- 计网
  - 端口号：
  - 三次握手
  - 四次挥手
  - 流量控制（滑动窗口）
  - 拥塞控制
  - 主机访问网站时解析域名的过程
  - get和post区别
  - HTTP状态码
  - TCP和UDP的区别
  - TCP的可靠性保证
  - Http1.0和Http1.1
  - http2.0与HTTP 1.1相比，主要区别包括
  - http和https
  - https加密过程
  - tcp首部
  - 粘包问题
  - 广播风暴
  - RST包
- 数据库（sql）
  - 索引
  - 数据库优化
  - 事务
  - 索引分类
  - 索引失效
  - 储存引擎的区别（MyISAM和InnoDB）
  - #与$
  - 数据库索引实现
  - 主从复制
  - 分库分表
- 数据结构
  - morris遍历二叉树，O(1)空间复杂度，O(n)时间
  - A*算法
- 操作系统
  - 进程与线程的区别
  - 作业调度算法
  - 进程通信方式
  - 死锁条件
  - 分页分段与缺页调度
  - 磁盘调度
- Redis
  - 基础数据结构
  - 对象
  - 缓存一致性问题
  - 缓存穿透与缓存雪崩
  - 数据库
  - 持久化
  - 事件
  - 客户端和服务器
  - 复制
  - redis死锁
  - redis淘汰缓存策略

哨兵
集群

计网

端口号：

http=TCP+80；https=TCP+443；ftp=TCP+21；POP3=TCP+110（电子邮件）；SMTP==TCP+25（电子邮件）；RDP=TCP+3389；共享文件夹=TCP+445；SQL=TCP+1433，实时状态更新，文件传输等

DNS=UDP+53 or TCP+53（很少），RIP=520，tftp=69，snmp=161（网络管理）。视频传输，实时通讯等

三次握手

- 1.客户向服务器发送同步数据包：SYN=1，ACK=0，seq=x（同步有效，确认号无效，序号是x），客户端从CLOSED状态变为SYN-SENT状态。
- 2.服务器回复同步数据包：SYN=1，ACK=1，seq=y，ack=x+1（同步有效，确认号有效，序号为y，确认号为x+1），服务器从CLOSED变为SYN-RCVD状态。
- 3.客户端回复：ACK=1，seq=x+1，ack=y+1（确认号有效，序号为x+1，确认号为y+1），发送后客户端和服务端都变为ESTABLISED状态
- 存在第三次握手的原因：防止已失效的连接请求报文段突然又到了B，从而产生错误。

  比如：A给B之前发送的请求报文滞留在了路上，没发过去，于是A重新给B发送了请求报文并成功，此时之前滞留的请求报文又到了，于是发送了两次。此时B以为A又要建立新的连接，于是又发回了确认，但A又不理B，只认前面那次连接，但B为A多开了一个连接，并等待A发来数据，导致B的资源浪费。
- 握手目的：确认对方存在，建立连接，协商参数（最大窗口值，窗口扩大选项，时间戳选项，服务质量），对运输实体资源进行分配（缓存大小，连接表项目）

四次挥手

- A向B发送TCP释放连接报文段，并停止传输数据，主动关闭TCP连接。（首部FIN=1，序号seq=u）等待B的确认。A从ESTABLISED状态变为FIN-WAIT-1状态
- B回复确认，（确认号ack=u+1，序号seq=v，ACK=1），TCP服务器进程通知高层，此时从A到B的通道就关闭了，但B到A的通道还开着（TCP连接半关闭状态），B若给A发数据，A仍然要接收。B从ESTABLISED状态变为CLOSE-WAIT状态，A变为FIN-WAIT-2状态（收到fin无回复），若收到fin后回复ack进入closing状态
- 此时B向A发送TCP释放连接报文段（首部FIN=1，序号seq=w，ACK=1，ack=u+1），确认号依然按照A发送的释放连接的序号来决定.B变为LAST-ACK状态
- A回复确认（确认号ack=w+1，序号seq=u+1，ACK=1），此时B到A的通道也关闭，B收到回复后直接进入CLOSED状态，而A发送后进入TIME-WAIT状态，并在等待2msl（msl称为最长报文段寿命，是2分钟，所以需要等待四分钟）的时间后进入CLOSED状态（只有进入CLOSED状态才能开始下一次连接）
- 设置TIME-WAIT状态的原因：
  - 最后回复给B的确认消息可能丢失，若在规定时间内，B没有收到确认信息，B会重新给A发送TCP释放连接报文段（首部FIN=1，序号seq=w，ACK=1，ack=u+1），此时的A需要重新回复，并重新开始2msl的倒计时。
  - 经过时间2msl后，可以确保本连接持续时间内所产生的报文段全部消失，使得下一次开始时不会收到之前说过的“已失效的连接请求报文段”

流量控制（滑动窗口）

- 以字节为单位的滑动窗口
- 接收方与发送方分别维护一个缓冲区（要求发送方缓冲小于等于接收方缓冲区）

- 使用一个窗口确定发送方可以发送范围，只要窗口范围内的数据还未发送过，发送方都能发出
- 接收方收到数据后会给发送方回复确认消息
  - 若正常收到连续块，回复消息中确认号为连续块最后一个字节编号加1，让发送方接着发
  - 若接收断层了，比如收到了12，345，89，中间的67断掉丢失了，接收方会给发送方回复6表示，并附带消息标记中间丢失的部分，让发送方重发67.
- 发送方每次收到接收方的确认消息，都会根据确认号，挪动窗口的相应位置，使得窗口内出现新的未发送字节。
- 接收方还会返回消息来现在滑动窗口的大小来限制发送方的发送速度
  - 糊涂窗口综合征

    由于接收方或者发送方性能远小于另一边，导致发送方每次发送的包都很小，当发送方性能差时，还没准备多少数据，就被接收方要求再发送数据，导致发送的数据量很少；当接收方性能差时，就让发送方滑动窗口很小，甚至直接为0，接收方刚腾出一点空间，让发送方开个小窗口，由于窗口小就只能发送很少的数据。这样每次发送数据少，且由于数据头部的固定开销存在，导致发送速度更慢，所以针对某一遍性能差有两种解决思路。
    - 发送方性能差

      使用nagle算法，强制发送方准备数据，不要发送短数据，一般时延是一个RTT 时会强制发送或者提前准备足够一个最大报文段也可发送。
    - 接收方性能差

      第一种使用Clark解决方法，接收端缓存满时让发送端窗口为0，当接收端缓存清理空间足够一个报文段长度或者缓冲区清理空间达到一半时才重设发送端的窗口大小。第二种是延迟确认ACK，这样发送端窗口内数据已经发送完毕，无法发送新的数据，接收端直到有足够空间才确认上一个ACK，这样对比第一种方案，还减少了通信量（调整窗口大小的通信）

拥塞控制

- **慢开始算法**：第一次发一个包，若不丢包，第二次发2个，若仍未出现丢包，再增加两倍（每个RTT时间翻倍，其实是收到一个包的ack+1，这样一个RTT时间后就翻倍了），直到达到每次发的个数大于等于门限值（初值为16）时，则让发包数等于门限值。若仍未出现丢包，后面就每次增加一个包。直到发送包数量为n时出现丢包现象，此时设置门限值为n/2，并从每次1个包再次开始发，每次增加两倍直到要大于等于门限值时，就将发包数定位和门限值一样，并又开始采取逐个增加发包数的手段，直到再一次丢包。
- **快重传和快恢复**：
  - 快重传：发送方连续发送包，接收方收到失序包，比如收到的是1.2.4，失序了3号，就回复确认号3，后面收到的每个包只要不是3都回复3，接收方连续收到3个同样的ack，不会等待过时才重传，而是直接重传确认号对应的包。
  - 快恢复：快恢复的快是相对于慢开始的，收到三个连续ack后，将发包次数从n立即变为n/2，用n/2作为新的门限，并开始逐个增加发包次数，直到再次丢包，不需要变为1再指数增长。

主机访问网站时解析域名的过程

首先和本机的hosts文件，没有就访问本地域名服务器，并提交访问的域名给DNS服务器，服务器首先检查这个域名服务器中是否有，有就直接返回

没有的话分两种查找方式

- 递归查找：本地dns服务器向根服务器提交域名，根服务器按照域名分级向下递归查询，直到找到对应负责的服务器并返回查询结果
- 迭代查询：本地dns服务器向根服务器提交域名，根服务器告诉客户负责这个域名的下一级服务器是谁，客户再次访问，依次直到找到对应的服务器。

get和post区别

get是对于数据库是安全幂等的，但post意味着可能对数据库产生修改；get的请求数据附在url后面，所以可能被剽窃，post的数据在http的包的包体（正文段）里面，不能被窃取剽窃，并且get由于保存在url里面所以有长度限制，post理论上没有长度限制；get是一次性的，post分两次；get得到的数据，浏览器会主动缓存，post的数据，不设置不会主动缓存。

HTTP状态码

*100	通知客户端接收成功，让客户端发剩下的数据
101	服务器已经理解了客户端的请求，并通知客户端，服务器将采用更好的协议来完成这个请求。
102	扩展的状态码，代表处理将被继续执行。
*200	请求已成功，请求所希望的响应头或数据体将随此响应返回。
201	请求已经被实现，有一个新的资源已经依据请求的需要而建立，且其 URI 已经随Location 头信息返回。
202	服务器已接受请求，但尚未处理。最终请求可能被执行也可能不被执行
203	服务器已成功处理了请求，但返回的实体头部元信息不是在原始服务器上的，而是来自本地或者第三方的拷贝。
204	服务器成功处理了请求，但不需要返回任何实体内容。
205	服务器成功处理了请求，且没有返回任何内容。但是与204响应不同，返回此状态码的响应要求请求者重置文档视图。比如重置表单
206	服务器已经成功处理了部分 GET 请求。实现断点续传的
207	扩展的状态码，代表之后的消息体将是一个XML消息。
300	被请求的资源有一系列可供选择的回馈信息，每个都有自己特定的地址和浏览器驱动的商议信息。用户或浏览器能够自行选择一个首选的地址进行重定向。
*301	被请求的资源已永久移动到新位置，并且将来任何对此资源的引用都应该使用本响应返回的若干个 URI 之一。
*302	请求的资源现在临时从不同的 URI 响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。
*303	对应当前请求的响应可以在另一个 URI 上被找到，而且客户端应当采用 GET 的方式访问那个资源。在1.1中用于get的重定向，直接发get重定向
304	如果客户端发送了一个带条件的 GET 请求且该请求已被允许，而文档的内容（自上次访问以来或者根据请求的条件）并没有改变，则服务器应当返回这个状态码。
305	被请求的资源必须通过指定的代理才能被访问。Location 域中将给出指定的代理所在的 URI 信息，接收者需要重复发送一个单独的请求，通过这个代理才能访问相应资源。
306	在最新版的规范中，306状态码已经不再被使用。
*307	请求的资源现在临时从不同的URI 响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。1.1中用于post的重定向，但很少使用，重定向时要询问客户端

*400	提交请求存在语法或其他错误，服务端无法理解
*401	未授权，客户端权限不够，当前请求需要用户验证。
402	该状态码是为了将来可能的需求而预留的。
*403	服务器拒绝执行请求。
*404	服务器上找不到请求的资源
*405	请求行中指定的请求方法不支持（没配置restful就不支持DELETE等）。
406	请求的资源的内容特性无法满足请求头中的条件，因而无法生成响应实体。
407	与401响应类似，只不过客户端必须在代理服务器上进行身份验证。
408	请求超时。
409	由于和被请求的资源的当前状态之间存在冲突，请求无法完成。
410	被请求的资源在服务器上已经不再可用，而且没有任何已知的转发地址。
411	服务器拒绝在没有定义 Content-Length 头的情况下接受请求。在添加了表明请求消息体长度的有效 Content-Length 头之后，客户端可以再次提交该请求。
412	服务器在验证在请求的头字段中给出先决条件时，没能满足其中的一个或多个。
413	服务器拒绝处理当前请求，因为该请求提交的实体数据大小超过了服务器愿意或者能够处理的范围。
414	请求的URI 长度超过了服务器能够解释的长度，因此服务器拒绝对该请求提供服务。
415	对于当前请求的方法和所请求的资源，请求中提交的实体并不是服务器中所支持的格式，因此请求被拒绝。
416	如果请求中包含了 Range 请求头，并且 Range 中指定的任何数据范围都与当前资源的可用范围不重合，同时请求中又没有定义 If-Range 请求头，那么服务器就应当返回416状态码。
417	在请求头 Expect 中指定的预期内容无法被服务器满足
421	从当前客户端所在的IP地址到服务器的连接数超过了服务器许可的最大范围。
422	从当前客户端所在的IP地址到服务器的连接数超过了服务器许可的最大范围。
422	请求格式正确，但是由于含有语义错误
424	由于之前的某个请求发生的错误，导致当前请求失败
425	在WebDav Advanced Collections 草案中定义，但是未出现在《WebDAV 顺序集协议》（RFC 3658）中。
426	客户端应当切换到TLS/1.0。（RFC 2817）
449	由微软扩展，代表请求应当在执行完适当的操作后进行重试。

*500	服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理。一般来说，这个问题都会在服务器的程序码出错时出现。
501	服务器不支持当前请求所需要的某个功能。当服务器无法识别请求的方法，并且无法支持其对任何资源的请求。
502	作为网关或者代理工作的服务器尝试执行请求时，从上游服务器接收到无效的响应。
*503	由于临时的服务器维护或者过载，服务器当前无法处理请求。这个状况是临时的
504	作为网关或者代理工作的服务器尝试执行请求时，未能及时从上游服务器（URI标识出的服务器，例如HTTP、FTP、LDAP）或者辅助服务器（例如DNS）收到响应。 注意：某些代理服务器在DNS查询超时时会返回400或者500错误
*505	服务器不支持，或者拒绝支持在请求中使用的 HTTP 版本。
506	代表服务器存在内部配置错误
507	服务器无法存储完成请求所必须的内容。这个状况被认为是临时的。
509	服务器达到带宽限制。
510	获取资源所需要的策略并没有没满足。

TCP和UDP的区别

1. TCP是**面向连接**的；UDP是**无连接**的（但可以使用connect，指定只与这台主机进行数据交互，不是的就抛异常），发送数据之前不需要建立连接
2. TCP是提供可靠**全双工**的通信服务。UDP是**半双工**，只能单向传播
3. 通过TCP连接可靠传送的数据，**可靠的**、**无差错**，**不丢失**，**不重复**，且**按序到达**;UDP则是**不可靠信道**，尽最大努力交付，即不保证可靠交付
4. TCP**面向字节流**，实际上是TCP把数据看成一连串**无结构的字节流**;UDP是**面向报文**的
5. TCP具有**拥塞控制**，UDP**没有拥塞控制**，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）
6. 每一条TCP连接只能是**点到点**的;UDP比较灵活，支持一对一，一对多，多对一和多对多的交互通信
7. TCP首部**开销20字节**;UDP的**首部开销小**，只有8个字节

TCP的可靠性保证

确认回复，超时重传，数据校验，合理分片与排序，流量与拥塞控制

Http1.0和Http1.1

1. **HTTP1.0 是短连接，HTTP1.1是长连接。** HTTP 1.0规定浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个TCP连接，服务器完成请求处理后立即断开TCP连接，服务器不跟踪每个客户也不记录过去的请求。如果一个html包含多个图片或资源时需要多次与服务器建立连接。 HTTP 1.1支持持久连接，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟。一个包含有许多图像的网页文件的多个请求和应答可以在一个连接中传输，但每个单独的网页文件的请求和应答仍然需要使用各自的连接.**HTTP 1.1采用了流水线的持久连接，即客户端不用等待上一次请求结果返回，就可以发出下一次请求**，但服务器端必须按照

接收到客户端请求的先后顺序依次回送响应结果，以保证客户端能够区分出每次请求的响应内容，这样也显著地减少了整个下载过程所需要的时间。

2. **HTTP 1.0不支持Host请求头字段，WEB浏览器无法使用主机头名来明确表示要访问服务器上的哪个WEB站点**，这样就无法使用WEB服务器在同一个IP地址和端口号上配置多个虚拟WEB站点。在HTTP 1.1中增加Host请求头字段后，WEB浏览器可以使用主机头名来明确表示要访问服务器上的哪个WEB站点，这才实现了在一台WEB服务器上可以在同一个IP地址和端口号上使用不同的主机名来创建多个虚拟WEB站点。
3. HTTP 1.1还提供了与身份认证、**状态管理**和**Cache缓存**等机制相关的请求头和响应头
4. **带宽优化**。HTTP/1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了。例如，客户端只需要显示一个文档的部分内容，又比如下载大文件时需要支持断点续传功能，而不是在发生断连后不得不重新下载完整的包。 HTTP/1.1中在请求消息中引入了range头域，它允许只请求资源的某个部分。
5. HTTP/1.1增加了OPTIONS方法，它允许客户端获取一个服务器支持的方法列表

http2.0与HTTP 1.1相比，主要区别包括

1. HTTP/2采用二进制格式而非文本格式，网络层和传输层之间建立二进制帧层
2. HTTP/2是完全多路复用的，而非有序并阻塞的——只需一个连接即可实现并行
3. 改进报头压缩算法，HTTP/2降低了开销
4. 优先级设计（如css>html这些）
5. 流量控制从端到端改进到逐跳
6. HTTP/2让服务器可以将响应主动“推送”到客户端缓存中（发点其他的）

http和https

1. http是HTTP协议运行在TCP之上。所有传输的内容都是**明文**，客户端和服务器端都无法验证对方的身份
2. https是HTTP运行在**SSL/TLS**之上，SSL/TLS运行在TCP之上。所有传输的内容都经过**加密**，加密采用对称加密，但**对称加密的密钥用服务器方的证书进行了非对称加密**。此外客户端可以验证服务器端的身份
3. 如果配置了客户端验证，服务器方也可以验证客户端的身份。
4. https协议需要到ca申请证书，一般免费证书很少，需要交费。
5. http是超文本传输协议，信息是明文传输，https 则是具有安全性的ssl加密传输协议
6. http和https使用的是完全不同的连接方式用的**端口也不一样**,前者是80,后者是443。

https加密过程

- 客户端向服务器发起请求（带上随机数1），服务器对客户端进行回应（带上随机数2），并带上自己的SSL证书
- 客户端会对证书进行校验（查看有效期，所有者等，并在操作系统中内置的受信任的证书发布机构CA，与服务器发来的证书中的颁发者CA比对，用于校验证书是否为合法机构颁发 ）
- 校验无问题后取出公钥，再产生一个随机数，使用三个随机数结合生成对称密钥，用对称密钥来对称加密内容，并用公钥非对称加密对称密钥后一起发送给服务器
- 服务器用私钥解密就能得到对称密钥，两者之后就一直使用这个对称密钥对信息进行加密通信三

tcp首部

- 源，目的端口：发送的端口起点和终点，每个2字节
- 序号：表示TCP数据部分是整个文件的第多少个字节，用于接收方拼装成文件，及查验漏包等
- 确认号： 接收方回复时，在确认号中写上次接收到文件的最后一个字节数+1（累加确认方式，收到多个时回复最后一个），方便发送方发送后续文件（既回复了确认消息，也让发送方知道这次该从那开始发）
- 数据偏移：记录TCP报文段的第多少个字节是数据，四个字节，最大15，标识第数据偏移值*4个字节，所以首部最长60字节，可变地方最长40字节
- 保留：6个字节，未使用
- 标记位：共六位，每位一个作用

- URG：若标记为1，则这个数据部分在缓冲中可以插队，提前发送（例：取消发送文件时使用，将TCP数据报URG置1插队放在前面先发给接收方，接收方就立即停止接收，而不用等缓冲区的东西先发完再停止）
- ACK：标识确认号是否有效，为0时表示确认号无效，为1表示确认号有效（例：建立会话时，发起方一般置0，回复方置1 ）
- PSH：若为1，接收方收到后，直接放到接收方缓冲区的首部，让接收方提前从缓冲区拿出来（不然接收方拿数据是按照先到先拿的方式）
- RST：为1表示异常中断（发送方停止发送时使用，直接停止传输）
- SYN：发请求通讯建立连接时，一般置1（来回都是1）
- FIN：数据发送完时，本位置1，表示传输结束，释放连接

- 检验和：用于首部的数据检验
- 紧急指针：紧急数据在尾部结束的位置

粘包问题

多个比较小的包沾合在一起，解决方法：发包定长（多拆少补）；特殊符号分隔；数据头上记录总长度。

广播风暴

- 不合理的网络划分。比如很多客户机处于同一个网段内。由于ARP、DHCP都是广播包的形式，那么有时候就会产生广播风暴。
- 环路。环路时，数据包会不断的重复传输，也一样会产生广播风暴。

RST包

用于关闭异常连接，RST包会让发送缓冲区丢弃所有其他包，然后发出RST，接受方不需要回信，收到后丢弃缓冲区的包。发送RST包的情况：①建立连接时服务器没有监听的端口，②请求超时时，③套接字的异常关闭和中断。

数据库（sql）

索引

为数据表某列排序储存，如果查找时使用这一列为关键字进行查询的，那么由于存储了列关键字与这一行对应值的地址的映射

CREATE/ALTER/DROP INDEX index_name ON table_name (column_list)

- **优缺点**：在列建立的索引上进行查询就大大的加快了查询速度，同样对适宜的列建立索引可以加速表连接（外键列），减少分组和排序的时间（分组，排序关键字），加快判断速度（where子句关键字）。但由于每建立一个索引就多时间创建，花费空间储存，所以对空间不友好。并且增删改数据对应索引也需要变化维护，所以索引的建立对空间不友好，降低增删改的效率。
- **索引建立位置**：索引的使用在大量查询的表中比较适合，索引适合建立在做查询字段，where子句上的列，分组关键列，排序关键列，外键列。

数据库优化

- 若表中很少对数据进行更新，且经常只查询某几个字段，可对这几个字段都创建索引，由表查询变为索引查询，大大加快查询效率。
- 通过explain来查看SQL语句的执行效果，可以帮助更好的选择索引和优化查询语句，写出更好的优化语句。通常我们可以对比比较复杂的尤其是设计到多表的SELECT语句，把关键字explain加到前 面，查看执行计划。例如：explain select * from news;
- 避免使用select * from t,用具体的字段列表替代“*”，不要返回用不到的任何字段。
- 不在索引列做运算或者使用函数
- 查询尽可能使用limit减少返回的行数，减少数据传输时间和带宽浪费

- 固态硬盘比较快的在不用索引，用遍历的时候（范围查找，Join链接查询）用force index强行使用索引
  - 范围*查找时，如果查询数据较少，那会优先选择使用索引，但数据项过多（大于20%），会自动使用聚集索引（遍历全表）（因为这个范围内建立的索引有序，但用辅助索引得到主键后去访问聚集索引时还是散列查找的）。但如果查询的是个别字段，且这些字段有覆盖索引，会走覆盖索引。

事务

ACID：

- A：原子性mysql中通过回滚日志来保证
- C：一致性
  - 这里的一致性指的是操作前后的数据不会违背数据库的数据的完整性约束，转账这类操作的事务逻辑的正确，而CAP中的一致性指的是分布式数据库中多机数据的一致。
- I：隔离性 隔离级别
  - DEFAULT：后端数据库默认隔离级别（mysql：repeatable read；oracle：read commited）
  - read uncommited：最低级别，不加锁，允许读取未提交数据，可能出现脏读，不可重复读，幻读
  - read commited：写的过程行锁锁住数据，避免脏读，可能出现不可重复读，幻读
  - repeatable read：行锁锁住数据整个事务，避免脏读和不可重复读，可能出现幻读
  - serializable：表锁锁住数据整个事务，避免三种情况，效率最低，锁整个表。

  而事务隔离级别的实现就使用了锁的机制、
  - **三种粒度的锁**
    - 表锁：访问期间直接锁整张表，其他进程无法读写整个表，开销小，不会死锁，加锁快，并发度低
    - 行锁：对访问的那条记录加锁，开销大，可能会死锁，加锁慢，并发度高
    - 页锁：对访问记录所在页（多条记录构成的组）加锁，性质位于前面两者中间
  - **悲观锁和乐观锁**
    - 悲观锁：把要修改的数据锁起来，直到任务完成，不允许其他线程接触，加锁时间很长，并发性不好。实现比如行锁，表锁等。读锁，写锁等操作前加锁都是。
    - 乐观锁：访问都不加锁，直到提交修改时才短暂锁住数据，可能出现脏读现象，实现方式：一般是利用版本号机制实现（改前读一次，提交修改前再查一次，版本号一样就提交，提交后版本号+1）
  - **MVCC和间隙锁和意向锁**

  MVCC是一种版本号表锁机制，每条记录都有一个版本号和删除版本号，事务也有自己的版本号，每个事务只能读取版本号小于等于自己，删除版本号大于自己的或删除版本号为空的记录，每条事务新提交数据时，会让记录的版本号等于事务自身的版本号，删除版本号为空；事务删除记录时，会让记录的删除版本号等于自己的版本号；事务修改记录时，会让记录的删除版本号等于自己的版本号并新建一条记录为修改后的结果，MVCC保障了InnoDB的可重复读。

  间隙锁是指事务操作对于表中某个范围进行操作时，锁上范围内所有存在和不存在的键，比如对25-28进行操作，数据库中只有25和27，这两条记录上锁的同时也对不存在的26和28两个键上锁，也就是说此时不允许另一个事务新建键为26的数据项。间隙锁保障了InnoDB不会出现幻读。

  意向锁分共享和排他，是表级的，当事务申请行级锁时，对应也给其分配意向锁，如果此时另一个事务要申请表锁，会根据当前有哪些意向锁来判断能不能申请，这样就不用遍历每一行去判断有哪些行锁了。
- D：持久性mysql，通过重做日志来保证，对错误提交的事务进行重做。

索引分类

普通索引，主键索引，唯一索引，复合索引，聚集索引，非聚集索引

- 联合索引：用多个键进行创建的联合索引，（a,b,c），那么会把a，b，c的所有组合作为键，所有，有最左匹配的原则，在用a或者（a,b）时都能用这个索引，并且第二三个键是排序了的，以a and b为条件，c做排序条件时，也可使用这个索引。
- 全文索引：innodb采用倒排索引的full inverted index，一般倒排索引只包括关键词和对应的文章id，full inverted index中还会存储关键字在文章中具体出现位置
- 覆盖索引：innodb中辅助索引找到的是主键，再用主键去找聚集索引找到正确的行数据，而覆盖索引中辅助索引中就可以得到查询数据（需要查询的字段的值）

索引失效

- 使用不等号时索引没用
- where子句中使用了函数，不是直接的列关键字时，索引没用
- 字符串要用引号，否则不会使用索引
- join多表联查时，如果两表没用外键关联，索引失效
- 使用like或者regexp这样的模糊查询，如果开头字符是匹配符号，索引失效
- order by操作中排序条件是查询条件表达式时索引失效（多表联查分组时索引有用也帮助不大）
- 使用or时，如果多个关键字中有一个没索引，那么索引失效
- 列项若为几个数据的重复（0/1），建立索引也没用。
- **数据库三级范式** 1NF：字段不可再分，原子性 2NF：满足第二范式必须满足第一范式。且非主键属性必须完全依赖于主键属性。 3NF：满足第三范式必须先满足第二范式。每列都与主键有直接关系，不存在传递依赖。
- **死锁**：资源互斥，请求保持，资源不可抢占，环路等待。

储存引擎的区别（MyISAM和InnoDB）

- MyISAM是非事务安全（不支持事务），不支持外键，InnoDB是事务安全的，且支持外键。
- MyISAM锁的粒度是表级，InnoDB是行级。
- MyISAM和InnoDB数据索引都使用B+树实现，MyISAM可以没有主键，在B+数存放的是实际数据对应的指针，而InnoDB必须要主键，其在叶子节点上存放的就是实体数据，所以InnoDB根据主键建立的索引其实是聚集索引，其物理位置对应的就是逻辑位置。InnoDB的非主键索引的叶子节点上存的是这个索引对应项的主键，然后拿到主键又去主键对应的B+树上取。这样做的好处在于当实际数据移动时索引的维护十分友好，因为索引只与主键建立联系（实际数据在叶节点上，增删改数据容易导致移动）
- **储存引擎选择**：MyISAM的查询效率更加可观，而且内存占用很少，适用大量查询，需要高速检索的表设计。InnoDB支持ACID事务支持，适合高并发频繁写的时候。

#与$

运行时便于用作占位符，但#会自动完成java类型到jdbc类型的转化，类似sql中使用？，预编译就处理了；而$直接拼接sql串，在预编译（提高了安全性与效率）之后执行时占位符对应的串只会被作为数据，后续不再解析准备sql。#可以有效的防止sql注入的发生，一般在表名替换时使用dollar

防止sql注入：预编译，正则过滤

数据库索引实现

- 哈希：类似于哈希表，进行散列并解决hash冲突，键为索引关键字，值为对应数据指针或数据本体
- B树：是一个多叉排序树，左小右大，每个节点储存了多个对应索引关键字及对应地址，且按关键字排序左小右大，在同一节点下两个相邻关键字中间的指针指向其孩子节点，孩子节点内部的所有关键字小于父节点指针右边关键字，大于左边。依次排布。寻找关键字时从根节点开始依次往下寻找，搜索性能等价与二分查找，但多叉减少了磁盘IO加载索引的次数，因为所有节点都存放的有数据，所有搜索可能在非叶子节点结束。根节点的**非叶子节点孩子**个数一般为2个（也可一个或没有），下面第m层节点，非叶子节点孩子个数为[m/2,m] (向上取整)，每个节点内关键字数量为**孩子节点数-1**;新建节点时从对应叶子节点中插入，如导致此叶子节点容量超标则将插入节点依次上移，直到不出现容量超标，或移动到了根节点（注意上移过程中，导致出现新的分区，那么叉路也会相应增多）。删除时也是对应一致的，若删除后导致树的不平衡，需要进行左旋或者右旋来调整。

- B+树：结构和b树十分类似，但其非叶子节点上只存放索引关键字，不存放关键字对应的数据或指针，每个节点中分区后上一层的关键字会重复出现在下一层，每个节点若有n个子树，则节点中也有n个关键字，所有的数据或指针都存放在最后一层的叶子节点中，且最后一层的叶子节点按从小到大（从左到右依次连接）。
- B+树优点：首先非叶子节点只存放索引而不存放数据，使得每个节点可以保存更多的关键字，所有树就更加的矮胖，io加载次数更少，不管找那个节点都要找到叶子，使得查找稳定（不慢），且叶子节点用指针连接，在进行范围查找时更方便，不像b树范围查找时需要中序遍历。且B+树数据的删除和增加都只在叶子节点上进行。

主从复制

实现方式有两种，第一个是基于BinaryLog的方式，另一种是基于GlobalTransactionIdentifiers (GTIDs) 的。

- 主库每次的更新变化都会记录到BinaryLog中，从库可以配置读取BinaryLog，并会记录自己读取到的位置，使得每个从库可以独立工作，即使从库崩溃了以后也能根据位置记录将宕机期间变化的数据同步回来。

分库分表

表中数据过多事导致基本的CRUD都没办法进行，要进行分库分表。先垂直后水平

- 垂直划分：
  - 垂直分库一般比如把用户，商品，订单分为三个库放在三个服务器上方，即不同对象放不同库
  - 垂直分表一般把一个对象下的多个列划分到不同的表，比如不常用的，数据比较长的列都可以划出去
- 水平划分
  - 水平分库，单张很长的表根据范围，hash等作为依据，划分到不同服务器上，每个服务器的库和表都一样，只是数据不一样
  - 水平分表，同一个服务器下根据范围，hash等作为依据，划分成多个表

数据结构

morris遍历二叉树，O(1)空间复杂度，O(n)时间

- 从头结点开始，将头结点当做当前节点
- 若当前节点无左孩子，则打印节点并把其右孩子当做当前节点
- 若当前节点有左孩子
  - 找到对应中序遍历的前驱节点（当前节点左孩子的最右节点）
  - 若这个前驱节点的右孩子为空，则把当前节点作为前驱节点的右孩子。更新当前节点为当前节点的左孩子
  - 若这个前驱节点的右孩子不为空，则打印当前节点，并把前驱节点的右孩子置位空（树的恢复）。更新当前节点为当前节点的右孩子
- 以新的当前节点回到第二步

``

```
/**
 * @author Duan
 * @date 2019/5/1 14:22
 */

/*
输入：
15

0 1 0 0 3 1
1 4 0 3 2 0 3 5 1
2 6 0 2 11 1
```

```
6 14 0 14 8 1
11 10 0 10 7 1
7 9 0 7 13 1
5 12 1
end
```

输出：

```
This is the result of levelTraver
------------------------
0 1 3 4 2 5 6 11 12 14 10 8 7 9 13
This is the result of morrisTraver(inorder traversal)
------------------------
4 1 0 14 8 6 2 10 9 7 13 11 3 5 12
*/

public class Morris {
    public static void main(String[] args) {
        TreeNode head=creatTree();
        levelTraver(head);
        morrisTraver(head);
    }

    private static void morrisTraver(TreeNode head) {
        System.out.println("This is the result of morrisTraver(inorder traversal)");
        System.out.println("------------------------");
        TreeNode node=head;
        while(node!=null){
            if (node.left==null){
                System.out.print(node.data+" ");
                node=node.right;
                continue;
            }else {
                TreeNode preNode=searchPreviouNode(node);
                if (preNode.right==null){
                    preNode.right=node;
                    node=node.left;
                }else {
                    preNode.right=null;
                    System.out.print(node.data+" ");
                    node=node.right;
                }
            }
        }
    }

    private static TreeNode searchPreviouNode(TreeNode nodeNow) {
        TreeNode node=nodeNow.left;
        while (!(node.right==null||node.right.data==nodeNow.data)){
            node=node.right;
        }
        return node;
    }

    private static void levelTraver(TreeNode head) {
        System.out.println("This is the result of levelTraver");
        System.out.println("------------------------");
```

```
        LinkedList<TreeNode> treeNodes=new LinkedList<>();
        treeNodes.offer(head);
        while (!treeNodes.isEmpty()){
            TreeNode treeNode=treeNodes.poll();
            System.out.print(treeNode.data+" ");
            if (treeNode.left != null) {
                treeNodes.offer(treeNode.left);
            }
            if (treeNode.right != null) {
                treeNodes.offer(treeNode.right);
            }
        }
        System.out.println();
    }

    private static TreeNode creatTree() {
        Scanner scanner=new Scanner(System.in);
        System.out.println("How Many nodes are the tree has?");
        int num=scanner.nextInt();
        TreeNode[] tree=new TreeNode[num];
        for (int i=0;i<num;i++){
            tree[i]=new TreeNode();
            tree[i].data=i;
        }
        System.out.println("What is the relationship between nodes?");
        while (scanner.hasNextInt()){
            int treeFaNum=scanner.nextInt();
            int treeSonNum=scanner.nextInt();
            if (scanner.nextInt()==0){
                tree[treeFaNum].left=tree[treeSonNum];
            }else {
                tree[treeFaNum].right=tree[treeSonNum];
            }
        }
        System.out.println();
        return tree[0];
    }
}
class TreeNode{
    int data;
    TreeNode left=null;
    TreeNode right=null;
}
```

A*算法

有F=G+H，G是从起点开始到这个点的代价，H是估计代价（横向距离等）

步骤：

- 把起点加入 open list 。
- 重复如下过程：
  - 遍历 open list ，查找 F 值最小的节点，把它作为当前要处理的节点。
  - 把这个节点移到 close list 。

- 对当前方格的 8 个相邻方格的每一个方格？

  ◆ 如果它是不可抵达的或者它在 close list 中，忽略它。否则，做如下操作。

  ◆ 如果它不在 open list 中，把它加入 open list ，并且把当前方格设置为它的父亲，记录该方格的 F ， G 和 H 值。

  ◆ 如果它已经在 open list 中，检查这条路径 (即经由当前方格到达它那里) 是否更好，用 G 值作参考。更小的 G 值表示这是更好的路径。如果是这样，把它的父亲设置为当前方格，并重新计算它的 G 和 F 值。如果你的 open list 是按 F 值排序的话，改变后你可能需要重新排序。

- 直到到达终点

```
public class AStar {

    public static final int[][] NODES = {
                { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 1, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 1, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 1, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 1, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        };

    public static final int STEP = 10;

    private ArrayList<Node> openList = new ArrayList<Node>();
    private ArrayList<Node> closeList = new ArrayList<Node>();

    public Node findMinFNodeInOpneList() {
        Node tempNode = openList.get(0);
        for (Node node : openList) {
            if (node.F < tempNode.F) {
                tempNode = node;
            }
        }
        return tempNode;
    }

    public ArrayList<Node> findNeighborNodes(Node currentNode) {
        ArrayList<Node> arrayList = new ArrayList<Node>();
        // 只考虑上下左右，不考虑斜对角
        int topX = currentNode.x;
        int topY = currentNode.y - 1;
        if (canReach(topX, topY) && !exists(closeList, topX, topY)) {
            arrayList.add(new Node(topX, topY));
        }
        int bottomX = currentNode.x;
        int bottomY = currentNode.y + 1;
        if (canReach(bottomX, bottomY) && !exists(closeList, bottomX, bottomY)) {
            arrayList.add(new Node(bottomX, bottomY));
        }
        int leftX = currentNode.x - 1;
        int leftY = currentNode.y;
        if (canReach(leftX, leftY) && !exists(closeList, leftX, leftY)) {
```

```
            arrayList.add(new Node(leftX, leftY));
        }
        int rightX = currentNode.x + 1;
        int rightY = currentNode.y;
        if (canReach(rightX, rightY) && !exists(closeList, rightX, rightY)) {
            arrayList.add(new Node(rightX, rightY));
        }
        return arrayList;
    }

    public boolean canReach(int x, int y) {
        if (x >= 0 && x < NODES.length && y >= 0 && y < NODES[0].length) {
            return NODES[x][y] == 0;
        }
        return false;
    }

    public Node findPath(Node startNode, Node endNode) {

        // 把起点加入 open list
        openList.add(startNode);

        while (openList.size() > 0) {
            // 遍历 open list ，查找 F值最小的节点，把它作为当前要处理的节点
            Node currentNode = findMinFNodeInOpneList();
            // 从open list中移除
            openList.remove(currentNode);
            // 把这个节点移到 close list
            closeList.add(currentNode);

            ArrayList<Node> neighborNodes = findNeighborNodes(currentNode);
            for (Node node : neighborNodes) {
                if (exists(openList, node)) {
                    foundPoint(currentNode, node);
                } else {
                    notFoundPoint(currentNode, endNode, node);
                }
            }
            if (find(openList, endNode) != null) {
                return find(openList, endNode);
            }
        }

        return find(openList, endNode);
    }

    private void foundPoint(Node tempStart, Node node) {
        int G = calcG(tempStart, node);
        if (G < node.G) {
            node.parent = tempStart;
            node.G = G;
            node.calcF();
        }
    }

    private void notFoundPoint(Node tempStart, Node end, Node node) {
        node.parent = tempStart;
```

```
        node.G = calcG(tempStart, node);
        node.H = calcH(end, node);
        node.calcF();
        openList.add(node);
    }

    private int calcG(Node start, Node node) {
        int G = STEP;
        int parentG = node.parent != null ? node.parent.G : 0;
        return G + parentG;
    }

    private int calcH(Node end, Node node) {
        int step = Math.abs(node.x - end.x) + Math.abs(node.y - end.y);
        return step * STEP;
    }

    public static void main(String[] args) {
        Node startNode = new Node(5, 1);
        Node endNode = new Node(5, 5);
        Node parent = new AStar().findPath(startNode, endNode);

        for (int i = 0; i < NODES.length; i++) {
            for (int j = 0; j < NODES[0].length; j++) {
                System.out.print(NODES[i][j] + ", ");
            }
            System.out.println();
        }
        ArrayList<Node> arrayList = new ArrayList<Node>();

        while (parent != null) {
            // System.out.println(parent.x + ", " + parent.y);
            arrayList.add(new Node(parent.x, parent.y));
            parent = parent.parent;
        }
        System.out.println("\n");

        for (int i = 0; i < NODES.length; i++) {
            for (int j = 0; j < NODES[0].length; j++) {
                if (exists(arrayList, i, j)) {
                    System.out.print("@, ");
                } else {
                    System.out.print(NODES[i][j] + ", ");
                }

            }
            System.out.println();
        }

    }

    public static Node find(List<Node> nodes, Node point) {
        for (Node n : nodes) {
            if ((n.x == point.x) && (n.y == point.y)) {
                return n;
            }
        }
```

```
        return null;
    }

    public static boolean exists(List<Node> nodes, Node node) {
        for (Node n : nodes) {
            if ((n.x == node.x) && (n.y == node.y)) {
                return true;
            }
        }
        return false;
    }

    public static boolean exists(List<Node> nodes, int x, int y) {
        for (Node n : nodes) {
            if ((n.x == x) && (n.y == y)) {
                return true;
            }
        }
        return false;
    }

    public static class Node {
        public Node(int x, int y) {
            this.x = x;
            this.y = y;
        }

        public int x;
        public int y;

        public int F;
        public int G;
        public int H;

        public void calcF() {
            this.F = this.G + this.H;
        }

        public Node parent;
    }
}
```

- 红黑树，B，B+树需要进行了解。

操作系统

进程与线程的区别

- 进程是拥有资源的基本单位，线程只拥有一点运行中必不可少的资源（堆栈，寄存器，优先级），线程是调度的基本单位
- 进程之间可以并发，一个进程的线程之间可以并发
- 创建和撤销进程的开销比线程大很多（内存空间，IO设备）

作业调度算法

先来先服务；短作业优先；优先权调度；高响应比优先调度（动态优先级），时间轮转，多级时间轮转

进程通信方式

1. **管道**：管道是一种**半双工的通信**方式，数据只能单向流动，而且只能在**具有亲缘关系**的进程间使用。进程的亲缘关系通常是指父子进程关系。管道是单向的、先进先出的、无结构的、固定大小的字节流，它把一个进程的标准输出和另一个进程的标准输入连接在一起。写进程在管道的尾端写入数据，读进程在管道的首端读出数据。
2. **命名管道 (named pipe)**： 命名管道也是半双工的通信方式，它克服了管道没有名字的限制，并且它允许**无亲缘关系进程间**的通信。命令管道在文件系统中有对应的文件名，命名管道通过命令mkfifo或系统调用mkfifo来创建。
3. **信号量(semophore)**： 信号量是一个计数器，可以用来控制多个进程对**共享资源**的访问。它常作为一种**锁机制**，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
4. **消息队列(message queue)**： 消息队列是由**消息的链表**结构实现，存放在内核中并由消息队列标识符标识。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
5. **信号 (sinal)**：用于通知接收进程某个事件已经发生。除了用于进程通信外，进程还可以发送信号给进程本身。
6. **共享内存(shared memory)**：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的IPC方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量配合使用，来实现进程间的同步和通信。
7. **套接字(socket)**： 也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。

同一进程下的线程使用的是同一个地址空间，可以直接通信，做好互斥就行，用锁，信号量，信号通信。不同进程下的线程通信其实就是进程的通信了。

死锁条件

资源互斥等待；不可剥夺（资源不可抢占）；请求和保持（阻塞不放）；循环等待

分页分段与缺页调度

分页：页大小相等，没有外部碎片，程序不需要连续存放，但有内部碎片，不能体现程序逻辑

分段：段大小不等，程序逻辑性强，动态增长，会产生外部碎片，不如页号映射方便

段页式：不等的段分为了多个等长的页，段页号映射。管理方便，映射方便，存在内部碎片，抖动（频繁的缺页调度）

缺页调度算法：先来先出，最优置换（理论），最近最少使用（LRU），时钟算法（二次机会改进）

磁盘调度

先来先服务；最短寻道优先；扫描算法；循环扫描算法

Redis

基础数据结构

- 字符串：记录字符串长度，常数复杂度获取字符串长度，杜绝缓冲区溢出，减少修改字符串带来的内存重分配次数（空间预分配，惰性空间释放），二进制安全，兼容部分C字符串函数。
- 链表：双端无环带计数器链表，值多态，发布订阅，慢查询，缓冲区都能用
- 字典：两个数组（rehash用），链地址法，头插，扩容是两倍扩充，rehash分普通和渐进式（负担因子要根据是否在进行写rdb和重写aof（创建子进程，子进程会写时复制）决定（1和5）），收缩0.1。
- 跳跃表：每个成员都有分值，同分值成员用分值决定顺序。
- 整数集合：有序存放，用一个属性记录数据类型（最大数决定），升级，没降级
- 压缩列表：记录了字节数和位节点的位置，可以正序遍历，每个节点保存字节数组或整数值，每个节点会记录前面节点的大小（用于从后往前遍历），接着记录本节点数据类型及数据内容，连锁更新现象（由于记录前一个节点大小的单元有一字节和五字节两种）

对象

redisObject结构，存类型（什么对象）和编码（什么数据结构）及指向结构的指针

- 字符串对象（STRING）：整数long，raw（实体字符串和redisobject不在一起），embstr（连续存放）。
- 列表对象（LIST）：链表或压缩列表，数量小且单个数据小用压缩列表
- 哈希对象（HASH）：字典或压缩列表，数量小且单个数据小用压缩列表，按键在前，值在后排放
- 集合对象（SET）：字典或整数集合，数量小且全是整数用整数集合。
- 有序集合对象（ZSET）：跳跃表+字典或压缩列表。数量小且全是整数用压缩列表，按照成员在前分值在后挨着排放，且所有数据按照分值从小到大排序。跳跃表+字典：成员为键，分值为值，O(1)找到成员的分值，跳跃表中有序（zrank，zrange有利）存放节点，维持数据的有序存放。存放指针不费空间。
- 注：一个对象的指令能操作多种底层数据结构体现了多态性，链表中也体现了多态

缓存一致性问题

由于并发的访问缓存，导致缓存数据与实际物理数据库的不一致情况

先更新数据库再更新缓存：多个请求进行更新操作时，先修改数据库再相应的修改缓存，可能由于网络等原因导致缓存数据与物理数据不一致，且不使用于写请求频繁的场景

先删缓存，再更新数据库：当出现两个请求，A更新，B读取。执行顺序为：A删除缓存，B读取硬盘读到旧数据，B把旧数据写入缓存，A把数据写入数据库。这个顺序下来就导致缓存与数据库的数据不一致。

所以使用**延时双删策略**：先删除缓存，更新数据库，延时，再次删除缓存，就能大概率避免上面说的由于另一个线程的读更新了缓存导致的数据不一致（删除失败时，可以把删除失败的key放入消息队列，另写代码去删除）

先更新数据库，再删缓存：可能缓存刚好失效，请求a查询数据库得到旧值，b更新数据库再更新缓存，a更新缓存，导致数据不一致（但发送概率很低，这样做是最优的）

缓存穿透与缓存雪崩

- 缓存穿透
  - 大量恶意访问。且访问的key对应的value是不存在的，所以访问全部会落到数据库中，导致缓存无用，持久层压力过大
  - 解决方法
    - 压力过大时，在缓冲中建立过期时间比较短的对应键的null，来缓解持久层的压力
    - 使用布隆过滤器（bitmap来存数据对应的hash，1表示存在，0表示不存在），过滤掉那些访问不存在的数据的请求
- 缓存雪崩（缓冲击穿对于单热点key的过期）
  - 缓冲集中在一段时间内大量失效，发生了大量的缓冲穿透（比如缓冲宕机了）
  - 解决方法
    - 雪崩后在持久层访问时加锁或使用中间件控制写缓冲的线程容量
    - key的过期时间不要设置的密集于同一时间
    - 双缓冲策略

数据库

16个，每个数据库两个字典，一个记录存的键与值，一个存的键和过期时间，对于过期键，使用定期删除和惰性删除两种结合（过期键不会保存到rdb；载入rbd主服务器忽略过期键，从服务器不会，但空间会被主服务器同步时清空；键过期时会写一条删除进AOF，AOF重写不会重写过期键），主服务器删除会通知从服务器，不然从服务器不删。

持久化

RDB和AOF，rdb三种产生方式（配置文件，flushdb，save（阻塞进程）和bgsave（生成子进程），bgsave拒bgsave和save，延迟aof重写。aof重写拒绝所有。重写aof会重写键值对和过期时间（aof重写可以停掉线程重写，也可以创子进程重写，但需要建立重写缓冲区）

rdb适合大规模数据恢复，且宕机丢失更多，fork会克隆一份数据，aof文件比rdb大，aof运行效率比rdb慢

事件

文件事件：客户端通过套接字与服务器进行连接通讯，**文件事件是服务器对套接字进行操作的抽象**。文件事件处理器使用多路IO复用监听多个套接字，并根据任务来关联不同的文件处理器（连接应答，命令请求，命令回复），客户端和服务器都有缓冲区，实现通信

时间事件：服务器内部定时执行的任务或循环执行的任务。（serverCron函数周期执行：资源管理，请过期，尝试初始化，集群定期检查连接）

事件调度：为时间事件设置一个定时阻塞，等待文件事件并执行，如此反复。

客户端和服务器

客户端和服务器都有对应的数据结构进行描述，包括基本信息，各种功能，身份验证，套接字，缓冲区，指令及参数储存等。客户端把命令写到自己的缓冲区。然后从缓冲区发给服务器套接字，服务器启动读事件，读取套接字内容，写到自身数据结构中，对应分拆为指令，参数个数，参数，指令会寻找指令表指向对应的函数，执行时访问函数并传参。返回内容写到客户端的输出缓冲区中等待客户端读取。

复制

旧版同步，每次同步都是完整全同步，主生成rdb文件给从，从使用rdb进行同步，主每次也发自己的修改操作给从执行。但从掉线重连以后还是要完整重同步，可能只少了一部分，非常浪费双方的资源。

改良：主内部维护一个环形缓冲区，并为每一块带上复制偏移量，内部存储一条修改命令，从连接时带上之前连的主服务器信息和自己当前的复制偏移量，若第一次连还是完整重同步，重连的会查看是不是本机的从服务器，是的话看对应复制偏移量是否还在缓冲区中，有的话就只发送对应缺失的部分，已经被覆盖了就完整全同步。

从服务器可以通过指令主动请求复制主服务器，主从服务器都会记录自己的状态和自己的主（从）服务器的信息，定时进行心态检测（从服务器发自己的复制偏移量给主服务器，表示自己或者顺便检测指令丢失）

redis死锁

为redis加锁，加锁直接在redis上创建key，作为锁，使用setnx创建保证建锁的原子性，但如果客户端异常退出，没有解锁就导致了死锁，所以一般对作为锁的key需要设置过期时间来避免死锁，可直接对键设置过期时间，也可以用value保存过期时间，读取时查看value对应的时间是否过期，过期了就能使用getset操作拿锁，若执行期间被其他线程抢到了锁，就会重新去读取查看是否过期。

利用储存时间还有致命的缺陷，当线程A时间过期时，本步操作还没完成，锁过期被B线程加锁。导致出现了不安全的情况，并且A线程这个时候执行完了操作释放掉了B的锁，导致C线程的加锁进入线程不安全状态。对于后面这种，可以对键值对设置过期时间，值设置为加锁线程的ip和端口，每个线程释放时只能是释放掉和当前ip与端口一样的线程。但由于操作时间大于设置的过期时间导致进入不安全状态仍然不可避免

redis淘汰缓存策略

- volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰,
- volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
- volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
- allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

- allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
- no-enviction（驱逐）：禁止驱逐数据

lru适用于键的访问率不同的，random适用于键访问率差不多时。

哨兵

哨兵的数据结构也是服务器的数据结构，但很多功能不使用，哨兵会成为主服务器的一个客户端，记录主服务器的信息，创建了两个连接（一个给主服务器发命令，一个接收哨兵间的消息）

每个十秒，哨兵就会去获取一次主服务器的信息，并拿到主从服务器的信息，更新自己的主从服务器的字典。哨兵也以十秒一次的频率去获取从服务器的消息。哨兵向主服务器发送请求的同时也会被其他哨兵收到，通过服务器为中转，监视同一个服务器的哨兵就发现了彼此的存在并更新自己对其他哨兵的认知（反应到数据结构上就是更新sentinels字典），同时哨兵也会与其他哨兵间建立命令连接。

哨兵在发现主机疑似下线时（获取不到信息），会发命令询问其他哨兵对主服务器下线状态的判断，达到足够数量会判断主服务器客观下线，此时会选举领头哨兵（主机拉票，每个哨兵会把票给第一个给他发送拉票信息的哨兵，那个哨兵票数过半就当领头哨兵，没有票数过半的重投）。领头哨兵对主服务进行故障转移，并从从服务器中选择一个成为新的主服务器。旧的主服务器重新上线后成为从服务器

集群

集群构建：客户端给服务器A发送指令，让它连接B；A给B发消息请求连接ping；B回复消息确认pong；最后A再次回复消息给Bping确认。

槽指派：集群用分片的方式保存数据库中的键值对，数据库被分为16384个槽，每个键都是属于一个槽，每个槽都有多个点，每个节点负责其中任意多个槽。每个节点都会维持一个char数组16384/8，用位为1表示当前节点负责这个槽，也维护一个16348的指针数组，对应下标的指针指向这个下标值对应槽负责的节点。

重新分片，会将任意多个已经指派给某个节点的槽改为指派到另一个节点，若此时访问正在移动的键，若键已经搬走了，那么会给客户端返回ASK错误，并引导客户端到正确的节点，访问正确客户端之前会执行一个ASKING命令，让客户端带上标记，只有带上标记的客户端才能在正确的节点中拿到搬运中的节点，拿到后关闭这个标记。

集群中的所有节点都是互相联系，并定时彼此交换信息，若某个主节点检测到另一个主节点疑似下线后，会给其他节点发送消息，其他主节点检测后，半数认为其下线，就会在该主节点的从节点中选择一个成为主节点，并负责下线主节点的槽指派，选举方式还是从节点给其他主节点发拉票请求，主节点把票给第一个发消息的从节点，有从节点拿到半数以上就成为主节点，不然重新投票。