一、网络和HTTP

1.1 OSI 层级关系

应用层 文件传输，电子邮件，文件服务，虚拟终端 TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet

表示层 数据格式化，代码转换，数据加密 没有协议

会话层 解除或建立与别的接点的联系 没有协议

传输层 提供端对端的接口 TCP，UDP

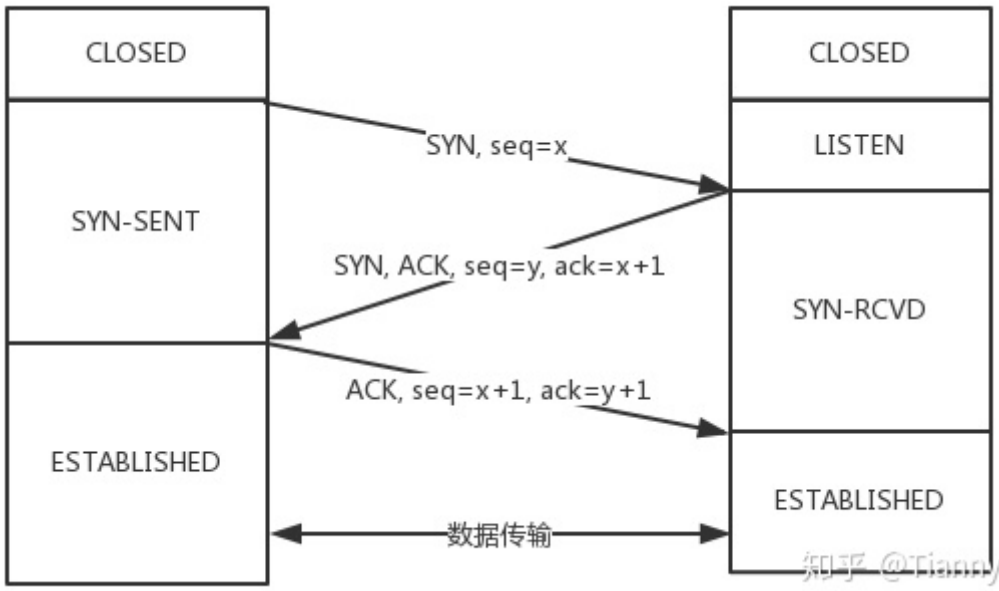
网络层 为数据包选择路由 IP，ICMP，RIP，OSPF，BGP，IGMP

数据链路层 传输有地址的帧以及错误检测功能 SLIP，CSLIP，PPP，ARP，RARP，MTU

物理层 以二进制数据形式在物理媒体上传输数据 ISO2110，IEEE802，IEEE802.2

1.2 TCP 三次握手

重点：**状态转化** 发送的内容



- ①首先 Client 端向Server端发送连接请求报文，这时，Client 进入SYN-SENT（同步已发送）状态。
- ②Server 段接受连接后回复 ACK 报文，并为这次连接分配资源。
- ③Client 端接收到 ACK 报文后也向 Server 段发生 ACK 报文，并分配资源，这样 TCP 连接就建立了。

1.3 HTTP

请求头

- 状态行　[GET|POST]　[URL]　[HTTP/1.1]
- 请求头　Host User-Agent 等
- 消息主体

返回

- 状态行　HTTP/1.1 200 OK
- 响应头
- 响应正文
- HTTP/1.1是目前互联网上使用最广泛的协议，功能也非常完善；

- HTTP/2基于Google的SPDY协议，注重性能改善，但还未普及； 【多路复用，请求头，响应头二进制传输 HTTPS 】
- HTTP/3基于Google的QUIC协议，是将来的发展方向。

1.4 浏览器输入一个url 中间经历的过程

DNS查询（是否有缓存）-》TCP握手》http 请求》反向nginx》uwsgi/gunicorn》 web app响应》TCP挥手

1. 浏览器地址栏输入url
2. 浏览器先查看DNS缓存，如果有跳过第三步
3. DNS 域名解析，解析相应的IP地址
4. 浏览器向服务器发起tcp链接，与服务器创建tcp三次握手。
5. 握手成功后，浏览器向服务器发送http请求，请求数据包
6. 服务器处理收到的请求，将数据返回至浏览器
7. 浏览器根据相应返回数据渲染页面

1.4.1 什么是RESTful

一种软件架构的风格， 就是用URL定位资源，用HTTP METHOD 描述操作。

Rosources 资源

Representation 表现层

State Transfer 状态转化 使用GET POST PUT DELETE http 动词来操作资源，实现资源状态的改变

1.4.2 web安全

1. **SQL 注入** 没有对输入进行过滤，直接动态拼接SQL

   防范： 永远不要相信用户的任何输入

   对输入参数做好检查；过滤和转义特殊字符

   不要直接拼接SQL,使用ORM 可以降低SQL 注入的风险

   数据库层：做好权限管理配置；不要明文存储敏感信息

2. **XSS 跨站脚本攻击** 插入未经转换的js 代码，其他用户也会访问他 比如**评论**

   黑客获取cookie, 盗取敏感数据

3. **CSRF 跨站请求伪造**

   首先用户C**浏览并登录了受信任站点A**；

   登录信息验证通过以后，站点A会在返回给浏览器的信息中带上已登录的**cookie**，cookie信息会在浏览器端保存一定时间（根据服务端设置而定）；

   完成这一步以后，用户在没有登出（清除站点A的cookie）站点A的情况下，访问**恶意站点B**；

   这时恶意站点 **B的某个页面向站点A发起请求，而这个请求会带上浏览器端所保存的站点A的cookie**；

   站点A根据请求所带的cookie，判断此请求为用户C所发送的。

   危害： 数据泄露 或者数据安全

   防御：

   1. 尽量使用POST 限制使用GET, GET 接口可以构造img 标签等
   2. 将cookie设置为HttpOnly
   3. 添加token
   4. 通过Referer 识别

1.5 IO复用之select，poll，epoll

select

对于给定的描述符进行轮询，当有描述符就绪时返回，返回值是就绪描述符的数目，select返回后，用户程序需要遍历确定给定的描述符中哪一个已经就绪，从而执行后面的操作。select模型每次调用都会讲描述符的内容（一个数组）从用户空间拷贝到内核空间，当数量较大时，效率会很低。 此外，select会轮询遍历所有的关注的描述符，描述符数量较大时效率也会很低。

poll

内部实现与select基本相同，只是传入参数的方式不同，而且会把就绪的描述符放置在一个参数中返回，用户程序可以遍历这个数组以得到就绪的描述符

epoll

将用户关心的文件描述符的时间存放在内核中的一个事件表中，用户空间和内核空间只需要复制一次。当某个描述符就绪时，内核会采用类似callback的回调机制（软件中断）迅速激活这个文件描述符，当进程调用epoll_wait()时便得到通知

二、python 常见问题集合

1.函数装饰器有什么作用？请列举说明?

1，引入日志 2，函数执行时间统计3，执行函数前预备处理4，执行函数后清理功能5，权限校验等场景6，缓存7，事务处理

2.Python中变量的作用域？ （变量查找顺序)

函数作用域的LEGB顺序

1.什么是LEGB?

L： local 函数内部作用域

E: enclosing 函数内部与内嵌函数之间

G: global 全局作用域

B： build-in 内置作用

python在函数里面的查找分为4种，称之为LEGB，也正是按照这是顺序来查找的

4.Python的内存管理机制及调优手段?

gc模块 https://docs.python.org/3.7/library/gc.html

```
gc.disable()  # 暂停自动垃圾回收.
gc.collect()  # 执行一次完整的垃圾回收, 返回垃圾回收所找到无法到达的对象的数量.
gc.set_threshold()  # 设置Python垃圾回收的阈值.
gc.set_debug()  # 设置垃圾回收的调试标记. 调试信息会被写入std.err.
```

5.Python有两种共存的内存管理机制: 引用计数和垃圾回收

垃圾回收机制:

1. 引用计数 PyObject

   python里每一个东西都是对象，它们的核心就是一个结构体：PyObject

   PyObject是每个对象必有的内容，其中ob_refcnt就是做为引用计数。当一个对象有新的引用时，它的ob_refcnt就会增加，当引用它的对象被删除，它的ob_refcnt就会减少

   引用计数也是一种垃圾收集机制，而且也是一种最直观，最简单的垃圾收集技术。当 Python 的某个对象的引用计数降为 0 时，说明没有任何引用指向该对象，该对象就成为要被回收的垃圾了。比如某个新建对象，它被分配给某个引用，对象的引用计数变为 1。如果引用被删除，对象的引用计数为 0，那么该对象就可以被垃圾回收。不过如果出现循环引用的话，引用计数机制就不再起有效的作用了

2. 标记清除

   标记-清除机制，顾名思义，首先标记对象（垃圾检测），然后清除垃圾（垃圾回收）。

   首先初始所有对象标记为白色，并确定根节点对象（这些对象是不会被删除），标记它们为黑色（表示对象有效）。

   将有效对象引用的对象标记为灰色（表示对象可达，但它们所引用的对象还没检查），检查完灰色对象引用的对象后，将灰色标记为黑色。

   重复直到不存在灰色节点为止。最后白色结点都是需要清除的对象。

3. 分代回收

   从前面“标记-清除”这样的垃圾收集机制来看，这种垃圾收集机制所带来的额外操作实际上与系统中总的内存块的数量是相关的，当需要回收的内存块越多时，垃圾检测带来的额外操作就越多，而垃圾回收带来的额外操作就越少；反之，当需回收的内存块越少时，垃圾检测就将比垃圾回收带来更少的额外操作

   1、新创建的对象做为0代

   2、每执行一个【标记-删除】，存活的对象代数就+1

   3、代数越高的对象（存活越持久的对象），进行【标记-删除】的时间间隔就越长。这个间隔，江湖人称阀值。

4. 三种情况触发垃圾回收 1、调用gc.collect() 2、GC达到阀值时 3、程序退出时

调优手段:

```
1.手动垃圾回收

2.调高垃圾回收阈值

3.避免循环引用
```

6. 进程

进程间不共享全局变量

进程之间的通信-Queue

7. 简述Django请求生命周期

一般是用户通过浏览器向我们的服务器发起一个请求(request),这个请求会去访问视图函数，如果不涉及到数据调用，那么这个时候视图函数返回一个模板也就是一个网页给用户） 视图函数调用模型毛模型去数据库查找数据，然后逐级返回，视图函数把返回的数据填充到模板中空格中，最后返回网页给用户。

1.wsgi ,请求封装后交给web框架（Flask，Django)

2.中间件，对请求进行校验或在请求对象中添加其他相关数据，例如：csrf,request.session

3.路由匹配 根据浏览器发送的不同url去匹配不同的视图函数

4.视图函数，在视图函数中进行业务逻辑的处理，可能涉及到：orm，templates

5.中间件，对响应的数据进行处理

6.wsgi，将响应的内容发送给浏览器

8.__new__和__init__的区别

这个__new__确实很少见到,先做了解吧.

1. __new__是一个静态方法,而__init__是一个实例方法.
2. __new__方法会返回一个创建的实例,而__init__什么都不返回.
3. 只有在__new__返回一个cls的实例时后面的__init__才能被调用.
4. 当创建一个新实例时调用__new__,初始化一个实例时用__init__.

9.单例模式

这个绝对常考啊.绝对要记住1~2个方法,当时面试官是让手写的.

9.1 使用__new__方法

```
class Singleton2(object):
    __isinstance = False
    def __new__(cls, *args, **kwargs):
        if cls.__isinstance:
            return cls.__isinstance
        cls.__isinstance = object.__new__(cls)
        return cls.__isinstance
```

9.2 共享属性

创建实例时把所有实例的__dict__指向同一个字典,这样它们具有相同的属性和方法.

```
class Borg(object):
    _state = {}
    def __new__(cls, *args, **kw):
        ob = super(Borg, cls).__new__(cls, *args, **kw)
        ob.__dict__ = cls._state
        return ob

class MyClass2(Borg):
    a = 1
```

9.3 装饰器版本

```
def singleton(cls, *args, **kw):
    instances = {}
    def getinstance():
        if cls not in instances:
            instances[cls] = cls(*args, **kw)
        return instances[cls]
    return getinstance

@singleton
class MyClass:
  ...
```

9.4 import方法

作为python的模块是天然的单例模式

```
# mysingleton.py
class My_Singleton(object):
    def foo(self):
        pass

my_singleton = My_Singleton()

# to use
from mysingleton import my_singleton

my_singleton.foo()
```

10. 进程 线程 协程

简单点说协程是进程和线程的升级版,进程和线程都面临着内核态和用户态的切换问题而耗费许多切换时间,而协程就是用户自己控制切换的时机,不再需要陷入系统的内核态.

进程：一个运行的程序（代码）就是一个进程，没有运行的代码叫程序，进程是系统资源分配的最小单位，进程拥有自己独立的内存空间，所有进程间数据不共享，开销大。

线程: cpu调度执行的最小单位，也叫执行路径，不能独立存在，依赖进程存在，一个进程至少有一个线程，叫主线程，而多个线程共享内存（数据共享，共享全局变量),从而极大地提高了程序的运行效率。

协程: 是一种用户态的轻量级线程，协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。协程调度时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

11.Cookie和Session

	Cookie	**Session**
储存位置	客户端	服务器端
目的	跟踪会话，也可以保存用户偏好设置或者保存用户名密码等	跟踪会话
安全性	不安全	安全

session技术是要使用到cookie的，之所以出现session技术，主要是为了安全

三、数据库

3.1 事务

数据库并发控制的基本单位，可以看做一系列SQL语句的集合

事务必须要么全部执行成功，要么全部执行失败（回滚）

例子：转账操作

3.2 事务的四个特性

ACID是事务的四个特性

原子性Atomicity：一个事务中所有操作全部完成或失败（回滚）

一致性Consistency:事务开始和结束之后数据完整性没有被破坏（转账的例子：转到钱没有凭空变多或消失）

拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

隔离性Isolation:允许多个事务同时对数据库修改和读写

持久性Durability：事务结束后，修改是永久不会丢失

3.3 事务并发的隔离性

脏读（读取未提交的数据）

A事务读取B事务**尚未提交**的数据，此时如果B事务发生错误并执行**回滚**操作，那么A事务读取到的数据就是脏数据。就好像原本的数据比较干净、纯粹，此时由于B事务更改了它，这个数据变得不再纯粹。这个时候A事务立即读取了这个脏数据，但事务B良心发现，又用回滚把数据恢复成原来干净、纯粹的样子，而事务A却什么都不知道，最终结果就是事务A读取了此次的脏数据，称为脏读。

这种情况常发生于转账与取款操作中

时间顺序	转账事务	取款事务
1		开始事务
2	开始事务	
3		查询账户余额为2000元
4		取款1000元，余额被更改为1000元
5	查询账户余额为1000元（产生脏读）	
6		取款操作发生未知错误，事务回滚，余额变更为2000元
7	转入2000元，余额被更改为3000元（脏读的1000+2000）	
8	提交事务	
备注	按照正确逻辑，此时账户余额应该为4000元	

不可重复读（前后多次读取，数据内容不一致）

事务A在执行读取操作，由整个事务A比较大，前后读取同一条数据需要经历很长的时间 。而在事务A第一次读取数据，比如此时读取了小明的年龄为20岁，事务B执行更改操作，将小明的年龄更改为30岁，此时事务A第二次读取到小明的年龄时，发现其年龄是30岁，和之前的数据不一样了，也就是数据不重复了，系统不可以读取到重复的数据，成为不可重复读。

时间顺序	事务A	事务B
1	开始事务	
2	第一次查询，小明的年龄为20岁	
3		开始事务
4	其他操作	
5		更改小明的年龄为30岁
6		提交事务
7	第二次查询，小明的年龄为30岁	
备注	按照正确逻辑，事务A前后两次读取到的数据应该一致	

幻读（前后多次读取，数据总量不一致）

事务A在执行读取操作，需要两次统计数据的**总量**，前一次查询数据总量后，此时事务B执行了新增数据的操作并提交后，这个时候事务A读取的数据总量和之前统计的不一样，就像产生了幻觉一样，平白无故的多了几条数据，成为幻读。

时间顺序	事务A	事务B
1	开始事务	
2	第一次查询，数据总量为100条	
3		开始事务
4	其他操作	
5		新增100条数据
6		提交事务
7	第二次查询，数据总量为200条	
备注	按照正确逻辑，事务A前后两次读取到的数据总量应该一致	

不可重复读和幻读到底有什么区别呢?

(1) 不可重复读是读取了其他事务更改的数据，**针对update操作**

解决：使用行级锁，锁定该行，事务A多次读取操作完成后才释放该锁，这个时候才允许其他事务更改刚才的数据。

(2) 幻读是读取了其他事务新增的数据，**针对insert和delete操作**

解决：使用表级锁，锁定整张表，事务A多次读取数据总量之后才释放该锁，这个时候才允许其他事务新增数据。

数据库的隔离级别

数据库事务的隔离级别有4个，由低到高依次为

读未提交(Read uncommitted):别的事务可以读取到未提交的改变

读已提交(Read committed):别的事务只能读取到已经提交的改变

可重复读(Repeatable read): 同一个事务先后查询结果一样

串行化(Serializable) :事务完全串行化的执行，隔离级别最高，执行效率最低

	脏读	不可重复读	幻读
Read uncommitted	√	√	√
Read committed	×	√	√
Repeatable read	×	×	√
Serializable	×	×	×

MySQL 的默认隔离属性为Repeatable read　（可重复读）

3.4 乐观锁与悲观锁

悲观锁

悲观锁是先获取锁再进行操作。一锁二查三更新 select for update

```
sqlalchemy: Address.query.filter_by(user_id=3).with_for_update().first()
```

乐观锁

乐观锁先修改，更新的时候发现数据已经变了就回滚（一般使用版本号和时间戳）

1. 乐观锁 适合查多改少，经常被并发修改的数据可能老是出现版本不一致导致有的线程操作常常失败。
2. 悲观锁 适合短事务（长事务导致其它事务一直被阻塞，影响系统性能），查少改多。

3.5 InnoDB vs MyISAM

常见区别	MyISAM	Innodb
事务	不支持	支持
外键	不支持	支持
锁支持（锁是避免资源争用的一个机制，MySQL锁对用户几乎是透明的）	只支持表级锁定	行级锁定、表级锁定，锁定力度小并发能力高
全文索引	支持	不支持

3.6 什么是索引?

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询和排序。索引的实现通常使用B-树或其变种的B+树。

3.7 为什么要创建索引?

- 通过建立唯一索引，可以保证数据库表中每一行数据的唯一性；
- 可以大大加快数据检索速度（主要原因）；
- 可以加速表与表之间的连接，特别是在实现数据的参考完整性方面有重要作用；
- 可以在检索数据时，大大减少查询中分组和排序花费的时间；
- 使用索引可以在查询过程中使用优化隐藏器，提高系统性能。

3.8 索引存在不利的方面

- 创建和维护索引需要耗费时间，且会随着数据量的增加而增加；
- 创建索引需要占物理空间，聚集索引占空间更大；
- 对表中的数据进行增删改时，索引也需要动态维护。

3.9 索引的使用场景

需要创建索引	不应该创建索引
1.经常需要搜索的列	1.查询中很少使用或者参考的列
2.经常用在连接上的列	2.只有很少数值的列（如：性别）
3.作为主键的列	3.定义为text, image, bit数据类型的列
4.经常需要根据范围搜索的列	4.修改性能远远大于检索性能时
5.经常需要排序的列	
6.经常在Where子句中的列	

3.10 索引的分类

- **普通索引**

  最基本的数据库索引，没有任何限制；

- **唯一索引**

  要求索引列的值必须唯一，但允许空值；

- **主键索引**

是一种特殊的唯一索引，不允许空值，一般在创建表的时候同时创建主键索引；

- **聚集索引**

  聚集索引中，各行的物理顺序与索引键值的逻辑顺序一致，且每张表最多只能有一个聚集索引。创建聚集索引要花更长的时间。（A,B,C）相当于分别建立了（A,B,C），（A,B）, (A) 三种组合的索引。

聚集与否

类别	解释
聚集索引	表数据按照索引的顺序来存储，叶子节点即存储了真实的数据行
非聚集索引	表数据存储与索引顺序无关，叶子节点包含索引字段值及指向数据页数据行的逻辑指针

Mysql索引分类 复杂维度

索引是在Mysql的存储引擎层实现的，而不是在服务层实现的。所以每种存储引擎的索引并不完全相同

类别	解释
B-Tree索引	最常见的索引类型，大部分引擎都支持B-Tree索引
Hash索引	即使用列的Hash值作为索引查找，只用Memory引擎支持，使用场景简单
R-Tree索引	MyISAM的一种特殊索引，主要用于地理空间数据类型
Full-text	全文索引，MyISAM 和 InnoDB 支持

3.12 为什么索引数据结构选择B-树或B+树?

索引文件很大，一般存储在磁盘上，而磁盘I/O十分耗时。使用索引的过程中要尽可能地减少查找过程中磁盘I/O的次数。B+树作为多路查找树，高度小，磁盘I/O次数少。数据库系统巧妙地应用磁盘预读原理，将一个结点的大小设置为一页，所以每个节点一次I/O就可以完全载入。而红黑树类似的结构，高度更深，且逻辑上很近的节点物理上也可能很远，无法利用局部性。

3.13 如何创建索引?

```
CREATE INDEX indexName ON mytable(username(length));
     //普通索引
CREATE UNIQUE INDEX indexName ON mytable(username(length));
    //唯一索引
CREATE TABLE mytable(ID INT NOT NULL,username VARCHAR(16) NOT NULL,PRIMARY
KEY(ID) ); //主键索引
ALTER TABLE mytable ADD INDEX indexName (username(16),time(10))
    //聚集索引
```

3.14 何时会发生索引失效?

- 隐式转换导致索引失效;
- 对索引列进行计算导致索引失效;
- like "%_" 百分号在前;
- 索引字段进行判空查询时;
- 复合索引中的前导列没有被作为查询条件;
- 判断索引列是否不等于某个值时。

3.15 MySQL存储引擎

特点	MyISAM	InnoDB
存储限制	无	64TB
事务安全		支持
锁机制	表锁	行锁
B树索引	支持	支持
哈希索引		支持
全文索引	支持	
数据缓存		支持
支持外键		支持
操作效率	高	低

备注：最新版本的MySQL默认的存储引擎是InnoDB。count(*)操作，因为MyISAM内置了计数器，所以更快，InnnoDB需要扫描全表。

3.16 MySQL调优

MySQL数据库是常见的两个瓶颈是CPU和I/O的瓶颈，CPU在饱和的时候一般发生在数据装入内存或从磁盘上读取数据时候。磁盘I/O瓶颈发生在装入数据远大于内存容量的时候。对于MySQL系统本身，可以使用工具来优化数据库的性能，通常有三种：使用索引，使用EXPLAIN分析查询以及调整MySQL的内部配置。

在优化MySQL时，通常需要对数据库进行分析，常见的分析手段有**慢查询日志**，**EXPLAIN 分析查询**，**profiling分析**以及show命令查询系统状态及系统变量，通过定位分析性能的瓶颈，才能更好的优化数据库系统的性能。

通过慢日志查询可以知道哪些SQL语句执行效率低下，通过explain我们可以得知SQL语句的具体执行情况，索引使用等，还可以结合show命令查看执行状态。

通过explain命令可以得到:

- 表的读取顺序
- 数据读取操作的操作类型
- 哪些索引可以使用
- 哪些索引被实际使用
- 表之间的引用
- 每张表有多少行被优化器查询

如果觉得explain的信息不够详细，可以同通过profiling命令得到更准确的SQL执行消耗系统资源的信息。

3.17 MySQL 对于千万级的大表要怎么优化

可依次进行以下优化:

- 优化SQL和索引
- 应用缓存机制 (memcached,redis)
- 进行主从复制或主主复制，读写分离
- 应用MySQL自带的分区表
- 垂直拆分
- 水平拆分

3.18 MySQL数据库复制（主从、主主）

MySQL复制解决的问题

- 数据分布
- 负载平衡
- 备份
- 高可用性和容错行

主从复制:

主从复制的原理:

分为同步复制和异步复制，实际复制架构中大部分为异步复制。 复制的基本过程如下:

1).Slave上面的IO进程连接上Master，并请求从指定日志文件的指定位置（或者从最开始的日志）之后的日志内容;

2).Master接收到来自Slave的IO进程的请求后，通过负责复制的IO进程根据请求信息读取制定日志指定位置之后的日志信息，返回给Slave 的IO进程。返回信息中除了日志所包含的信息之外，还包括本次返回的信息已经到Master端的bin-log文件的名称以及bin-log的位置;

3).Slave的IO进程接收到信息后，将接收到的日志内容依次添加到Slave端的relay-log文件的最末端，并将读取到的Master端的 bin-log的文件名和位置记录到master-info文件中，以便在下一次读取的时候能够清楚的告诉Master“我需要从某个bin-log的哪个位置开始往后的日志内容，请发给我”；

4).Slave的Sql进程检测到relay-log中新增加了内容后，会马上解析relay-log的内容成为在Master端真实执行时候的那些可执行的内容，并在自身执行。

对于主从复制，Mysql数据库的版本，两个数据库版本要相同，或者slave比master版本低。

主主复制

互为对方的从服务器，每台服务器即是对方的主服务器，又是对方的从服务器。但可能会面对主键重复等问题。

3.19 读写分离

读写分离，基本的原理是让主数据库(Master)处理事务性增、改、删操作（INSERT、UPDATE、DELETE），而从数据库(Slave)处理SELECT查询操作。数据库复制被用来把事务性操作导致的变更同步到集群中的从数据库。

通过数据库**主从复制**实现读写分离，

为什么读写分离可以提高提高性能?

- 物理服务器增加，负荷能力增加
- 主从只负责各自的写和读，极大程度的缓解X锁和S锁争用
- 从库可配置myisam引擎，提升查询性能以及节约系统开销
- 读写分离适用与读远大于写的场景
- 分摊读取
- 可以增加冗余，提高可用性(一台数据库服务器宕机后能通过调整另外一台从库来以最快的速度恢复服务)

3.20 数据库分库分表策略

通过集群方案，解决了数据库宕机带来的单点数据库不能访问的问题；通过读写分离策略更是最大限度了提高了应用中读取 （Read）数据的速度和并发量。水平切分数据库，可以降低单台机器的负载，同时最大限度的降低了了宕机造成的损失。

数据切分可以是物理 上的，对数据通过一系列的切分规则将数据分布到不同的DB服务器上，通过路由规则路由访问特定的数据库，这样一来每次访问面对的就不是单台服务器了，而是N台服务器，这样就可以降低单台机器的负载压力**（分库）**。数 据切分也可以是数据库内的 ，对数据通过一系列的切分规则，将数据分布到一个数据库的不同表中**（分表）**。

分库分表方式和规则：

- 设置分割区间（1-1000,1001-2000...）

  优点：可部分迁移

  缺点：数据分布不均

- hash取模（%4）

  优点：数据分布均匀

  缺点：数据迁移的时候麻烦，不能按照机器性能分摊数据

- 设置认证库保存数据映射关系

  优点：灵活性强，一对一关系

  缺点：每次查询之前都要多一次查询，性能大打折扣