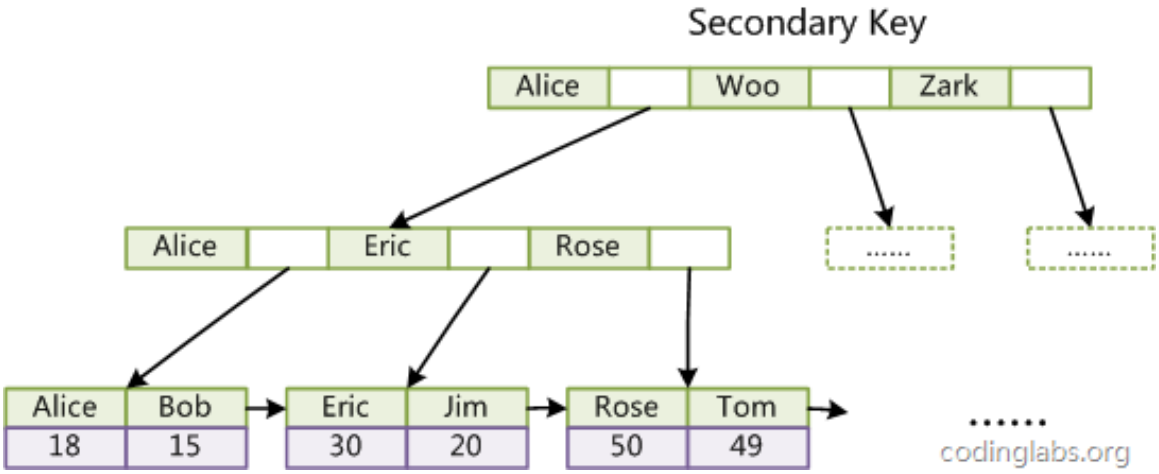


InnoDB：

主索引：叶子节点存储完整value，表数据文件本身就是一个按B+树组织的索引结构



二级索引：叶子节点存储主键，所以都需要查俩次



最左前缀原理

为什么会存在最左前缀?

> 因为联合索引在B+树里是多字段一起存储的，**最左边的字段总是排好序的**，所以不带最左边字段查询就无序，得扫全表了
>
> 所以(a,b,c)遇到a=x and b >1 and c=3 的区间查询时，会用到(a,b)

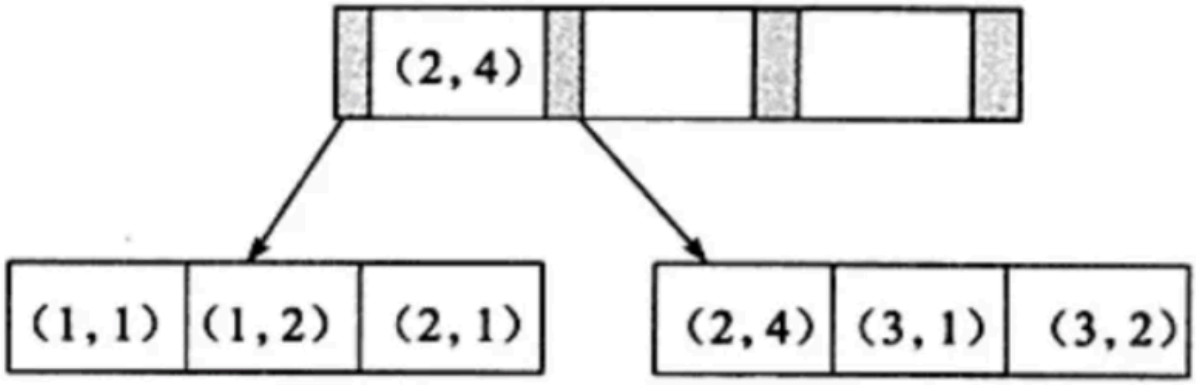


图 5-22　多个键值的 B+ 树

其他

索引选择度

> 等价于不重复值的占比，越大越好
>
> ELECT count(DISTINCT(title))/count(*) AS Selectivity FROM employees.titles;
>
> +-------------+
>
> | Selectivity |

> +-------------+
>
> | 0.0000 |

SQL中字段的顺序不影响索引生效

> 因为会被优化器优化掉，所以顺序不影响索引生效情况

主键用ID字段

> 是因为id能保证有序，减少移动数 据去插入

假如查询 A in (), MySQL 是针对 N 个值分别查一次索引, 还是有更好的操作?

> 不知道，有了解的同学可以留言 (补充, @BillyLu 贴出了文档 equality-range-optimization, 大意是对非唯一索引 MySQL 会使用 index dive 的方式估算这个 range index 涉及的行数, 结合where optimization 中说明的在走 index 时假如涉及行数过多会走 full table scan, 那么假如 estimation 认为这次 IN 不够好, 是会走全表扫描的. 不知道除此之外, 面试官还有没有想考察的点)
> https://dev.mysql.com/doc/refman/8.0/en/range-optimization.html#equality-range-optimization

事务

事务四大特征：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）

一致性的含义：事务前后要满足定义的规则，符合所有约束、级联、触发器。（来自英文的维基百科），跟什么转账前后前后数据一致无关，那是业务实现的问题，不是数据库该干的。

```
In database systems, consistency (or correctness) refers to the requirement that any given database transaction must change affected data only in allowed ways. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof.
```

脏读：在一个事务处理过程里读取了另一个未提交的事务中的数据

不可重复读：前后读到的数据不一致，针对update

幻读：多次读取的数据总量不一致，读到了其他事务新增的数据 ，针对 insert/delete

隔离级别：未提交读（READ UNCOMMITTED）、提交读（READ COMMITTED）、可重复读（REPEATABLE READS）、可串行化

隔离级别	脏读	不可重复读	幻读
未提交读	可能发生	可能发生	可能发生
提交读	-	可能发生	可能发生
可重复读	-	-	可能发生
可序列化	-	-	-

好文：https://zh.wikipedia.org/wiki/%E4%BA%8B%E5%8B%99%E9%9A%94%E9%9B%A2#.E9.BB.98.E8.AE.A4.E9.9A.94.E7.A6.BB.E7.BA.A7.E5.88.AB

MVCC

好文：https://zhuanlan.zhihu.com/p/52977862

原理

多版本并发控制（MVCC） 是可重复读（REPEATABLE READ） 和 提交读（READ COMMITTED）中，通过保存每条记录的快照来 实现不加锁的读写并发。 实现方案是在每行里添加隐藏字段 当前事务ID、回滚指针、软删除标识 （可以不提）。

readview：是一个快照状态，包含活跃的事务ID列表，活跃事务里最小的事务ID、系统中最大事务ID+1（不等于活跃里最大+1）

trx_ids: 当前系统活跃(未提交)事务版本号集合。
low_limit_id: 创建当前read view 时“当前系统最大事务版本号+1”。（这俩没写反，大于它的不要，只查在Up和low中间的，跟活跃事务匹配）
up_limit_id: 创建当前read view 时“系统正处于活跃事务最小版本号” （这俩没写反）
creator_trx_id: 创建当前read view的事务版本号；

> 查询时只查事务ID小于最小活跃ID 或者等于自身的记录，（事务开始时已提交的或者自身）这些是可见的。
>
> 如果是大于等于最小活跃ID，再看是否在活跃列表里，再就说明是未提交的事务，不可见但可以通过回滚指针找到可见的版本（小于活跃ID），不在列表里就代表已经提交了，要保留。
>
> 如果大于最大事务ID，则说明是创建read view后新增的事务，不要。

> 更新时把原数据copy到undoLog，然后修改数据并把回滚指针指向undoLog那条记录，如果有多次修改就用链表头插法链到最后修改的undolog

> （可以不提）删除时做软删除，再单独开线程去删 标记为软删除，并且事务ID小于系统中最老的活跃事务ID的记录，这些记录是可以安全删除的。

- undo log用于事务回滚，也用于MVCC，记录的是 修改前的值，而binlog记录修改后的值
- MVCC解决读写冲突，写写冲突则需要靠乐观、悲观锁
- 另外俩个事务隔离级别不适合，因为

  未提交读（READ UNCOMMITTED），总是读取最新的数据行，而不是符合当前事务版本的数据行。

  可串行化（SERIALIZABLE） 则会对所有读取的行都加锁

RC,RR级别下的InnoDB快照读有什么不同?

RR 在同一个事务的第一次读时生成ReadView，后续读取都使用同一个ReadView，所以一个事务里看到的数据一样-可重复读

Read Committed 在同一个事务每次读时都生成事务ID，所以能看到其他事务的最新提交

好资料：https://pdai.tech/md/db/sql-mysql/sql-mysql-mvcc.html

默认隔离级别 RR可重复读 解决幻读问题了么?

没有

- 针对普通 select 语句，通过 MVCC **快照读** 解决了幻读。
- 针对select ... for update,lock in share model 这种当前读语句，每次读最新数据，是通过 间隙锁解决幻读的。

锁相关

不想看，好资料：https://xiaolincoding.com/mysql/lock/deadlock.html

> - 共享锁(S):select * from t1 where ... lock in share mode; 其他事务可以读，但不能写
> - 排他锁(X):select * from t1 where ... for update;

分表分库

哈希、按数据范围、按月份分

月份划分：数据分布不均匀，也容易在某个月单点一直查询，并且不好做跨月查询，每年也要记得新建表，适合做日志存储这种场景。

数据范围：数据分布均匀，容易有热点问题

哈希划分：http://blog.itpub.net/69912579/viewspace-2839030/

还可以在订单ID上存储 所在库表，但没法扩展

中间件：TDDL、MyCat、shardingsphere

其他已知内容

索引优化：explain　　CPU排查： show processlist 和 kill

给大表加索引：建相同结构的新表，进行表结构变更，通过触发器或者DRC组件复制操作期间的数据到新表，然后一个RENAME命令给俩个表换名字，让新表接流量，删除老表

归档方案

分表分库

读写分离：从库设置只读不写

Redis

基础问题

redis 为什么快

> 纯内存
>
> 单线程减少了上下文切换
>
> IO多路复用（epoll 复用同一个线程，实现了一个线程可以监视多个文件句柄；一旦某个文件句柄就绪，就能够通知应用程序进行相应的读写操作；而没有文件句柄就绪时,就会阻塞应用程序，交出cpu）

Redis 与 Memcached 区别：

> Memcached 纯内存，没有持久化
>
> Memcached 原生不支持集群模式
>
> 数据类型上，Memcached只有KV类型，Redis支持Set Hash等
>
> Memcached没有发布订阅、事务、 lua等功能

五种数据结构

> List列表（**双向链表或压缩列表**）、String字符串、Hash（**压缩列表或哈希表**）、Set集合、SortedSet有序集合

SortedSet：元素少（少于128个且元素长度小于64字节）的时候用压缩列表(ziplist)，多的时候就用 哈希表+跳表一起用，哈希方便ZSCORE查成员的分，跳表方便范围查询

压缩列表ziplist：为了节省内存做的一个字节数组（避免了链表的指针存储），数组里每个元素会保留上一个元素的长度，进而实现往回遍历。压缩列表包含 总长度、节点数、最后个元素的偏移量、节点列表、结尾标记

跳表

Zset有序集合的底层结构，是一个多层的有序链表，第一层就是正常链表，插入的时候概率性提升一层，上面每层都会跳过几个节点，查询时从上往下尝试，尽可能跳过更多节点，达到了类似二分的效果。

> Redis概率是1/4提升，最多32层，每个跳跃表节点的层高都是1至32之间的随机数
>
> 不用平衡树/B树，是因为可以范围查询，可以实现简单，而且因为是内存所以不用像MySQL一样考虑层高

压缩列表ziplist

为了节省内存的一个连续型结构，类似于数组

结构里包括 压缩列表总字节数、最后个节点的偏移量、节点数量、节点列表、结束分隔符

其中每个节点会存储上个节点的长度，再配合最后个节点的偏移量，就可以做到直接跳到最后个节点，并且倒序查询

HyperLogLogs 基数统计:

> 举个例子，A = {1, 2, 3, 4, 5}， B = {3, 5, 6, 7, 9}；那么基数（不重复的元素）= 1, 2, 4, 6, 7, 9； （允许容错，即可以接受一定误差）
>
> 这个结构可以非常省内存的去统计各种计数，比如注册 IP 数、每日访问 IP 数、页面实时UV、在线用户数，共同好友数等。

Bitmap：

> 统计用户信息，活跃，不活跃！ 登录，未登录！ 打卡，不打卡！ **两个状态的，都可以使用 Bitmaps**！
>
> 如果存储一年的打卡状态需要多少内存呢？ 365 天 = 365 bit 1字节 = 8bit 46 个字节左右

geo底层的实现原理实际上就是Zset,

Redis 是单线程吗?

> 答：处理请求的是单线程，但在RDB、AOF重写、过期删除等情况下会开后台线程
>
> 主线程是单线程，官方说法是 因为Redis是纯内存操作，CPU 并不是制约 Redis 性能表现的瓶颈所在，如果要多线程可以多开几个节点或者走集群方式

> 不过6.0以上版本也开了多线程处理网络IO，但命令执行还是单线程。

先更新数据库，还是先更新缓存?

> 答：简单做就 先更新数据库再删缓存 ，复杂做就订阅MySQL的DRC变更去写数据库，业务请求时不修改缓存。
>
> 可能的问题：「先更新数据库， 再删除缓存」其实是两个操作，有可能缓存删除失败，需要给删除操作加重试（消息队列可以）
>
> 为什么不更新缓存：高并发下依然会因为先后顺序而不一致

管道和事务都是一次性一批请求过去，有什么区别

> 管道： 客户端行为，目的是减少IO次数，适合多个请求之间没有先后依赖的情况，一次性打包多个请求给服务端，服务 端一条条执行，会缓存执行结果一起返回，期间可能有其他命令插入
> 事务： 服务端行为，会把请求丢到队列里，返回queued给客户端，最后一次性执行，执行期间不会有其他命令插入
> 一批请求

Lua脚本跟事务的区别：

> 事务的每条命令之间是独立的，后面的步骤没法依赖前面的 结果，但脚本可以。
>
> 根据redis.io 说法：脚本也是一种事务，引入时间比事务晚，事务能做的脚本也能做，但短期不打算废弃事务，因为事务提供了不用脚本的方式，除非未来用户都只使用脚本，那官方会考虑废弃事务

事务满足一致性吗?

> Redis事务只检查语法，不检查执行结果，在有语法错的情况会拒绝所有命令执行，但在执行错误下会跳过该命令，其他命令继续执行，所以不满足一致性。
>
> 对此的解法：通常来说要提供回滚功能，但事务不支持回滚，因为官方认为 1.这是代码BUG 2. Redis应该简洁。
>
> 所以折中方案是 在事务开始前watch你想执行的键值看有没变化进而停止执行

Hash扩容

hash怎么扩容

> 字典里有俩个哈希表(dictht)，平常只用一个，rehash的时候给另外一个扩容一倍，然后开始渐进性rehash。期间读操作要先读老表(ht[0]，没有再读新表(hc[1])，但新增key只在新表写。
>
> 渐进性rehash具体指啥：
>
> Redis的哈希存储是数组+头插法链表，字典(ht[0])里有一个变量记录head数组迁移进度，每次读写操作时会顺带把这个下标对应的head数组后面的数据迁过去，这个数组后链表数据迁完了就加一。慢慢等迁完

过期删除

延迟删除+定期删除俩种：

key过期后不会立马删除，会在下一次读写时 读取一个过期字典(O(1)) 看是否过期，过期则删

也有定期删除：每隔一段时间（10秒）从数据库随机取一部分key(20个) 判断过期就删除

> 详细策略是 取20个key判断删除后，如果没超时25毫秒并且实际过期数量超过5个（1/4），就继续取20key直到超时

缓存雪崩、击穿、穿透

缓存雪崩：大量key同时过期。　　解决：加随机数 或者 设为无限期，通过消息队列或者定时任务去更新删除

缓存击穿：热点key过期　　解决：热点key不设过期时间。 或者我想的拆key

缓存穿透：不存在的key被访问。　　解决：缓存空值、bloom过滤判断不存在就不查了

数据持久化

介绍AOF：Redis数据持久化的增量方案，执行完写命令后把命令写到AOF缓冲区，然后每秒一次写到磁盘上。如果AOF 日志过大会触发文件重写，fork个子进程去扫现在的数据库键值，生成命令写进日志（期间新增的写入命令会写到AOF重写缓冲区[这里也同时双写了AOI缓冲区]，重写完成后追加进去）

> Tips-用进程而非线程：父子进程共享副本，copy on write机制，如果是线程就是共享内存数据，但对于写操作还得加锁

介绍RDB：Redis数据持久化的全量方案，bgsave fork个子进程去把当前内存快照写到文件里（二进制方式，具体dump方案书和博文没写），除了人工执行还可以配置每几秒内有多少写操作就自动执行

还原优先级：还原的时候，同时开了AOF和RDB的情况，优先使用AOF，因为数据新

混合持久化：Redis4.0开始的，在**AOF重写**（注意是重写）过程中把先把当前内存数据以RDB格式写入AOF文件，再把期间的增量命令（AOF重写缓冲区）用AOF形式追加进文件。所以结果文件前半段是RDB格式，后半段是AOF格式。

面试题-大key对持久化的影响

> AOF的持久化策略选每秒就没影响，因为是异步的，选always就阻塞时间变长
>
> 在AOF重写中，fork进程和CopyOnWrite的阻塞时间变长

Redis 主从集群

主从复制/主从同步怎么实现的?

分全量和增量

1. 全量复制：用于从库首次连接

   从库发PSYNC命令给主库（包含偏移量「问号」/主服务器运行ID），主库调 bgsave 保存rdb文件，同时把保存期间的新写命令存到缓冲区，然后把rdb文件和缓冲区命令分别发给从库（从库加载完rdb后发一个确认消息，主库再传输缓冲区的命令）。同步完成后，双方维护一个TCP连接来传输写命令

   为了避免给主服务器多个从服务器同步RDB数据占用网络宽带问题，主同步完一个从后会让从库作为其他库的主库去同步(replicaof <目标服务器的IP> 6379)，分散主库压力

2. 增量复制（PSYNC）：用于从库挂了的情况

   主库会在平常同步过程中维护一个最近同步命令的缓冲区（环形队列默认1M），如果从库挂了再恢复，从库发一个PSYNC命令和偏移量给主库，主库判断缺的数据在缓冲区里就把缓冲区数据增量同步过去，否则走全量修复。

   同时每秒也有心跳（REPLCONF ACK 参数偏移量），心跳时从会把偏移量给主，也能在命令丢失的情况主增量复制。

主服务器运行ID：在全量复制后主库发给从库的，增量过程中如果当前主库ID跟从库发来的ID不一样，说明主库发生过变化，需要重新全量同步

相关资料：https://xiaolincoding.com/redis/cluster/master_slave_replication.html#%E5%A2%9E%E9%87%8F%E5%A4%8D%E5%88%B6

怎么判断 Redis 某个节点是否正常工作?

> 相互ping-pong 心跳，在集群中有半数节点对某个节点无响应就认为该节点掉线了
>
> 主节点每10秒会给从节点发ping命令，从节点除了回pong，也会每秒给主节点上报自己的主从复制偏移量，用于检测数据丢失情况

主从复制中，过期key如何处理?

> 主库发送del命令发送给从库

脑裂

脑裂：主节点与多个从节点之间的网络异常，哨兵机制认为主节点挂了就重新选举主节点，等主节点恢复后，出现俩个主节点，此时会把原主节点降为从节点做一次全量同步。

脑裂丢数据：虽然主节点和从节点之间的网络异常，但主节点和客户端中间网络正常，于是客户端正常写入，在主库被降为从库后全量同步会丢数据。

减少脑裂丢数据的方案：

配置 主节点最少1个从节点 或者 主从延迟不超过N秒，否则主节点禁写

ZK没有脑裂问题：因为zk有半数投票，就变成要么只有一个主节点，要么没有主节点，而redis的主从选举，是靠哨兵leader选 人工优先级高、复制延迟小、运行服务ID小的机器选的，没有经过投票。

哨兵(用于主从)

哨兵通常跟主从部署一起用

哨兵节点主要负责三件事情：**监控、选主、通知**。

哨兵通常也是集群部署，多个哨兵节点来避免误判。

如何判断主节点真的故障了?

> 哨兵会每隔 1 秒给所有主从节点发送 PING 命令，要求收到回应。
>
> 对于主节点会通过多部署几个哨兵，在一个哨兵认为主下线后咨询其他哨兵来投票判断主节点(赞成票大于配置项)是不是真下线了

哨兵中，谁去进行主从故障迁移?

> 发现主节点下线的哨兵都会发起投票，看谁拿到半数以上，成为leader
>
> 其实有延迟
>
> 延迟公式：DELAY = 500ms +random(0 ~ 500ms) + SLAVE_RANK * 1000ms

主从故障转移的过程：

> 主从故障转移操作包含以下四个步骤：
>
> - 第一步：在从节点中，根据复制偏移量最大和运行ID最小的规则，选一个从节点升为新主。
> - 第二步：给其他从节点发slaveof命令，认新主为主节点，开始数据同步
> - 第三步：将新主节点的 IP 地址和信息，通过「发布者/订阅者机制」通知给客户端；
> - 第四步：继续监视旧主节点，当这个旧主节点重新上线时，将它设置为新主节点的从节点；
>
> **https://xiaolincoding.com/redis/cluster/sentinel.html#**

Redis Cluster（分布式存储）

数据分给N个节点存

通信协议：Gossip协议

https://z.itpub.net/article/detail/61F53FE9A1487984804DB48AAB9EF262

redis冷热分离

> 基于key访问次数(LFU)的热度统计算法识别出热点数据，并将热点数据保留在redis中，对于无访问/访问次数少的数据则转存到SSD上，如果SSD上的key再次变热，则重新将其加载到redis内存中。
>
> ssd可以用 rocksdb结构存

网络

TCP/IP 网络模型有哪几层?

> 应用层（HTTP/FTP）、 传输层（TCP/UDP/KCP）、网络层（IP协议，增加IP）、网络接口层（MAC地址，增加帧头帧尾）
>
> 传输层在 应用数据 上加TCP头，网络层在这个基础上加 IP头，网络接口层再加帧头帧尾

HTTP

HTTP状态码

> 200 OK　　204 No Content，响应里没有body数据　　206 Partial Content
>
> 301 永久重定向　302 Found 临时重定向　304 Not Modified
>
> **400 Bad Request**　401 Unauthorized　**403 Forbidden**　**404 Not Found**
>
> **500 Internal Server Error**　**502 Bad Gateway**　**503 Service Unavailable**

HTTP头的 Keep-Alive 和 TCP 的 Keepalive

> HTTP头的Keep-Alive（HTTP长连接）是复用同一个TCP连接，能避免每次都建立TCP连接
>
> 附带好处：同时能做到客户端连续发送多次请求而不用等待服务端返回
>
> 连接浪费问题：nginx 这种web服务提供参数(keepalive_timeout)可以设置超时时间（nginx默认75秒），这段时间内没有未没有请求就断开TCP连接

WebSocket

> TCP是全双工的，但HTTP是半双工，**同一时间里**只能有一方请求，WebSocket可以双方发请求，适合网页游戏场景

RPC和HTTP

> **RPC 不算是协议，而是一种调用方式**，目的是希望程序员能像调用本地方法那样去调用远端的服务方法，而像 gRPC 和 Thrift 这样的具体实现才是协议

扫码登录

- 方案1：前端每秒轮询问后端有没有人扫过码 应用方：微信公众号　缺点：扫码后用户要等1、2秒
- 方案2：长轮询，一次HTTP请求30秒才返回　应用方：百度网盘 用户体验好，技术上没有明显缺点

TCP

TCP 是**面向连接的、可靠的、面向字节流**的传输层通信协议。

> 连接： 用于保证可靠性和流量控制维护的某些状态信息，这些信息的组合，**包括 Socket、序列号和窗口大小称为连接**
>
> *Connections: The reliability and flow control mechanisms described above require that TCPs initialize and maintain certain status information for each data stream. The combination of this information, including sockets, sequence numbers, and window sizes, is called a connection.*

TCP保证可靠

> checkSum校验（在header里）、ACK、超时重传
>
> 其中滑动窗口和拥塞控制不算

TCP 和 UDP 区别

> TCP三面向连接的可靠的协议，反过来就是UDP，UDP不用建立连接直接丢数据，也没有拥塞控制、流量控制、ACK

如何理解面向字节流?

> TCP 是面向字节流的，UDP 是面向报文的。
>
> 进程发送UDP数据时，操作系统不会做拆分，每次发送和收到的就是一个用户消息。
>
> 发送TCP数据会被拆分，拆多少需要看滑动窗口、拥塞控制等。
>
> 拆分是为了TCP的可靠传输

如何解决TCP粘包

> 加分隔符 或者 在消息前加长度字段、或者固定长度

如何唯一确定一个 TCP 连接呢?

> 四元组：源地址、源端口、目的地址、目的端口

有一个 IP 的服务端监听了一个端口，它的 TCP 的最大连接数是多少?

> 服务端取决于 Linux文件描述符数和内存，文件描述符数默认最大65535，可以用 ulimit 改为无限

TCP什么情况下，会发RST包?

> 端口不存在
>
> 应用程序想提前断开连接
>
> 在已关闭的TCP连接上还收到包会返回RST

TCP和UDP可以共用一个端口么?

> 可以，UDP没有监听(listen)的过程，只有绑定(bind)，收到数据后根据协议的头部判断是TCP还是UDP
>
> 但TCP和TCP之间不行，会在监听listen阶段报IP+端口重复（Address in use）

重启 TCP 服务进程时，为什么会有“Address in use”的报错信息?

> TCP四次挥手，客户端最后有一个俩倍MSL的**TIME_WAIT**时间，这段时间端口还占用，多等一会就行

服务端如果只 bind 了 IP 地址和端口，而没有调用 listen 的话，然后客户端对服务端发起了连接建立，会怎样?

> 拒绝连接，服务端会回 RST 报文

DNS正常情况下用UPD， 什么时候用TCP?

> 1. DNS响应报文长度超过UDP报文长度512字节
> 2. DNS主副服务器之间的数据同步（叫 区域传送）

三次握手

客户端发 SYN报文想建立连接；服务端回ACK和SYN； 客户端回ACK

为什么需要三次?

> 俩次不够，四次太多
>
> 在不可靠的通道中要确保可靠，起码双方都得拿到对方的一次回应，所以最少3次，并且这样有好处是能防止客户端过时的连接传到服务端后，浪费资源（话实际上客户端已经不用的连接 在网络中到了服务端，但服务端并不知道于是空连接）

三次握手丢了会怎样?

> 第一次握手丢了，客户端会重试(5次)，不成功就断开连接
>
> 第二次握手丢了， 因为同时包含ACK和SYN，所以俩边都会重传：服务端重传SYN+ACK，客户端会重传第一次握手(ACK报文是不会重传的，会由发送方重传）。
>
> 第三次握手丢了，服务端会认为是客户端没收到，所以重传第二次握手的ACK+SYN，所以服务端会重传

四次握手

客户端发 FIN报文；服务端回ACK确认，等数据处理完成后发 FIN ； 客户端回ACK

为什么需要四次?

> 因为服务端收到第一次挥手后回应ACK，此时可能还有数据还没发完，所以需要等处理完才发FIN，这里没法合并，所以变成了四次

为什么 TIME_WAIT 等待的时间是 2MSL?

> 在挥手第四步ACK丢失的情况下，服务端会重传FIN，客户端收到后要ACK，就是俩次网络交互了（最大生存时间）

如果已经建立了连接，但是服务端的进程崩溃会发生什么?

> 比如进程被kill了，因为内核要回收进程的资源，所以会代替进程向客户端发送挥手FIN报文，开始四次挥手

TCP重传

- 超时重传
- 快速重传：有丢失的情况接收方的ACK还是上一次的，发送方看到连续三次对一个序号的ack就重传对应序号的数据
- SACK、DSACK（不想看） https://xiaolincoding.com/network/3_tcp/tcp_feature.html#超时重传

流量控制/滑动窗口

流量控制：**为了控制发送方发送速率，保证接收方来得及接收**

接收方告诉发送方自己还有多大的缓冲区（这个缓冲区是操作系统提供的），在这个缓冲区内可以没有收到ACK也继续发。

> 另外发送方也会有窗口记录哪些是已发送已接ACK，哪些是还没有ACK的，跟着移动

长文链接：https://xiaolincoding.com/network/3_tcp/tcp_feature.html#%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%E7%BC%93%E5%86%B2%E5%8C%BA%E4%B8%8E%E6%BB%91%E5%8A%A8%E7%AA%97%E5%8F%A3%E7%9A%84%E5%85%B3%E7%B3%BB

拥塞控制

口语：拥塞控制是控制每次发送多少数据包，避免网络本身堵的时候加堵。

> 发送方 维护一个拥塞窗口值代表每次发几个TCP包。
>
> 刚开始慢启动，每次只发一个包，在收到ACK后拥塞窗口值*2，下次就发2个，4个，8个成倍递增。
>
> 到了慢启动阈值后进入拥塞避免阶段，每次只加 窗口值分之一1，变成线性增长。
>
> 如果又遇到拥堵（也就是触发了超时重传）就进入快速恢复阶段，窗口值变为一半，再次进入拥塞避免的线性增长。

那么怎么知道当前网络是否出现了拥塞呢?

> 「发送方」没有在规定时间内接收到 ACK 应答报文，也就是**发生了超时重传，就会认为网络出现了拥塞。**

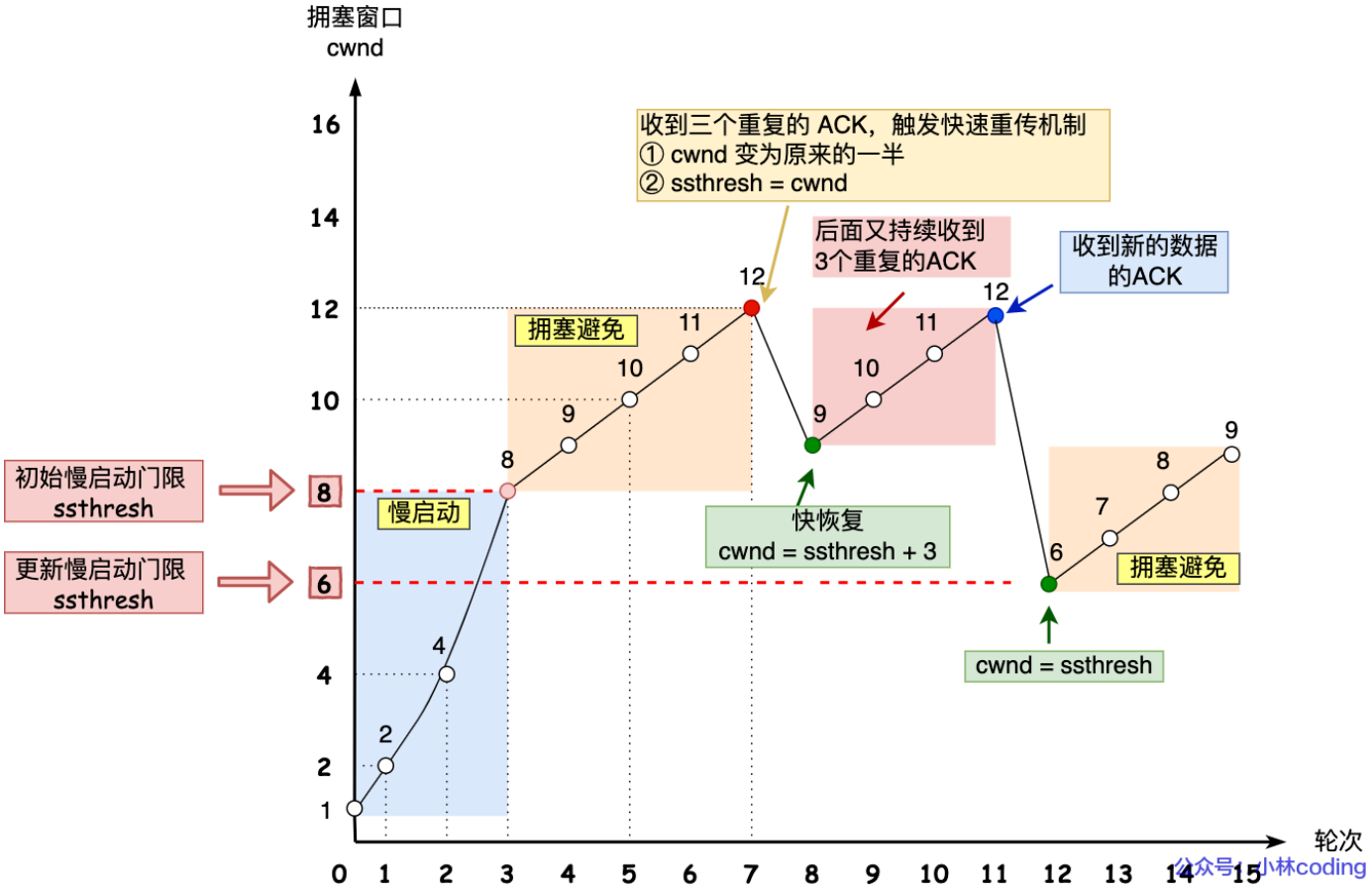
相关文：

https://xiaolincoding.com/network/3_tcp/tcp_feature.html#%E5%BF%AB%E9%80%9F%E6%81%A2%E5%A4%8D

https://bbs.huaweicloud.com/blogs/314817

四个阶段（直接看上面口语）：

- 慢启动：刚开始的时候每次收到ACK就增加一倍的数据，倍数增长，直到 慢启动阈值进入拥塞避免阶段
- 拥塞避免：到达阈值后，每次收到ACK改成线性增长，增加 **1/cwnd**。比如图里就是1/8，要8次ACK才加1。
- 拥塞发生：发生拥塞（有重传）后，慢启动阈值降为当前一半??
- 快速恢复：？？



HTTPS-TODO

HTTP和HTTPS区别

> 多了个ssl层避免明文传输，端口不一样，需要CA机构申请证书

Https相比于Http的缺点

> 握手阶段 耗时大
>
> 消耗CPU去加解密
>
> 证书要花钱

ARP

ARP 协议是已知 IP 地址求 MAC 地址，主机通过广播消息，在同个链路的设备收到后对比包里的IP如果跟自己一样就回复自己的MA地址。

RARP 协议正好相反，它是**已知 MAC 地址求 IP 地址**

分布式

分布式：把一个大任务拆成不同类型的小任务，给不同的人做，他们之间要相互协作。

把一个大业务拆分成多个子业务，子业务之间相互协作最终完成整体的大业务，

集群：多个人做相同的事，只是加了人手，但没有对问题做拆解。

CAP：**一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项**（AP或者CP）

C 一致性是指“所有节点同时看到相同的数据”，对外要么不返回，要么返回相同的数据，比如 zookeeper

A 可用性是指“任何时候，读写都是成功的”，即使返回的数据是老版本的，比如*Eureka(服务发现框架)*

P 分区容忍性具体是指“当部分节点故障的时候，分布式系统整体仍然能够继续运行”

涉及资金相关系统，要求强一致，选择CP

对于个人信息/服务发现这种不要求强一致的就选AP，分布式数据库其实也是这个，保持最终一致性就行

分布式锁

Redis单机： 直接用带NX标识位的 SET命令（ Set key value expireTime nx ） ，或者 setnx+expire放lua里

Redis多实例： 用Redis官方给的 **RedLock** ，原理是同时请求多个实例尝试加锁，有半数以上成功且不超时则认为成功。

RedLock的场景：多个实例之间不是主从关系，如果是主从关系的可能出现A给master加锁了但还没同步到从节点就挂了，B变成master，于是另外一个客户端调B成功拿到了锁

Redisson：一个Java的redis客户端，也实现了redlock，还有watchdog机制（如果没有设置超时时间才生效，在主动释放锁之前每10秒延期30秒避免代码还没跑完锁过期了）

ZooKeeper的临时顺序节点

> 每个客户端对某个方法加锁时，在ZooKeeper上与该方法对应的指定节点的目录下，生成一个唯一的临时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个临时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。

ZooKeeper

ZK的作用

- 分布式锁
- 唯一ID
- 服务注册与订阅
- 配置管理
- Leader 选举

ZooKeeper是怎么保持数据一致性的?

> 二阶段提交+半数投票（合起来叫 ZAB 协议，是zk自己实现的） 关键点：「**只有leader负责写入数据**」
>
> 具体做法：follower收到写请求后会转给leader，leader发起提案，让所有从节点写入磁盘然后返回ack，有半数ack就认为成功，leader再发起commit信息，让从节点提交（写内存，然后用户就能查到了）
>
> 如果leader崩溃了，重启后会检查磁盘上有没未commit的信息，如果有并且有半数ACK，就自己提交一下

ETCD也是分布式服务注册中心

ETCD使用[Raft]协议， ZK使用ZAB（类PAXOS协议）

唯一ID生成

UUID、雪花这、Redis自增或者 MySQL 批量取号，阿里用的MySQL批量取号

TDDL的区间取号方法（依赖MySQL）
　建一个表记录场景(name)、该场景下的最大ID(value)

　在机器启动/重启/号码快用完的时候查一下这个表，比如查到0，然后定一个布长比如1000，就代表这台机器这次使用0到999号，在用完1000个数后才查数据库。同时用乐观锁更新数据库的值为1000(update value=1000 where 场景=x and value = 0，避免多机器相争取到一个号) ，下次其他机器取号时查数据库时就从1000开始。

阿里TDDL和美团Leaf的区别：

　TDDL的步长在代码里，最大值计算在代码里，通过乐观锁更新数据库；Leaf步长在数据库字段里，把查最大ID和更新ID放在一个事务里

```
Begin
UPDATE table SET max_id=max_id+step WHERE biz_tag=xxx
SELECT tag, max_id, step FROM table WHERE biz_tag=xxx
Commit
```

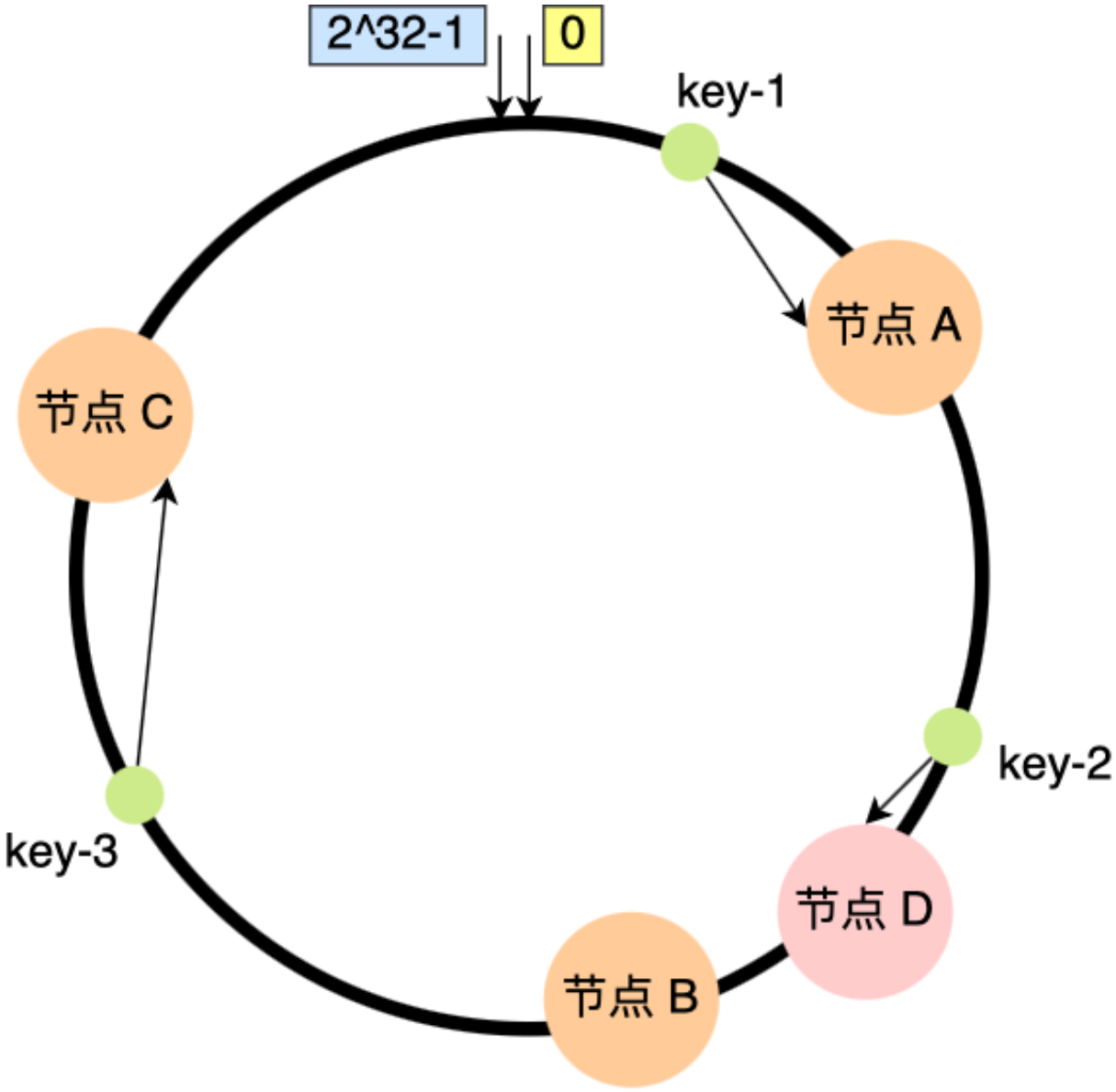
一致性哈希

一致性哈希：分布式下的负载均衡算法，可以用在分布式缓存(Redis) 上解决缩扩容带来的大批量数据迁移问题，也可以用在Nginx上。

方案：将「存储节点」和「数据」都映射到一个哈希环上， 在数据找存储节点时，hash一次后顺指针找最近的存储节点。

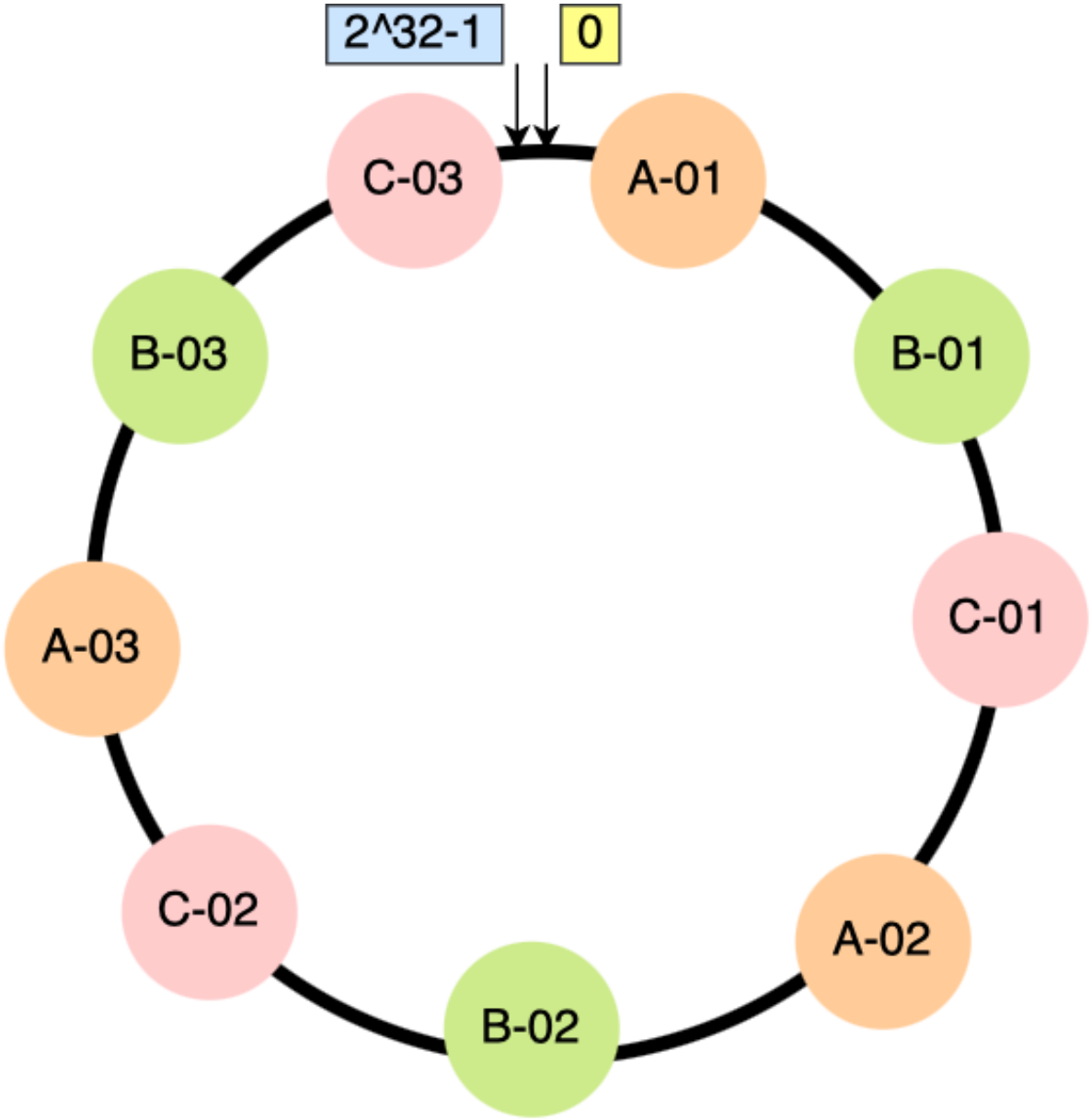
带来的好处是：**新增删除存储节点时，只影响这个节点在环上顺指针的下一个节点**

实现的一个最简单的方法，利用hashcode对2^32进行取模



缺点：节点数不够多的话，无法保证数据均匀

解决方法：对真实节点进行映射让一个真实节点虚拟出N个节点，Nginx是1个真实节点对应160个虚拟节点。



缺点：节点发生故障，把数据移动到下一个节点，可能造成雪崩

好文：https://developer.huawei.com/consumer/cn/forum/topic/0203810951415790238

Redis做法

没使用一致性哈希，使用哈希槽，把数据分到16384（2的14次方）个槽里，每个节点负责一片槽，新增节点时移动一俩个节点对应的槽数据就行。哈希方法：CRC16(key) % 16384

为什么是16384?

> 心跳包中包含节点信息，太大的话心跳包太大，并且Redis节点不太可能超过1千个(2的10次方)，够用

调用跟踪-TraceId

EagleEye 鹰眼做法： 在启动类注册，后面在执行业务逻辑前会先生成唯一的traceId存储在ThreadLocal，在RPC调用下游时传下去。同时ThreadLocal里还存了 traceId下唯一的RpcId ，在调用后序号+1带下去，用于判断调用关系

操作系统

僵尸进程:子进程已经退出，父进程没有调wait回收，会浪费进程描述符。解决方法是 重启或者杀掉父进程，让他变成孤儿进程就有init进程跟进了

如何避免僵尸进程：

父进程调用waitpid等待子进程结束，不过如果子进程在父进程wait之前就退出了，依然会变僵尸，所以可以用「俩次fork」解决：父进程fork出子进程，然后子进程再fork出孙子进程，然后子进程立马退出，这样孙子进程就变成孤儿进程了，会被init接管，就可以拿孙子进程去搞事了。

协程：用户态版的线程，有自己的寄存器上下文和栈，没有内核切换的开销，减少了线程切换的成本

进程间通讯：

> 管道 信号 信号量（**整型的计数器**） socket 消息队列（内核的） 共享内存
>
> 共享内存：把页表里的内存用 shmget（创建共享内存） 和 shmat（share memory attach 映射共享内存）俩个函数，分别映射到各自进程里。

线程间通讯：

> 锁、信号(wait/notify)、信号量(Semaphore)

进程有哪些状态?

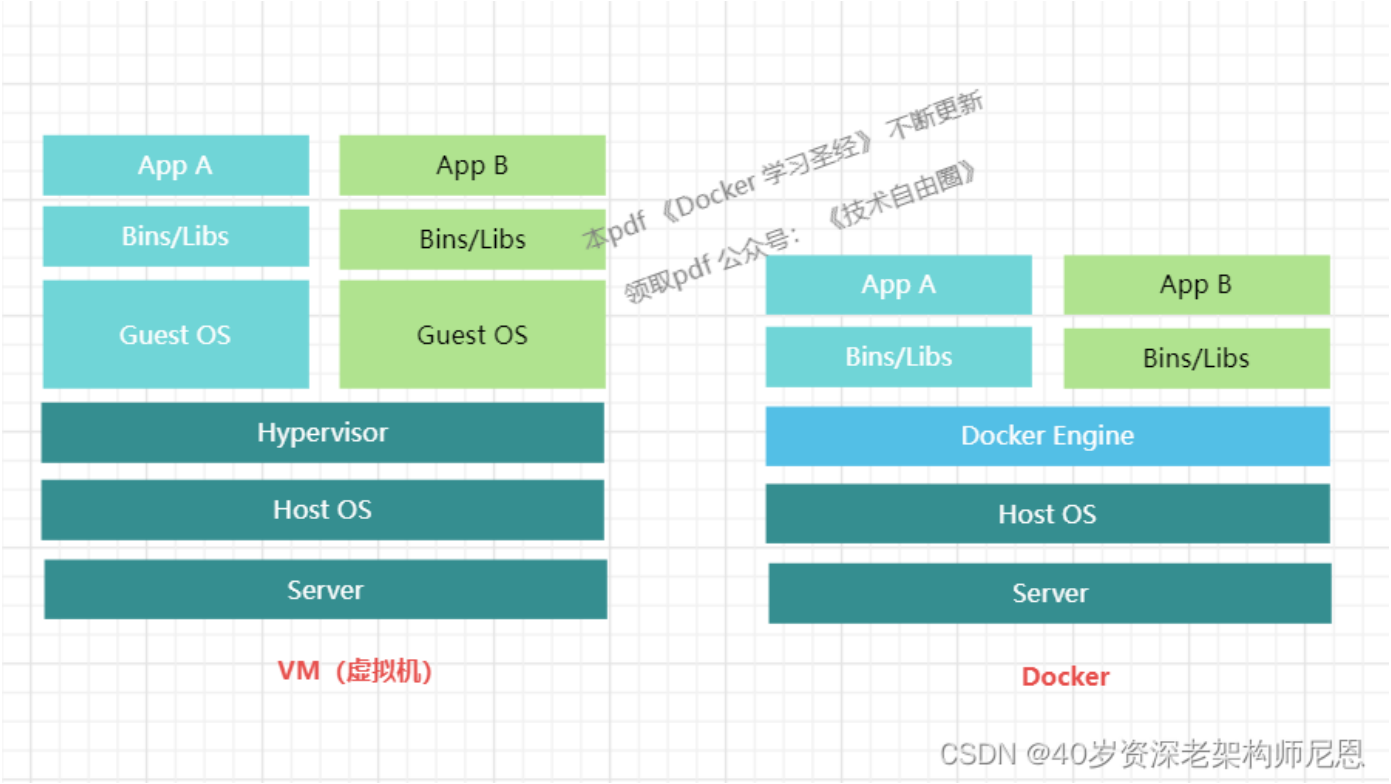
> 新建、就绪、执行、阻塞、终止

线程有哪些状态?

> NEW(新建)、RUNNABLE(运行)、BLOCKED(阻塞)、TIMED_WAITING(超时等待，可以等指定时间就唤醒)、WAITING(等待，要有join)、TERMINATED(终止、结束)

docker对比虚拟机的优势：

> 虚拟机在宿主机上新建了一个虚拟的操作系统，Docker用宿主机的操作系统，所以docker更快更小，更轻量。



Docker的底层机制

> 用的Linux的命名空间，详细不了解

数组和链表区别

> 物理顺序连续 和逻辑上连续 ， 分别代表了**随机访问**和**顺序访问**，数组查询快，链表修改快

异步阻塞和异步非阻塞

https://blog.csdn.net/daaikuaichuan/article/details/83957308

设计题

好文列表： https://zq99299.github.io/note-architect/hc/04/01.html

秒杀系统

无外乎 缓存 限流 降级 异步

- 在库存扣减上，简单做可以乐观锁，复杂做可以Redis扣减+异步改DB
- 如果量大，机器部署还可以做独立集群拆分，页面可以静态化

具体做不做就看QPS大小和公司所处阶段，没必要一来就完美方案，不过有一个点要注意的就是 要小心重试导致流量无限放大，我们就有过因为前端重试导致流量超预期的情况。

好文：https://juejin.cn/post/6990307911117307934#heading-9

短链接

先生成一个短ID（可以db/redis自增，也可以走唯一ID），然后把长短链接关系丢数据库里，然后把ID转成62进制，返回这个62进制的串就行（长度为7就足够用）

一个长地址最好可以对应多个短地址，方便带上用户ID之类的信息做来源识别

为了避免被扫：可以打乱字符顺序（奇偶数互换，字符串反转），加入无用的随机字符

数据库存储：不好存字符串的话，可以存10进制

防止黑客攻击消耗长度：搞个redis存储一天内长网址到ID的关系，这样可以防范相同长地址来调用，并且业务设置上一个长地址最多一天生成XX次短链接

好文：https://segmentfault.com/a/1190000023949445

超卖

数据库超卖：上乐观锁

Redis超卖：上事务，事务期间不会被打断

Feed/微博/朋友圈

好文：

架构师之路： https://cloud.tencent.com/developer/article/1168946

https://juejin.cn/post/7200672322113585207

https://toutiao.io/posts/yy8cawz/preview

好文2：https://xie.infoq.cn/article/19e95a78e2f5389588debfb1c

微信群聊用的推模型，因为人少

关注列表

> 存关注表，关注列表和被关注列表，分表分库，让 查询粉丝列表和关注列表都不跨分片
>
> 关注数、粉丝数这种高频查询的 数量字段，可以在用户表里也加字段，也写一份缓存

发推和收消息

> 发帖
>
> 这种feed流基本是推、拉模型。

> 推模式是写扩散，每个用户除了存自己发的feed，还要存储自己收到的feed流，在发贴的时候给关注人的feed表也写一份数据。（也可以只存储其他人的帖子ID，再读DB或者Redis，只是冗余消息内容会更容易些）
>
> 拉模式是读扩散，每个用户只存储自己发的feed，查看的时候遍历他的关注人列表，去拉他们的feed，逻辑比较复杂。

> ```
> 微博:可以俩个一起用，给活跃在线用户推，给其他用户拉，产品设计上要限制关注数上限
> 微信朋友圈：推模型合适，读多写少，在拉取朋友圈时的操作简单，控制每个用户的好友数量。
> ```
>
> 读贴的时候
> 　针对大V的帖子读取，除了Redis还可以上机器的本地缓存避免Redis的热key，另外可以在发帖后就推给机器做本地缓存，隔几秒再推给用户
> 　针对有热点上升趋势的热点贴也要上缓存
>
> 在机器部署上，要跟大V做部署上的拆分，机器、数据库、缓存 都可以单独部署

用户的feed流可以放缓存里，SortedSet?

feed分页问题：翻页有用户新发布了内容，避免重复，可以记录上一页最后条消息的ID，下一页＞它

限流

计数器算法、漏桶算法、令牌桶算法、滑动窗口

滑动窗口的实现：用Redis的SortedSet，key是限流标识ID，score是时间戳，请求的时候sadd一下，查询的时候，zremrangebyscore看俩个时间戳里有多少个请求就行。

令牌桶/漏斗：

方法1：Redis+Lua，记录上一次的访问时间，跟当前时间求差值算出有多少token

方法2：定时往Redis的list里加 UUID，取就行。

微信红包

https://www.zybuluo.com/yulin718/note/93148

https://developer.aliyun.com/article/936484

单点登录

用户在登录页面输入账号密码登录后，登录系统写入顶级域名cookie，然后跳转到业务系统，地址栏带上生成的token，业务系统再回调一下SSO，验证token有效就把token写入到cookie里

> 需要回调一下，是避免直接访问地址栏带上token的情况，还能验证token是否过期

第二个系统B接入的话，因为登录系统是登录过的，顶级cookie还在，所以用户不用再登录，直接跳转到系统B带上SSO的token，同样写到cookie里，就完成了多系统登录。

其他

秒杀场景下，Redis主库挂了，Redis会选个从库上来，但如果从库同步延迟怎么办

> 问题是主从同步带来的
>
> 方案1 主从同步从异步改成同步，缺点是redis吞吐量下降
> 方案2 直接干掉从节点，牺牲用户体验

7天榜单的实现：

> 多个单天榜单（sortedSet）求并集，zset有现成命令
>
> sortedSet在分数相同的情况下按key的字典序排，所以如果分数相同想按时间排，就在key前面加个时间戳

防止CROS请求

- 跳转相关都要做地址校验，是不是我们白名单域名内的
- 增加来源header的识别

用3个线程分别打印ABC，通过 synchronized 和 notify实现通知，第一个进去打印

https://zhuanlan.zhihu.com/p/370130458

```
class Wait_Notify_ACB {
•
    private int num;
    private static final Object LOCK = new Object();
•
    private void printABC(int targetNum) {
            synchronized (LOCK) {
                while (num % 3 != targetNum) {    //想想这里为什么不能用if代替while，想不起来可以看公众号上一篇文章
                    try {
                        LOCK.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                num++;
                System.out.print(Thread.currentThread().getName());
```

```
                LOCK.notifyAll();
            }
        }

        public static void main(String[] args) {
            Wait_Notify_ACB  wait_notify_acb = new Wait_Notify_ACB ();
            new Thread(() -> {
                wait_notify_acb.printABC(0);
            }, "A").start();
            new Thread(() -> {
                wait_notify_acb.printABC(1);
            }, "B").start();
            new Thread(() -> {
                wait_notify_acb.printABC(2);
            }, "C").start();
        }
    }
```

最近一个小时内访问频率Top10的ip

qps为10w，查询最近一个小时内访问频率Top10的ip，给出你的实现方案

3600个Hashmap< {IP},{cnt} > 或者Map 记录3600秒里每个IP的次数，然后维护个小根堆，存次数最多的TOP10，流量进来的时候更新当秒hahmap的value，遍历3600个map做当前IP的求和，跟堆里最小元素比较决定要不要进堆，查询的时候就直接返回这个堆里元素，然后开个线程每秒删除一个。

方案来自 https://www.bookstack.cn/read/system-design/cn-top-k-frequent-ip-in-one-hour.md

日志里统计TOPK

- 拆小文件然后 hashmap+小跟堆
- 直接 Linux命令

定时任务调度器

- java有个DelayQueue 延迟队列
- 时间轮，弄个循环队列模拟时钟，每秒一个格子，每一个格子后面挂一堆任务，大于60秒的就多记个圈数，转到当前格子并且圈数为0就丢异步任务去执行

算法

二叉查找树特点：

> 左子树的所有节点的值都<=根节点，右边大于等于

> 任意结点的左右子树仍为二叉查找树

AVL树：在二叉查找树基础上，控制左右子树的高度差小于等于1

动态规划：

- **最优子结构**：局部最优解能决定全局最优解
- 无后效性：子问题的解一旦确定，就不再改变，不受在这之后、包含它的更大的问题的求解决策影响。
- **重叠子问题**

滑动窗口：解决字符串问题，不管是否连续、是否重复，结果可以是数字或者字符串

每移动一次左指针就不断推右指针，直到不满足条件或者都满足条件（看具体问题），然后max(结果,窗口结果)

常见题

俩个栈实现队列：

入队时，入栈1

出队时，如果栈2为空，就栈1内容全部弹到栈2，然后出栈2顶部；如果非空，就直接弹出栈2。

然后就是处理 没有元素可以出栈的情况。

判断无向图有环：拓扑排序，把入度为1的进队列，遍历相邻点，度数-1，如果为0进队列

判断有向图有环：拓扑排序，把入度为0的进队列，遍历相邻点，度数-1，如果为0进队列

链表

234. 回文链表

1. 复制到List里，然后前后比较
2. 【面试官想要的O(1)空间】 先快慢指针找到中点，然后反转中点后面的，再双指针从头比较

```
class Solution {
    public boolean isPalindrome(ListNode head) {
        if (head == null) {
            return true;
        }

        // 找到前半部分链表的尾节点并反转后半部分链表
        ListNode firstHalfEnd = endOfFirstHalf(head);
        ListNode secondHalfStart = reverseList(firstHalfEnd.next);

        // 判断是否回文
        ListNode p1 = head;
        ListNode p2 = secondHalfStart;
        boolean result = true;
        while (result && p2 != null) {
            if (p1.val != p2.val) {
```

```
                result = false;
            }
            p1 = p1.next;
            p2 = p2.next;
        }

        // 还原链表并返回结果
        firstHalfEnd.next = reverseList(secondHalfStart);
        return result;
    }

    private ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode nextTemp = curr.next;
            curr.next = prev;
            prev = curr;
            curr = nextTemp;
        }
        return prev;
    }

    private ListNode endOfFirstHalf(ListNode head) {
        ListNode fast = head;
        ListNode slow = head;
        while (fast.next != null && fast.next.next != null) {
            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;
    }
}
```


如何判断一个链表是否有环，及求环的入口

> 快慢指针，一个每次走俩步，一个走一步，能遇到就是要环。

> 求环的入口：根据数学公式计算，相遇时，再开个指针从链表头开始，同慢指针一起每次走一步，相遇时就是入口

```
public static ListNode findCyclePosition(ListNode head){
    if (null == head || null == head.next){
        return null;
    }
    ListNode slow = head;
    ListNode fast = head;
    while (null != fast && null != fast.next){
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast){//快慢指针相遇时
            ListNode ptr = head;//定义一个新指针
            while (ptr != slow){
                ptr = ptr.next;
                slow = slow.next;
            }
            return ptr;//返回环的入口位置
        }
    }
    return null;
}
```

树

层序遍历（用 队列大小）、最近公共祖先 、最长路径和

二叉树的最近公共祖先（LCA）

如下，分别求俩边的lca，如果俩节点分布在左右子树就是父节点，否则就哪边非空就是哪边的LCA。查询多次的情况，可以 Tarjan

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (root == q || root == p || root == NULL) return root;
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);
        if (left != NULL && right != NULL) return root;
        if (left == NULL) return right;
        return left;
    }
};
```

DP

背包、打家劫舍、股票买

3. 无重复字符的最长子串

```
输入：s = "abcabcbb"
输出：3
解释：因为无重复字符的最长子串是 "abc"，所以其长度为 3。
```

解法：滑动窗口+HashSet，左右指针

移动左指针

set删除移动前的字符

while 不越界且Set不包含新字符就不断移动右指针

set追加

cnt++

ans = max(ans,cnt);