自我介绍

毕业院校、时间、工作时间

工作经历：

负责的项目：

E家校：基础数据管理和数据展示的平台 项目基础结构

我叫xxx，专业是网络工程，20年毕业于广东技术师范大学。

毕业之后就入职现在所在的公司，从事Java开发

目前，负责的是一个叫E家校的项目，

这是一个基础数据管理和数据展示的平台，接收各种设备的数据推送，

通过nginx把一些子系统集成到了这个平台当中，是绿晶校园方案的核心

（有web端、移动端、微信端）后台面向学校，移动端、微信端面向教师和家长

这个项目通过nginx、docker、tomcat搭建集群的方式来部署。

项目使用的是一个基于servlet的自研框架来开发的，

后来为了跟进技术的发展和向微服务方面改造就把自研框架和springboot集成到了一起

使用了MySQL 数据库和redis数据库

前端页面是用了jsp、jq、ajax、h5、css

算是负责整个项目，

主要负责了项目：权限模块、支付模块、VIP模块、消息推送模块

日常的工作是 功能的开发、优化 系统的运维、更新部署

关键字、修饰符

权限修饰符

	public	protected	default（空）	private
同一类中	√	√	√	√
同一包中	√	√	√	
不同包的子类	√	√		
不同包中的无关类	√			

static

final

序列化

序列化是什么，反序列化是什么，

序列化 (Serialization)是将对象的状态信息转换为可以存储或传输的形式的过程。

序列化会存

类的信息、属性的类型信息、属性的数据

序列化会存什么

实现序列化接口的类的属性

不会存:

static修饰的属性、没实现实现序列化接口的父类的属性、被ransient关键字修饰的属性

初识transient关键字

其实这个关键字的作用很好理解，就是简单的一句话：将不需要序列化的属性前添加关键字transient，序列化对象的时候，这个属性就不会被序列化。

transient底层实现原理是什么?

原理：

transient的作用就是把这个字段的生命周期仅存于调用者的内存中而不会写到磁盘里持久化

java的serialization提供了一个非常棒的存储对象状态的机制，说白了serialization就是把对象的状态存储到硬盘上 去，等需要的时候就可以再把它读出来使用。有些时候像银行卡号这些字段是不希望在网络上传输的，transient的作用就是把这个字段的生命周期仅存于调用者的内存中而不会写到磁盘里持久化，意思是transient修饰的age字段，他的生命周期仅仅在内存中，不会被写到磁盘中。

集合

JVM

GC 分代收集算法 VS 分区收集算法

分代收集算法

当前主流 VM 垃圾收集都采用”分代收集”(Generational Collection)算法, 这种算法会根据对象存活周期的不同将内存划分为几块, 如 JVM 中的 新生代、老年代、永久代，这样就可以根据各年代特点分别采用最适当的GC算法

分区收集算法

分区算法则将整个堆空间划分为连续的不同小区间, 每个小区间独立使用, 独立回收. 这样做的好处是可以控制一次回收多少个小区间 , 根据目标停顿时间, 每次合理地回收若干个小区间(而不是整个堆), 从而减少一次 GC 所产生的停顿。

redis

redis是什么

Redis是C语言开发的一个开源的（遵从BSD协议）高性能键值对（key-value）的内存数据库，

可以用作数据库、缓存、消息中间件等。它是一种NoSQL（not-only sql，泛指非关系型数据库）的数据库。

1、性能优秀，数据在内存中，读写速度非常快，支持并发10W QPS；

2、单进程单线程，是线程安全的，采用IO多路复用机制；

3、丰富的数据类型，支持字符串（strings）、散列（hashes）、列表（lists）、集合（sets）、有序集合（sorted sets）等；

4、支持数据持久化。可以将内存中数据保存在磁盘中，重启时加载；

5、主从复制，哨兵，高可用；6、可以用作分布式锁；

7、可以作为消息中间件使用，支持发布订阅

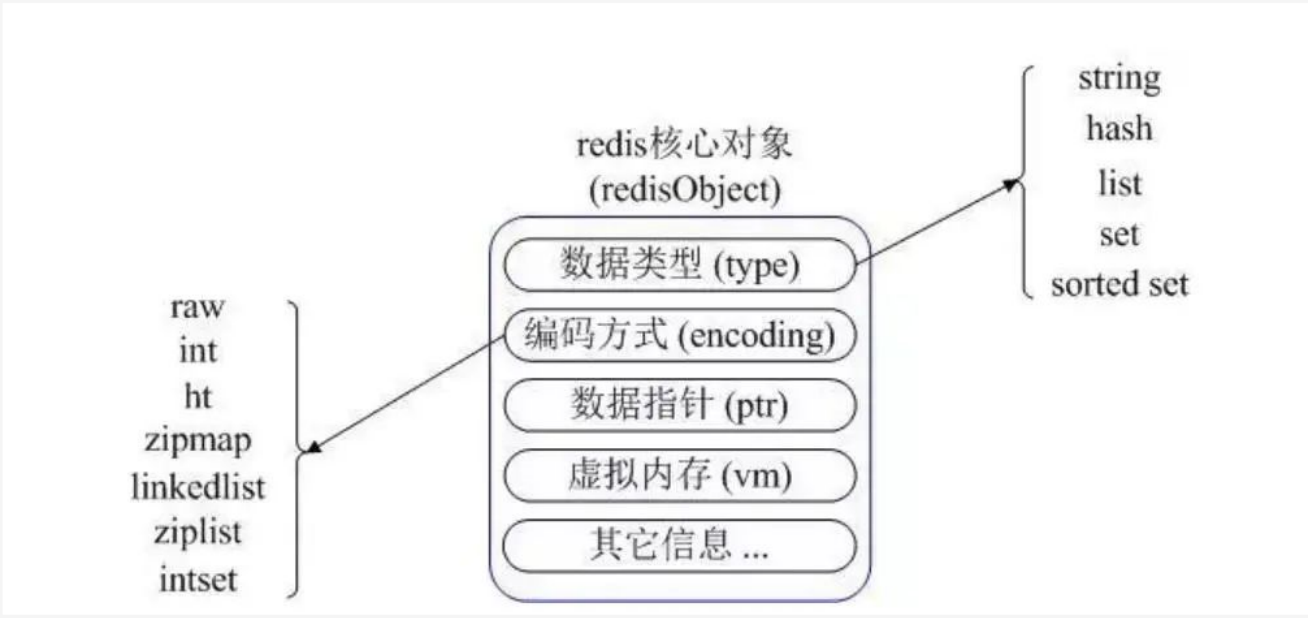
Redis的缺点
 是数据库容量受到物理内存的限制,不能用作海量数据的高性能读写,因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

五种数据类型

类型	简介	特性	场景
string（字符串）	二进制安全	可以包含任何数据，比如jpg图片或者序列化对象	---
Hash（字典）	键值对集合，即编程语言中的map类型	适合存储对象，并且可以像数据库中的update一个属性一样只修改某一项属性值	存储、读取、修改用户属性
List（列表）	链表（双向链表）	增删快，提供了操作某一元素的api	最新消息排行；消息队列
set（集合）	hash表实现，元素不重复	添加、删除、查找的复杂度都是O(1)，提供了求交集、并集、差集的操作	共同好友；利用唯一性，统计访问网站的所有Ip
sorted set（有序集合）	将set中的元素增加一个权重参数score，元素按score有序排列	数据插入集合时，已经进行了天然排序	排行榜；带权重的消息队列

- 1、string是redis最基本的类型，可以理解成与memcached一模一样的类型，一个key对应一个value。value不仅是string，也可以是数字。string类型是二进制安全的，意思是redis的string类型可以包含任何数据，比如jpg图片或者序列化的对象。string类型的值最大能存储512M。
- 2、Hash是一个键值（key-value）的集合。redis的hash是一个string的key和value的映射表，Hash特别适合存储对象。常用命令：hget,hset,hgetall等。
- 3、list列表是简单的字符串列表，按照插入顺序排序。可以添加一个元素到列表的头部（左边）或者尾部（右边） 常用命令：lpush、rpush、lpop、rpop、lrange(获取列表片段)等。应用场景：list应用场景非常多，也是Redis最重要的数据结构之一，比如twitter的关注列表，粉丝列表都可以用list结构来实现。数据结构：list就是链表，可以用来当消息队列用。redis提供了List的push和pop操作，还提供了操作某一段的api，可以直接查询或者删除某一段的元素。实现方式：redis list的是实现是一个双向链表，既可以支持反向查找和遍历，更方便操作，不过带来了额外的内存开销。
- 4、set是string类型的无序集合。集合是通过hashtable实现的。set中的元素是没有顺序的，而且是没有重复的。常用命令：sdd、spop、smembers、sunion等。应用场景：redis set对外提供的功能和list一样是一个列表，特殊之处在于set是自动去重的，而且set提供了判断某个成员是否在一个set集合中。
- 5、zset和set一样是string类型元素的集合，且不允许重复的元素。常用命令：zadd、zrange、zrem、zcard等。使用场景：sorted set可以通过用户额外提供一个优先级（score）的参数来为成员排序，并且是插入有序的，即自动排序。当你需要一个有序的并且不重复的集合列表，那么可以选择sorted set结构。和set相比，sorted set关联了一个double类型权重的参数score，使得集合中的元素能够按照score进行有序排列，redis正是通过分数来为集合中的成员进行从小到大的排序。实现方式：Redis sorted set的内部使用HashMap和跳跃表(skipList)来保证数据的存储和有序，HashMap里放的是成员到score的映射，而跳跃表里存放的是所有的成员，排序依据是HashMap里存的score，使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

Redis内部内存管理是如何描述这5种数据类型的



Redis为何这么快，Redis 为什么是单线程的

(一)纯内存操作
(二)单线程操作，避免了频繁的上下文切换
(三)采用了非阻塞I/O多路复用机制

官方FAQ表示，因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）Redis利用队列技术将并发访问变为串行访问
1）绝大部分请求是纯粹的内存操作（非常快速）2）采用单线程,避免了不必要的上下文切换和竞争条件
3）非阻塞IO优点：
速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)
支持丰富数据类型，支持string，list，set，sorted set，hash
支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行
丰富的特性：可用于缓存，消息，按key设置过期时间，过期后将会自动删除如何解决redis的并发竞争key问题
同时有多个子系统去set一个key。这个时候要注意什么呢？ 不推荐使用redis的事务机制。因为我们的生产环境，基本都是redis集群环境，做了数据分片操作。你一个事务中有涉及到多个key操作的时候，这多个key不一定都存储在同一个redis-server上。因此，redis的事务机制，十分鸡肋。
(1)如果对这个key操作，不要求顺序： 准备一个分布式锁，大家去抢锁，抢到锁就做set操作即可
(2)如果对这个key操作，要求顺序： 分布式锁+时间戳。 假设这会系统B先抢到锁，将key1设置为{valueB 3:05}。接下来系统A抢到锁，发现自己的valueA的时间戳早于缓存中的时间戳，那就不做set操作了。以此类推。
(3) 利用队列，将set方法变成串行访问也可以redis遇到高并发，如果保证读写key的一致性
对redis的操作都是具有原子性的,是线程安全的操作,你不用考虑并发问题,redis内部已经帮你处理好并发的问题了。

策略	描述
volatile-lru	从已设置过期时间的KV集中优先对最近最少使用(less recently used)的数据淘汰
volitile-ttl	从已设置过期时间的KV集中优先对剩余时间短(time to live)的数据淘汰
volitile-random	从已设置过期时间的KV集中随机选择数据淘汰
allkeys-lru	从所有KV集中优先对最近最少使用(less recently used)的数据淘汰
allKeys-random	从所有KV集中随机选择数据淘汰
noeviction	不淘汰策略，若超过最大内存，返回错误信息

集群

集群方案

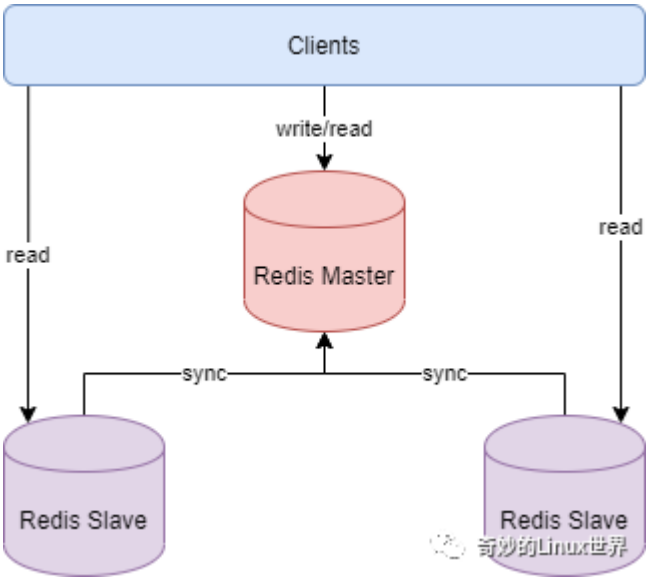
Redis支持三种集群方案

- 主从复制模式
- Sentinel（哨兵）模式
- Cluster模式

主从复制模式

1. 基本原理

主从复制模式中包含一个主数据库实例（master）与一个或多个从数据库实例（slave），如下图



客户端可对主数据库进行读写操作，对从数据库进行读操作，主数据库写入的数据会实时自动同步给从数据库。

具体工作机制为：

1. slave启动后，向master发送SYNC命令，master接收到SYNC命令后通过bgsave保存快照（即上文所介绍的RDB持久化），并使用缓冲区记录保存快照这段时间内执行的写命令
2. master将保存的快照文件发送给slave，并继续记录执行的写命令
3. slave接收到快照文件后，加载快照文件，载入数据
4. master快照发送完后开始向slave发送缓冲区的写命令，slave接收命令并执行，完成复制初始化
5. 此后master每次执行一个写命令都会同步发送给slave，保持master与slave之间数据的一致性

2. 部署示例

本示例基于Redis 5.0.3版。

redis.conf的主要配置

```
###网络相关###  # bind 127.0.0.1 # 绑定监听的网卡IP，注释掉或配置成0.0.0.0可使任意IP均可访问  protected-mode no # 关闭保护模式，使用密码访问  port 6379  # 设置监听端口，建议生产环境均使用自定义端口  timeout 30 # 客户端连接空闲多久后断开连接，单位秒，0表示禁用  ###通用配置###  daemonize yes # 在后台运行  pidfile /var/run/redis_6379.pid  # pid进程文件名  logfile /usr/local/redis/logs/redis.log # 日志文件的位置  ###RDB持久化配置### save 900 1 # 900s内至少一次写操作则执行bgsave进行RDB持久化  save 300 10  save 60 10000  # 如果禁用RDB持久化，可在这里添加 save ""  rdbcompression yes #是否对RDB文件进行压缩，建议设置为no，以（磁盘）空间换（CPU）时间  dbfilename dump.rdb # RDB文件名称  dir /usr/local/redis/datas # RDB文件保存路径，AOF文件也保存在这里  ###AOF配置### appendonly yes # 默认值是no，表示不使用AOF增量持久化的方式，使用RDB全量持久化的方式 appendfsync everysec # 可选值 always， everysec，no，建议设置为everysec  ###设置密码### requirepass 123456 # 设置复杂一点的密码
```

部署主从复制模式只需稍微调整slave的配置，在redis.conf中添加

```
replicaof 127.0.0.1 6379 # master的ip，port  masterauth 123456 # master的密码 replica-serve-stale-data no # 如果slave无法与master同步，设置成slave不可读，方便监控脚本发现问题
```

本示例在单台服务器上配置master端口6379，两个slave端口分别为7001,7002，启动master，再启动两个slave

```
[root@dev-server-1 master-slave]# redis-server master.conf  [root@dev-server-1
master-slave]# redis-server slave1.conf  [root@dev-server-1 master-slave]# redis-
server slave2.conf
```

进入master数据库，写入一个数据，再进入一个slave数据库，立即便可访问刚才写入master数据库的数据。如下所示

```
[root@dev-server-1 master-slave]# redis-cli   127.0.0.1:6379> auth 123456  OK
127.0.0.1:6379> set site blog.jboost.cn  OK  127.0.0.1:6379> get site
"blog.jboost.cn"  127.0.0.1:6379> info replication  # Replication  role:master
connected_slaves:2  slave0:ip=127.0.0.1,port=7001,state=online,offset=13364738,lag=1
slave1:ip=127.0.0.1,port=7002,state=online,offset=13364738,lag=0 ...
127.0.0.1:6379> exit  [root@dev-server-1 master-slave]# redis-cli -p 7001
127.0.0.1:7001> auth 123456  OK  127.0.0.1:7001> get site  "blog.jboost.cn"
```

执行info replication命令可以查看连接该数据库的其它库的信息，如上可看到有两个slave连接到master

3. 主从复制的优缺点

优点：

1. master能自动将数据同步到slave，可以进行读写分离，分担master的读压力
2. master、slave之间的同步是以非阻塞的方式进行的，同步期间，客户端仍然可以提交查询或更新请求

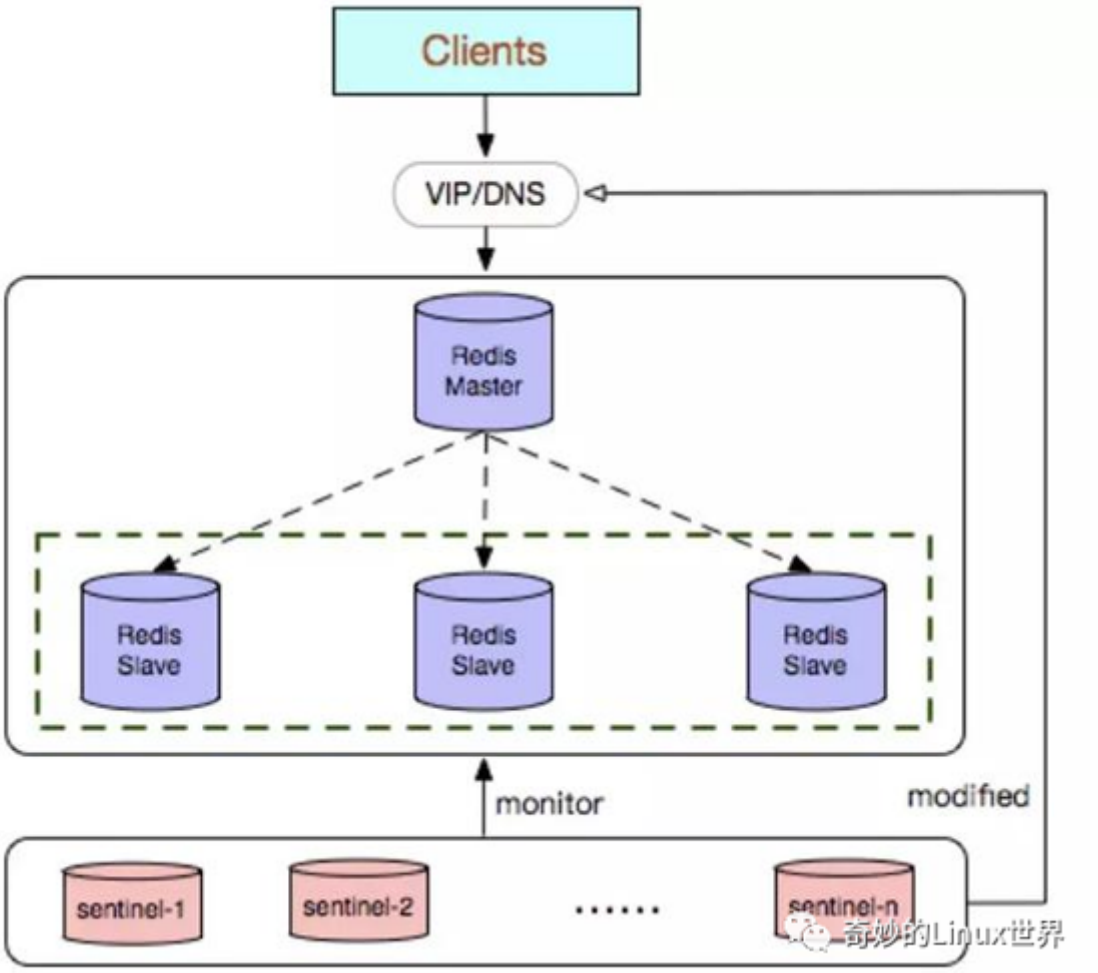
缺点：

1. 不具备自动容错与恢复功能，master或slave的宕机都可能导致客户端请求失败，需要等待机器重启或手动切换客户端IP才能恢复
2. master宕机，如果宕机前数据没有同步完，则切换IP后会存在数据不一致的问题
3. 难以支持在线扩容，Redis的容量受限于单机配置

Sentinel（哨兵）模式

1. 基本原理

哨兵模式基于主从复制模式，只是引入了哨兵来监控与自动处理故障。如图



哨兵顾名思义，就是来为Redis集群站哨的，一旦发现问题能做出相应的应对处理。其功能包括

1. 监控master、slave是否正常运行
2. 当master出现故障时，能自动将一个slave转换为master（大哥挂了，选一个小弟上位）
3. 多个哨兵可以监控同一个Redis，哨兵之间也会自动监控

哨兵模式的具体工作机制：

在配置文件中通过 sentinel monitor来定位master的IP、端口，一个哨兵可以监控多个master数据库，只需要提供多个该配置项即可。哨兵启动后，会与要监控的master建立两条连接：

1. 一条连接用来订阅master的*sentinel*:hello频道与获取其他监控该master的哨兵节点信息
2. 另一条连接定期向master发送INFO等命令获取master本身的信息

与master建立连接后，哨兵会执行三个操作：

1. 定期（一般10s一次，当master被标记为主观下线时，改为1s一次）向master和slave发送INFO命令
2. 定期向master和slave的*sentinel*:hello频道发送自己的信息
3. 定期（1s一次）向master、slave和其他哨兵发送PING命令

发送INFO命令可以获取当前数据库的相关信息从而实现新节点的自动发现。所以说哨兵只需要配置master数据库信息就可以自动发现其slave信息。获取到slave信息后，哨兵也会与slave建立两条连接执行监控。通过INFO命令，哨兵可以获取主从数据库的最新信息，并进行相应的操作，比如角色变更等。

接下来哨兵向主从数据库的sentinel:hello频道发送信息与同样监控这些数据库的哨兵共享自己的信息，发送内容为哨兵的ip端口、运行id、配置版本、master名字、master的ip端口还有master的配置版本。这些信息有以下用处：

1. 其他哨兵可以通过该信息判断发送者是否是新发现的哨兵，如果是的话会创建一个到该哨兵的连接用于发送PING命令。
2. 其他哨兵通过该信息可以判断master的版本，如果该版本高于直接记录的版本，将会更新
3. 当实现了自动发现slave和其他哨兵节点后，哨兵就可以通过定期发送PING命令定时监控这些数据库和节点有没有停止服务。

如果被PING的数据库或者节点超时（通过 sentinel down-after-milliseconds master-name milliseconds 配置）未回复，哨兵认为其主观下线（sdown，s就是Subjectively —— 主观地）。如果下线的是master，哨兵会向其它哨兵发送命令询问它们是否也认为该master主观下线，如果达到一定数目（即配置文件中的quorum）投票，哨兵会认为该master已经客观下线（odown，o就是Objectively —— 客观地），并选举领头的哨兵节点对主从系统发起故障恢复。若没有足够的sentinel进程同意master下线，master的客观下线状态会被移除，若master重新向sentinel进程发送的PING命令返回有效回复，master的主观下线状态就会被移除

哨兵认为master客观下线后，故障恢复的操作需要由选举的领头哨兵来执行，选举采用Raft算法：

1. 发现master下线的哨兵节点（我们称他为A）向每个哨兵发送命令，要求对方选自己为领头哨兵
2. 如果目标哨兵节点没有选过其他人，则会同意选举A为领头哨兵
3. 如果有超过一半的哨兵同意选举A为领头，则A当选
4. 如果有多个哨兵节点同时参选领头，此时有可能存在一轮投票无竞选者胜出，此时每个参选的节点等待一个随机时间后再次发起参选请求，进行下一轮投票竞选，直至选举出领头哨兵

选出领头哨兵后，领头者开始对系统进行故障恢复，从出现故障的master的从数据库中挑选一个来当选新的master,选择规则如下：

1. 所有在线的slave中选择优先级最高的，优先级可以通过slave-priority配置
2. 如果有多个最高优先级的slave，则选取复制偏移量最大（即复制越完整）的当选
3. 如果以上条件都一样，选取id最小的slave

挑选出需要继任的slave后，领头哨兵向该数据库发送命令使其升格为master，然后再向其他slave发送命令接受新的master，最后更新数据。将已经停止的旧的master更新为新的master的从数据库，使其恢复服务后以slave的身份继续运行。

2. 部署演示

本示例基于Redis 5.0.3版。

哨兵模式基于前文的主从复制模式。哨兵的配置文件为sentinel.conf，在文件中添加

```
sentinel monitor mymaster 127.0.0.1 6379 1 # mymaster定义一个master数据库的名称，后面是master的ip， port，1表示至少需要一个Sentinel进程同意才能将master判断为失效，如果不满足这个条件，则自动故障转移（failover）不会执行 sentinel auth-pass mymaster 123456 # master的密码 sentinel down-after-milliseconds mymaster 5000 # 5s未回复PING，则认为master主观下线，默认为30s sentinel parallel-syncs mymaster 2  # 指定在执行故障转移时，最多可以有多少个slave实例在同步新的master实例，在slave实例较多的情况下这个数字越小，同步的时间越长，完成故障转移所需的时间就越长 sentinel failover-timeout mymaster 300000 # 如果在该时间（ms）内未能完成故障转移操作，则认为故障转移失败，生产环境需要根据数据量设置该值
```

一个哨兵可以监控多个master数据库，只需按上述配置添加多套

分别以26379,36379,46379端口启动三个sentinel

```
[root@dev-server-1 sentinel]# redis-server sentinel1.conf --sentinel  [root@dev-server-1 sentinel]# redis-server sentinel2.conf --sentinel [root@dev-server-1 sentinel]# redis-server sentinel3.conf --sentinel
```

也可以使用redis-sentinel sentinel1.conf 命令启动。此时集群包含一个master、两个slave、三个sentinel，如图，

我们来模拟master挂掉的场景，执行 kill -9 3017 将master进程干掉，进入slave中执行 info replication查看，

```
[root@dev-server-1 sentinel]# redis-cli -p 7001  127.0.0.1:7001> auth 123456  OK
127.0.0.1:7001> info replication  # Replication  role:slave  master_host:127.0.0.1
master_port:7002  master_link_status:up  master_last_io_seconds_ago:1
master_sync_in_progress:0  # 省略  127.0.0.1:7001> exit  [root@dev-server-1
sentinel]# redis-cli -p 7002  127.0.0.1:7002> auth 123456  OK  127.0.0.1:7002> info
replication  # Replication  role:master  connected_slaves:1
slave0:ip=127.0.0.1,port=7001,state=online,offset=13642721,lag=1  # 省略
```

可以看到slave 7002已经成功上位晋升为master（role：master），接收一个slave 7001的连接。此时查看slave2.conf配置文件，发现replicaof的配置已经被移除了，slave1.conf的配置文件里replicaof 127.0.0.1 6379 被改为 replicaof 127.0.0.1 7002。重新启动master，也可以看到master.conf配置文件中添加了replicaof 127.0.0.1 7002的配置项，可见大哥（master）下位后，再出来混就只能当当小弟（slave）了，三十年河东三十年河西。

3. 哨兵模式的优缺点

优点：

1. 哨兵模式基于主从复制模式，所以主从复制模式有的优点，哨兵模式也有
2. 哨兵模式下，master挂掉可以自动进行切换，系统可用性更高

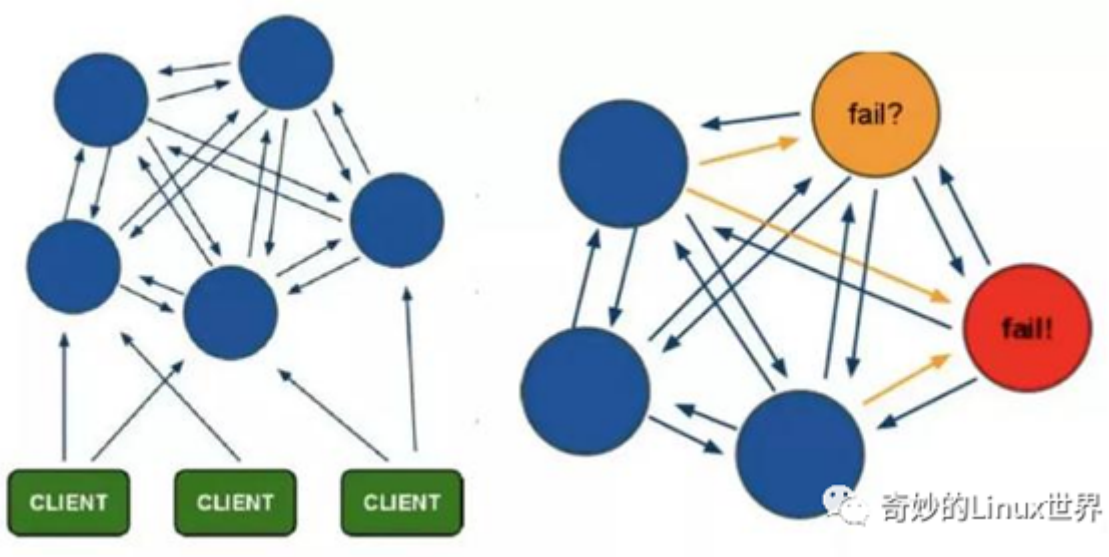
缺点：

1. 同样也继承了主从模式难以在线扩容的缺点，Redis的容量受限于单机配置
2. 需要额外的资源来启动sentinel进程，实现相对复杂一点，同时slave节点作为备份节点不提供服务

Cluster模式

1. 基本原理

哨兵模式解决了主从复制不能自动故障转移，达不到高可用的问题，但还是存在难以在线扩容，Redis容量受限于单机配置的问题。Cluster模式实现了Redis的分布式存储，即每台节点存储不同的内容，来解决在线扩容的问题。如图



Cluster采用无中心结构,它的特点如下:

1. 所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽
2. 节点的fail是通过集群中超过半数的节点检测失效时才生效
3. 客户端与redis节点直连,不需要中间代理层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可

Cluster模式的具体工作机制:

\1. 在Redis的每个节点上，都有一个插槽（slot），取值范围为0-16383

\2. 当我们存取key的时候，Redis会根据CRC16的算法得出一个结果，然后把结果对16384求余数，这样每个key都会对应一个编号在0-16383之间的哈希槽，通过这个值，去找到对应的插槽所对应的节点，然后直接自动跳转到这个对应的节点上进行存取操作

\3. 为了保证高可用，Cluster模式也引入主从复制模式，一个主节点对应一个或者多个从节点，当主节点宕机的时候，就会启用从节点

\4. 当其它主节点ping一个主节点A时，如果半数以上的主节点与A通信超时，那么认为主节点A宕机了。如果主节点A和它的从节点都宕机了，那么该集群就无法再提供服务了

Cluster模式集群节点最小配置6个节点(3主3从，因为需要半数以上)，其中主节点提供读写操作，从节点作为备用节点，不提供请求，只作为故障转移使用。

2. 部署演示

本示例基于Redis 5.0.3版。

Cluster模式的部署比较简单，首先在redis.conf中

```
port 7100 # 本示例6个节点端口分别为7100,7200,7300,7400,7500,7600   daemonize yes # r后台运行   pidfile /var/run/redis_7100.pid # pidfile文件对应7100,7200,7300,7400,7500,7600   cluster-enabled yes # 开启集群模式   masterauth passw0rd # 如果设置了密码，需要指定master密码  cluster-config-file nodes_7100.conf # 集群的配置文件，同样对应7100,7200等六个节点  cluster-node-timeout 15000 # 请求超时 默认15秒，可自行设置
```

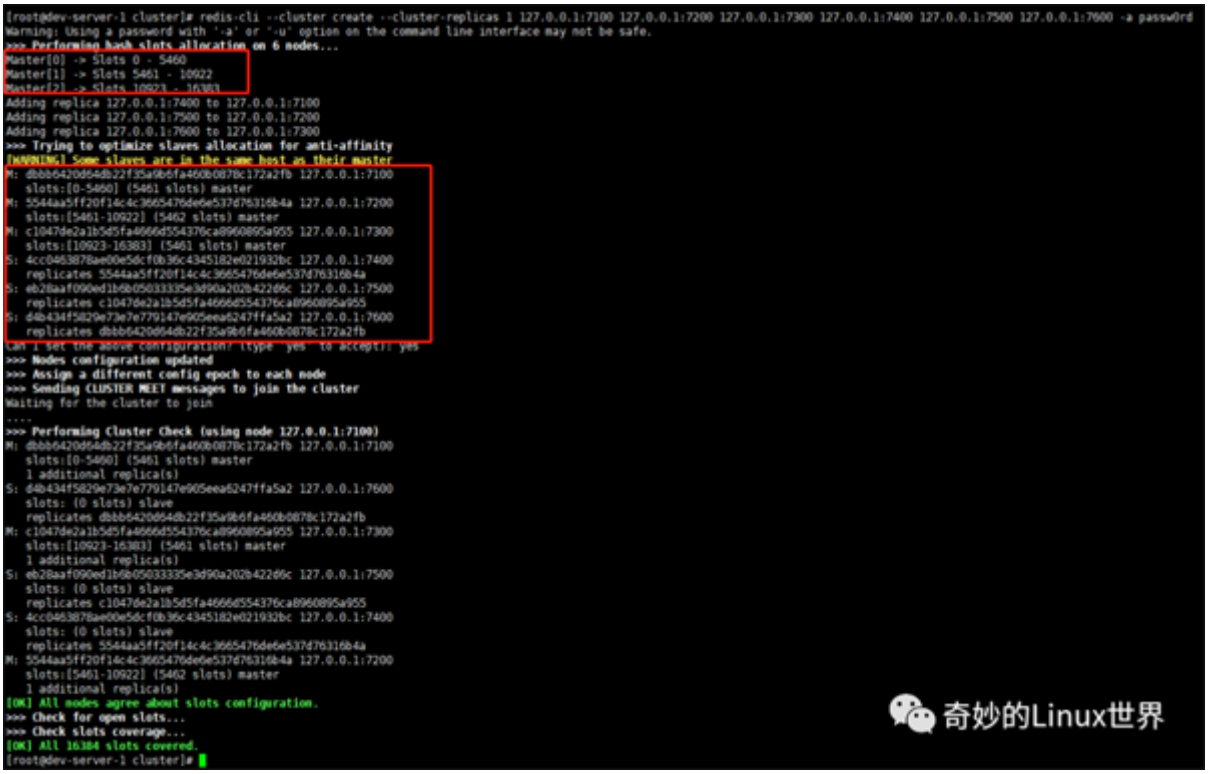
分别以端口7100,7200,7300,7400,7500,7600 启动六个实例(如果是每个服务器一个实例则配置可一样)

```
[root@dev-server-1 cluster]# redis-server redis_7100.conf  [root@dev-server-1 cluster]# redis-server redis_7200.conf  ...
```

然后通过命令将这个6个实例组成一个3主节点3从节点的集群，

```
redis-cli --cluster create --cluster-replicas 1 127.0.0.1:7100 127.0.0.1:7200
127.0.0.1:7300 127.0.0.1:7400 127.0.0.1:7500 127.0.0.1:7600 -a passw0rd
```

执行结果如图



可以看到 7100， 7200， 7300 作为3个主节点，分配的slot分别为 0-5460， 5461-10922， 10923-16383， 7600作为7100的slave， 7500作为7300的slave，7400作为7200的slave。

我们连接7100设置一个值

```
[root@dev-server-1 cluster]# redis-cli -p 7100 -c -a passw0rd  Warning: Using a
password with '-a' or '-u' option on the command line interface may not be safe.
127.0.0.1:7100> set site blog.jboost.cn  -> Redirected to slot [9421] located at
127.0.0.1:7200  OK  127.0.0.1:7200> get site  "blog.jboost.cn"  127.0.0.1:7200>
```

注意添加 -c 参数表示以集群模式，否则报 (error) MOVED 9421 127.0.0.1:7200 错误， 以 -a 参数指定密码，否则报(error) NOAUTH Authentication required错误。

从上面命令看到key为site算出的slot为9421，落在7200节点上，所以有Redirected to slot [9421] located at 127.0.0.1:7200，集群会自动进行跳转。因此客户端可以连接任何一个节点来进行数据的存取。

通过cluster nodes可查看集群的节点信息

```
127.0.0.1:7200> cluster nodes  eb28aaf090ed1b6b05033335e3d90a202b422d6c
127.0.0.1:7500@17500 slave c1047de2a1b5d5fa4666d554376ca8960895a955 0 1584165266071
5 connected  4cc0463878ae00e5dcf0b36c4345182e021932bc 127.0.0.1:7400@17400 slave
5544aa5ff20f14c4c3665476de6e537d76316b4a 0 1584165267074 4 connected
dbbb6420d64db22f35a9b6fa460b0878c172a2fb 127.0.0.1:7100@17100 master - 0
1584165266000 1 connected 0-5460  d4b434f5829e73e7e779147e905eea6247ffa5a2
127.0.0.1:7600@17600 slave dbbb6420d64db22f35a9b6fa460b0878c172a2fb 0 1584165265000
6 connected  5544aa5ff20f14c4c3665476de6e537d76316b4a 127.0.0.1:7200@17200
myself,master - 0 1584165267000 2 connected 5461-10922
c1047de2a1b5d5fa4666d554376ca8960895a955 127.0.0.1:7300@17300 master - 0
1584165268076 3 connected 10923-16383
```

我们将7200通过 kill -9 pid杀死进程来验证集群的高可用，重新进入集群执行cluster nodes可以看到7200 fail了，但是7400成了master，重新启动7200，可以看到此时7200已经变成了slave。

3. Cluster模式的优缺点

优点：

\1. 无中心架构，数据按照slot分布在多个节点。

\2. 集群中的每个节点都是平等的关系，每个节点都保存各自的数据和整个集群的状态。每个节点都和其他所有节点连接，而且这些连接保持活跃，这样就保证了我们只需要连接集群中的任意一个节点，就可以获取到其他节点的数据。

\3. 可线性扩展到1000多个节点，节点可动态添加或删除

\4. 能够实现自动故障转移，节点之间通过gossip协议交换状态信息，用投票机制完成slave到master的角色转换

缺点：

1. 客户端实现复杂，驱动要求实现Smart Client，缓存slots mapping信息并及时更新，提高了开发难度。目前仅JedisCluster相对成熟，异常处理还不完善，比如常见的“max redirect exception”
2. 节点会因为某些原因发生阻塞（阻塞时间大于 cluster-node-timeout）被判断下线，这种failover是没有必要的
3. 数据通过异步复制，不保证数据的强一致性
4. slave充当“冷备”，不能缓解读压力
5. 批量操作限制，目前只支持具有相同slot值的key执行批量操作，对mset、mget、sunion等操作支持不友好
6. key事务操作支持有线，只支持多key在同一节点的事务操作，多key分布不同节点时无法使用事务功能
7. 不支持多数据库空间，单机redis可以支持16个db，集群模式下只能使用一个，即db 0

Redis Cluster模式不建议使用pipeline和multi-keys操作，减少max redirect产生的场景

nginx

什么是nginx

nginx是一个 轻量级/高性能的反向代理Web服务器，

跨平台、配置简单、方向代理、高并发连接

优点：

占内存小，可实现高并发连接，处理响应快
可实现http服务器、虚拟主机、方向代理、负载均衡
Nginx配置简单
可以不暴露正式的服务器IP地址
缺点：
动态处理差：nginx处理静态文件好,耗费内存少，但是处理动态页面则很鸡肋，现在一般前端用nginx作为反向代理抗住压力，

location的作用是什么?

location指令的作用是根据用户请求的URI来执行不同的应用，也就是根据用户请求的网站URL进行匹配，匹配成功即进行相关的操作。

怎么搭建负载均衡

负载均衡策略

轮询 权重 weight ip_hash(IP绑定)

负载均衡配置

maven

依赖冲突的解决

先 最短路径 ， 路径长度相同，先声明有效，后声明无效

1、路径优先原则

直接依赖优于传递依赖。如果传递依赖的Jar包版本冲突了，那么可以单独声明一个指定版本的依赖Jar包，即可解决冲突。例如，b引用了a的0.0.1版本，c引用了b的0.0.2版本，如果我们想用的版本是0.0.3版本，可以直接单独声明一个a的0.0.3版本。

2、第一声明原则

在pom.xml配置文件中，如果有两个名称相同，版本的不同依赖声明，先写的会生效，所以先声明自己要用的版本。这里的名称相同，版本不同的依赖声明，既可以是直接依赖，也可以是传递依赖。

例如，b引用了a的0.0.1版本，c引用了b的0.0.2版本，如果我们想用的版本是0.0.1版本，那么可以将b的版本依赖放在c的前面。

3、排除原则

在发生传递依赖冲突时，如果依赖不是项目需要的，可以在对应的传递依赖声明中进行排除。例如，b引用了a的0.0.1版本，c引用了b的0.0.2版本，如果我们想用的版本是0.0.2版本，那么可以将b的版本依赖中排除a。
使用<dependency&gt的元素将会引起冲突的元素排除。

网络

GET 和 POST 两种基本请求方法的区别

- GET产生一个TCP数据包；POST产生两个TCP数据包。

- GET在浏览器回退时是无害的，而POST会再次提交请求。
- GET产生的URL地址可以被Bookmark，而POST不可以。
- GET请求会被浏览器主动cache，而POST不会，除非手动设置。
- GET请求只能进行url编码，而POST支持多种编码方式。
- GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
- GET请求在URL中传送的参数是有长度限制的，而POST么有。
- 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。
- GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。
- GET参数通过URL传递，POST放在Request body中。

三次握手

所谓三次握手（Three-Way Handshake）即建立TCP连接，就是指建立一个TCP连接时，需要客户端和服务端总共发送3个包以确认连接的建立。在socket编程中，这一过程由客户端执行connect来触发，整个流程如下图所示：

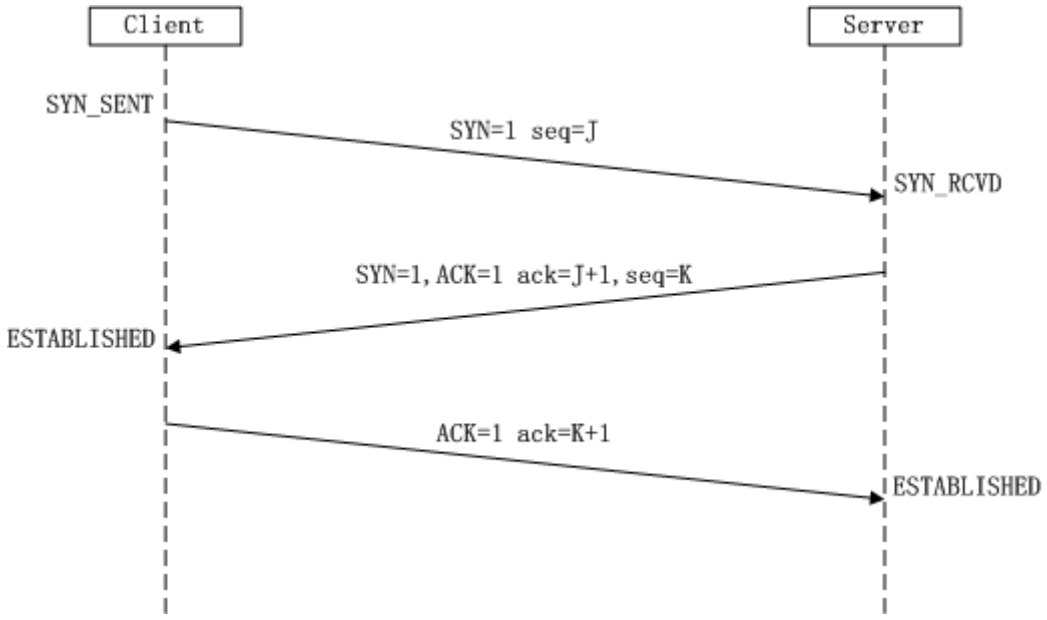


图2 TCP三次握手

（1）第一次握手：Client将标志位SYN置为1，随机产生一个值seq=J，并将该数据包发送给Server，Client进入SYN_SENT状态，等待Server确认。

（2）第二次握手：Server收到数据包后由标志位SYN=1知道Client请求建立连接，Server将标志位SYN和ACK都置为1，ack=J+1，随机产生一个值seq=K，并将该数据包发送给Client以确认连接请求，Server进入SYN_RCVD状态。

（3）第三次握手：Client收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给Server，Server检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手，随后Client与Server之间可以开始传输数据了。

四次挥手

三次握手耳熟能详，四次挥手估计就🤔，所谓四次挥手（Four-Way Wavehand）即终止TCP连接，就是指断开一个TCP连接时，需要客户端和服务端总共发送4个包以确认连接的断开。在socket编程中，这一过程由客户端或服务端任一方执行close来触发，整个流程如下图所示：

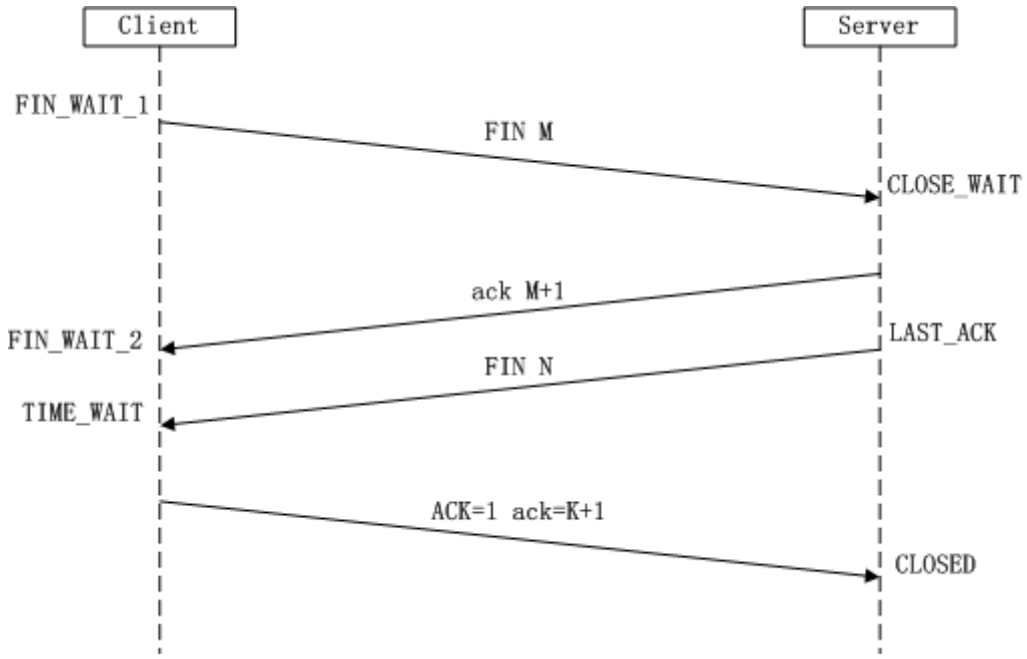


图3 TCP四次挥手

由于TCP连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个TCP连接上仍然能够发送数据，直到这一方向也发送了FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭，上图描述的即是如此。

（1）第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN_WAIT_1状态。

（2）第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE_WAIT状态。

（3）第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST_ACK状态。

（4）第四次挥手：Client收到FIN后，Client进入TIME_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。

为什么建立连接是三次握手，而关闭连接却是四次挥手呢?

这是因为服务端在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。而关闭连接时，当收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，己方也未必全部数据都发送给对方了，所以己方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送。

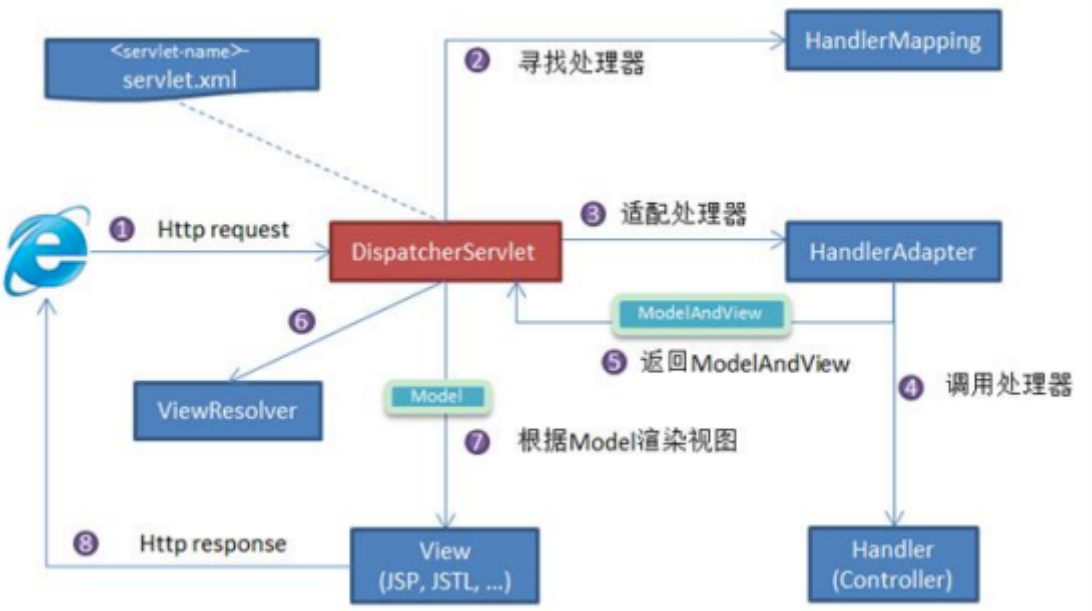
输入url到加载完页面发生了什么

1. DNS解析
2. TCP连接
3. 发送HTTP请求
4. 服务器处理请求并返回HTTP报文
5. 浏览器解析渲染页面
6. 连接结束

pringMVC是什么：

springMVC是一个MVC的开源框架，springMVC=struts2+spring，springMVC就相当于是Struts2加上sring的整合，但是这里有一个疑惑就是，springMVC和spring是什么样的关系呢？这个在百度百科上有一个很好的解释：意思是说，springMVC是spring的一个后续产品，其实就是spring在原有基础上，又提供了web应用的MVC模块，可以简单的把springMVC理解为是spring的一个模块（类似AOP，IOC这样的模块），网络上经常会说springMVC和spring无缝集成，其实springMVC就是spring的一个子模块，所以根本不需要同spring进行整合。

工作流程



1、向服务器发送 HTTP 请求，请求被前端控制器 DispatcherServlet 捕获。
2、 DispatcherServlet 根据 -servlet.xml 中的配置对请求的 URL 进行解析，得到请求资源标识符（URI）。然后根据该 URI，调用 HandlerMapping 获得该 Handler 配置的所有相关的对象（包括 Handler 对象以及 Handler 对象对应的拦截器），最后以 HandlerExecutionChain 对象的形式返回。
3、 DispatcherServlet 根据获得的 Handler，选择一个合适的 HandlerAdapter。（附注：如果成功获得 HandlerAdapter 后，此时将开始

执行拦截器的 preHandler(...)方法）。
4、提取 Request 中的模型数据，填充 Handler 入参，开始执行 Handler（ Controller)。在填充 Handler 的入参过程中，根据你的配置，Spring 将帮你做一些额外的工作：
HttpMessageConveter：将请求消息（如 Json、xml 等数据）转换成一个对象，将对象转换为指定的响应信息。
 数据转换：对请求消息进行数据转换。如 String 转换成 Integer、Double 等。
 数据根式化：对请求消息进行数据格式化。如将字符串转换成格式化数字或格式化日期等。
 数据验证：验证数据的有效性（长度、格式等），验证结果存储到 BindingResult 或 Error 中。
5、Handler(Controller)执行完成后，向 DispatcherServlet 返回一个 ModelAndView 对象；
6、根据返回的 ModelAndView，选择一个适合的 ViewResolver（必须是已经注册到 Spring 容器中的 ViewResolver)返回给 DispatcherServlet。
7、 ViewResolver 结合 Model 和 View，来渲染视图。
8、视图负责将渲染结果返回给客户端。

mvc流程

1、 用户发送请求至前端控制器DispatcherServlet。
2、 DispatcherServlet收到请求调用HandlerMapping处理器映射器。
3、 处理器映射器找到具体的处理器(可以根据xml配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
4、 DispatcherServlet调用HandlerAdapter处理器适配器。
5、 HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。
6、 Controller执行完成返回ModelAndView。
7、 HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet。
8、 DispatcherServlet将ModelAndView传给ViewReslover视图解析器。
9、 ViewReslover解析后返回具体View。
10、DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。
11、 DispatcherServlet响应用户。
组件说明：
以下组件通常使用框架提供实现：
DispatcherServlet：作为前端控制器，整个流程控制的中心，控制其它组件执行，统一调度，降低组件
之间的耦合性，提高每个组件的扩展性。
HandlerMapping：通过扩展处理器映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。
HandlAdapter：通过扩展处理器适配器，支持更多类型的处理器。
ViewResolver：通过扩展视图解析器，支持更多类型的视图解析，例如：jsp、freemarker、pdf、excel等。
springMVC中的组件：
1、前端控制器DispatcherServlet（不需要工程师开发）,由框架提供
作用：接收请求，响应结果，相当于转发器，中央处理器。有了dispatcherServlet减少了其它组件之间
的耦合度。
用户请求到达前端控制器，它就相当于mvc模式中的c，dispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet的存在降低了组件之间的耦合性。
2、处理器映射器HandlerMapping(不需要工程师开发),由框架提供
作用：根据请求的url查找Handler
HandlerMapping负责根据用户请求找到Handler即处理器，springmvc提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。
3、处理器适配器HandlerAdapter
作用：按照特定规则（HandlerAdapter要求的规则）去执行Handler
通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。
4、处理器Handler(需要工程师开发)
注意：编写Handler时按照HandlerAdapter的要求去做，这样适配器才可以去正确执行Handler

Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具
体的用户请求进行处理。
由于Handler涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发Handler。
5、视图解析器View resolver(不需要工程师开发),由框架提供
作用：进行视图解析，根据逻辑视图名解析成真正的视图（view）
View Resolver负责将处理结果生成View视图，View Resolver首先根据逻辑视图名解析成物理视图名即
具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。
springmvc框架提供了很多的View视图类型，包括：jstlView、freemarkerView、pdfView等。
一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。
6、视图View(需要工程师开发jsp...)
View是一个接口，实现类支持不同的View类型（jsp、freemarker、pdf...）