技术文档

# Linux

## install

### rpm安装方式

#### wget安装

下载rpm包

<http://mirrors.163.com/centos/6.2/os/x86_64/Packages/wget-1.12-1.4.el6.x86_64.rpm>

rpm安装

rpm ivh wget-1.12-1.4.el6.x86\_64.rpm

### yum安装方式

-y, --assumeyes answer yes for all questions

#### wget安装

yum -y install wget

#### gcc

yum -y install gcc

#### gcc-c++

yum -y install gcc-c++

#### tcl

yum -y install tcl

## Shell

### 系统信息

arch ##显示机器的[处理器架构](https://www.baidu.com/s?wd=%E5%A4%84%E7%90%86%E5%99%A8%E6%9E%B6%E6%9E%84&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9nHN9nvc1nHbkPhf3rH0s0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErHD4n1fdn1R4Pj6vnWc3n1n4)

uname -m ##显示机器的[处理器架构](https://www.baidu.com/s?wd=%E5%A4%84%E7%90%86%E5%99%A8%E6%9E%B6%E6%9E%84&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9nHN9nvc1nHbkPhf3rH0s0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErHD4n1fdn1R4Pj6vnWc3n1n4)

uname -r ##显示正在使用的[内核版本](https://www.baidu.com/s?wd=%E5%86%85%E6%A0%B8%E7%89%88%E6%9C%AC&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9nHN9nvc1nHbkPhf3rH0s0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErHD4n1fdn1R4Pj6vnWc3n1n4)

cat /proc/cpuinfo ##显示CPU info的信息

cat /proc/cpuinfo | grep -E 'processor|model name'

[grep](#_grep)正则表达式获取processor和model name信息

cat /proc/interrupts ##显示中断

cat /proc/meminfo ##校验内存使用

cat /proc/swaps ##显示哪些swap被使用

cat /proc/version ##显示内核的版本

cat /proc/net/dev ##显示[网络适配器](https://www.baidu.com/s?wd=%E7%BD%91%E7%BB%9C%E9%80%82%E9%85%8D%E5%99%A8&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9nHN9nvc1nHbkPhf3rH0s0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErHD4n1fdn1R4Pj6vnWc3n1n4)及统计

cat /proc/mounts ##显示已加载的文件系统

### 关机

halt ##立刻关机

poweroff ##立刻关机

shutdown -h now ##立刻关机（root用户使用）

shutdown -h 10 ##10分钟后自动关机

### 用户管理

#### 创建用户

useradd 用户名

-c comment 指定一段注释性描述。

-d 目录 指定用户主目录，如果此目录不存在，则同时使用-m选项，可以创建主目录。

-g 用户组 指定用户所属的用户组。

-G 用户组，用户组 指定用户所属的附加组。

-s Shell文件 指定用户的登录Shell。

-u 用户号 指定用户的用户号，如果同时有-o选项，则可以重复使用其他用户的标识号。

useradd master -d /home/master -m

#### 修改用户密码

passwd 用户名

#### 设置用户密码

passwd 用户名

#### 删除用户

userdel 用户名

#### 查看所有用户

cat /etc/passwd | awk -F : '{print $1}'

### 查找

#### grep

## CentOS

### CentOS7

#### firewall

查看已经开放的端口：

firewall-cmd --list-ports

开启端口

firewall-cmd --zone=public --add-port=80/tcp --permanent

命令含义：

–zone #作用域

–add-port=80/tcp #添加端口，格式为：端口/通讯协议

–permanent #永久生效，没有此参数重启后失效

重启防火墙

firewall-cmd --reload #重启firewall

systemctl stop firewalld.service #停止firewall

systemctl disable firewalld.service #禁止firewall开机启动  
firewall-cmd --state #查看默认防火墙状态（关闭后显示notrunning，开启后显示running）

#### firewall防火墙改为iptables防火墙

##### 关闭firewall

systemctl stop firewalld.service

systemctl disable firewalld.service

systemctl mask firewalld.service

##### 安装iptables防火墙

yum install iptables-services -y

##### 启动设置防火墙

# systemctl enable iptables

# systemctl start iptables

##### 查看防火墙状态

systemctl status iptables

##### 编辑防火墙，增加端口

vi /etc/sysconfig/iptables #编辑防火墙配置文件

-A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT

-A INPUT -m state --state NEW -m tcp -p tcp --dport 80 -j ACCEPT

-A INPUT -m state --state NEW -m tcp -p tcp --dport 3306 -j ACCEPT

:wq! #保存退出

##### 重启配置，重启系统

systemctl restart iptables.service #重启防火墙使配置生效

systemctl enable iptables.service #设置防火墙开机启动

## 权限

# Server

## Nginx

### 简介

Nginx("engine x")是一款是由俄罗斯的程序设计师Igor Sysoev所开发高性能的 Web和 反向代理 服务器，也是一个 IMAP/POP3/SMTP 代理服务器。

在高连接并发的情况下，Nginx是Apache服务器不错的替代品。

### 安装

#### 安装编译工具及库文件

yum -y install make zlib zlib-devel gcc-c++ libtool openssl openssl-devel

#### 安装PCRE

PCRE 作用是让 Nginx 支持 Rewrite 功能。

wget <http://downloads.sourceforge.net/project/pcre/pcre/8.35/pcre-8.35.tar.gz>

tar zxvf pcre-8.35.tar.gz

cd pcre-8.35

./configure

make && make install

pcre-config –version

#### 安装Nginx

wget <http://nginx.org/download/nginx-1.6.2.tar.gz>

tar zxvf nginx-1.6.2.tar.gz

cd nginx-1.6.2

./configure --prefix=../nginx --with-http\_stub\_status\_module --with-http\_ssl\_module --with-pcre=../pcre-8.35

make && make install ##编译安装

$home/nginx/sbin/nginx -v ##查看nginx版本

### nginx.conf配置

$home/nginx/conf/nginx.conf

user nginx nginx; ##Nginx用户及组：用户 组。window下不指定。

worker\_processes 8; ##工作进程：数目。根据硬件调整，通常等于CPU数量或者2倍于CPU。

error\_log logs/error.log; ##错误日志存放路径。

error\_log logs/error.log notice; ##错误日志存放路径。

error\_log logs/error.log info; ##错误日志存放路径。

pid logs/nginx.pid; ##pid（进程标识符）存放路径。

http

{

server

{

listen 80; ##监听端口

server\_name localhost; ##域名

index index.html index.htm index.php;

root /usr/local/webserver/nginx/html; ##站点目录

#######location前缀含义:#################

## = ：精确匹配（必须全部相等）。 ##

## ~ ：大小写敏感。 ##

## ~\* ：忽略大小写。 ##

## ^~ ：只需匹配uri部分。 ##

## @ ：内部服务跳转。 ##

location [ = | ~ | ~\* | ^~ ] uri {

}

}

}

#### location

基础知识:

location 是在 server 块中配置。

可以根据不同的 URI 使用不同的配置（location 中配置），来处理不同的请求。

location 是有顺序的，会被第一个匹配的location 处理。

配置语法:

**location** **[ = | ~ | ~\* | ^~ ] uri** { ... }

**location** **@name** { ... }

前缀含义:

**=** **##精确匹配**

location = / {

#规则

}

# 则匹配到 `http://www.example.com/` 这种请求。

**~ ##大小写敏感**

location ~ /Example/ {

#规则

}

#请求示例

#http://www.example.com/Example/ [成功]

#http://www.example.com/example/ [失败]

**~\* ##大小写忽略**

location ~\* /Example/ {

#规则

}

# 则会忽略 uri 部分的大小写

#http://www.example.com/Example/ [成功]

#http://www.example.com/example/ [成功]

**^~ ##只匹配以 uri 开头**

location ^~ /img/ {

#规则

}

#以 /img/ 开头的请求，都会匹配上

#http://www.example.com/img/a.jpg [成功]

#http://www.example.com/img/b.mp4 [成功]

**@ ##nginx内部跳转**

location /img/ {

error\_page 404 @img\_err;

}

location @img\_err {

##规则

}

**##**以 /img/ 开头的请求，如果链接的状态为 404。则会匹配到 @img\_err 这条规则上。

### 启动Nginx

$home/nginx/sbin/nginx

### 重新载入配置文件

$home/nginx/sbin/nginx -s reload

### 重启Nginx

$home/nginx/sbin/nginx -s reopen

### 停止Nginx

$home/nginx/sbin/nginx -s stop

# 系统

## 可维护性

可维护性用平均维修时间（MTTR）来度量，即系统发生故障后维修和重新恢复正常运行平均花费的时间。系统的可维护性越好，平均维修时间越短。

## 高可用性

计算机系统的可用性用[平均无故障时间](https://baike.baidu.com/item/%E5%B9%B3%E5%9D%87%E6%97%A0%E6%95%85%E9%9A%9C%E6%97%B6%E9%97%B4)（MTTF）来度量，即计算机系统平均能够正常运行多长时间，才发生一次故障。系统的可用性越高，平均无故障时间越长。计算机系统的可用性定义为：MTTF/(MTTF+MTTR) \* 100%。由此可见，计算机系统的可用性定义为系统保持正常运行时间的百分比。

## failover(故障切换）

指系统中其中一项设备或服务失效而无法运作时，另一项设备或服务即可自动接手原失效系统所执行的工作。

# 分布式

## CAP原理

### 一致性（Consistency）

这个和数据库ACID的一致性类似，但这里关注的所有数据节点上的数据一致性和正确性，而数据库的ACID关注的是在在一个事务内，对数据的一些约束。系统在执行过某项操作后仍然处于一致的状态。在分布式系统中，更新操作执行成功后所有的用户都应该读取到最新值。

### 可用性（Availability）

每一个操作总是能够在一定时间内返回结果。需要注意“一定时间”和“返回结果”。“一定时间”是指，系统结果必须在给定时间内返回。“返回结果”是指系统返回操作成功或失败的结果。

### 分区容忍性（Partition tolerance）

是否可以对数据进行分区。这是考虑到性能和可伸缩性。服务对网络分区故障的容错性。

任何一个分布式系统都无法同时满足一致性（Consistency）、可用性（Availability）和分区容忍性（Partition tolerance），最多只能同时满足两项。所以，很多系统在设计之初就要对这三者做出取舍。在互联网领域的绝大多数的场景中，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证“最终一致性”，只要这个最终时间是在用户可以接受的范围内即可。

## 分布式事务

### 两阶段提交协议（2PC）

两阶段提交协议是协调所有分布式原子事务参与者，并决定提交或取消（回滚）的分布式算法。

#### 协议参与者

在两阶段提交协议中，系统一般包含两类机器（或节点）：

##### 协调者（coordinator）

通常一个系统中只有一个；

##### 事务参与者（participants，cohorts或workers）

一般包含多个，在数据存储系统中可以理解为数据副本的个数。

协议中假设每个节点都会记录写前日志（write-ahead log）并持久性存储，即使节点发生故障日志也不会丢失。协议中同时假设节点不会发生永久性故障而且任意两个节点都可以互相通信。

#### 两个阶段的执行

##### 请求阶段（commit-request phase，或称表决阶段，voting phase）

在请求阶段，协调者将通知事务参与者准备提交或取消事务，然后进入表决过程。

在表决过程中，参与者将告知协调者自己的决策：同意（事务参与者本地作业执行成功）或取消（本地作业执行故障）。

##### 提交阶段（commit phase）

在该阶段，协调者将基于第一个阶段的投票结果进行决策：提交或取消。

当且仅当所有的参与者同意提交事务协调者才通知所有的参与者提交事务，否则协调者将通知所有的参与者取消事务。

参与者在接收到协调者发来的消息后将执行相应的操作。

#### 两阶段提交的缺点

同步阻塞问题

##执行过程中，所有参与节点都是事务阻塞型的。

当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。

单点故障

##由于协调者的重要性，一旦协调者发生故障。

参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）

数据不一致

##在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了commit请求。

而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据不一致性的现象。

#### 两阶段提交无法解决的问题

当协调者出错，同时参与者也出错时，两阶段无法保证事务执行的完整性。

考虑协调者再发出commit消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。

那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

### 消息事务+最终一致性

所谓的消息事务就是基于消息中间件的两阶段提交，本质上是对消息中间件的一种特殊利用，它是将本地事务和发消息放在了一个分布式事务里，保证要么本地操作成功成功并且对外发消息成功，要么两者都失败，开源的RocketMQ就支持这一特性，具体原理如下：



1、A系统向消息中间件发送一条预备消息

2、消息中间件保存预备消息并返回成功

3、A执行本地事务

4、A发送提交消息给消息中间件

通过以上4步完成了一个消息事务。对于以上的4个步骤，每个步骤都可能产生错误，下面一一分析：

* 步骤一出错，则整个事务失败，不会执行A的本地操作
* 步骤二出错，则整个事务失败，不会执行A的本地操作
* 步骤三出错，这时候需要回滚预备消息，怎么回滚？答案是A系统实现一个消息中间件的回调接口，消息中间件会去不断执行回调接口，检查A事务执行是否执行成功，如果失败则回滚预备消息
* 步骤四出错，这时候A的本地事务是成功的，那么消息中间件要回滚A吗？答案是不需要，其实通过回调接口，消息中间件能够检查到A执行成功了，这时候其实不需要A发提交消息了，消息中间件可以自己对消息进行提交，从而完成整个消息事务

基于消息中间件的两阶段提交往往用在高并发场景下，将一个分布式事务拆成一个消息事务（A系统的本地操作+发消息）+B系统的本地操作，其中B系统的操作由消息驱动，只要消息事务成功，那么A操作一定成功，消息也一定发出来了，这时候B会收到消息去执行本地操作，如果本地操作失败，消息会重投，直到B操作成功，这样就变相地实现了A与B的分布式事务。原理如下：



虽然上面的方案能够完成A和B的操作，但是A和B并不是严格一致的，而是最终一致的，我们在这里牺牲了一致性，换来了性能的大幅度提升。当然，这种玩法也是有风险的，如果B一直执行不成功，那么一致性会被破坏，具体要不要玩，还是得看业务能够承担多少风险。

### 三阶段提交协议(3PC)

三阶段提交协议在协调者和参与者中都引入超时机制，并且把两阶段提交协议的第一个阶段拆分成了两步：询问，然后再锁资源，最后真正提交。

#### ****三个阶段的执行****

##### CanCommit阶段

3PC的CanCommit阶段其实和2PC的准备阶段很像。

协调者向参与者发送commit请求，参与者如果可以提交就返回Yes响应，否则返回No响应。

##### PreCommit阶段

Coordinator根据Cohort的反应情况来决定是否可以继续事务的PreCommit操作。

根据响应情况，有以下两种可能。

A.假如Coordinator从所有的Cohort获得的反馈都是Yes响应，那么就会进行事务的预执行：

发送预提交请求。Coordinator向Cohort发送PreCommit请求，并进入Prepared阶段。

事务预提交。Cohort接收到PreCommit请求后，会执行事务操作，并将undo和redo信息记录到事务日志中。

响应反馈。如果Cohort成功的执行了事务操作，则返回ACK响应，同时开始等待最终指令。

B.假如有任何一个Cohort向Coordinator发送了No响应，或者等待超时之后，Coordinator都没有接到Cohort的响应，那么就中断事务：

发送中断请求。Coordinator向所有Cohort发送abort请求。

中断事务。Cohort收到来自Coordinator的abort请求之后（或超时之后，仍未收到Cohort的请求），执行事务的中断。

##### DoCommit阶段

该阶段进行真正的事务提交，也可以分为以下两种情况:

执行提交

**A.发送提交请求。**Coordinator接收到Cohort发送的ACK响应，那么他将从预提交状态进入到提交状态。并向所有Cohort发送doCommit请求。

**B.事务提交。**Cohort接收到doCommit请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。

**C.响应反馈。**事务提交完之后，向Coordinator发送ACK响应。

**D.完成事务。**Coordinator接收到所有Cohort的ACK响应之后，完成事务。

中断事务

Coordinator没有接收到Cohort发送的ACK响应（可能是接受者发送的不是ACK响应，也可能响应超时），那么就会执行中断事务。

#### 三阶段提交协议和两阶段提交协议的不同

对于协调者(Coordinator)和参与者(Cohort)都设置了超时机制（在2PC中，只有协调者拥有超时机制，即如果在一定时间内没有收到cohort的消息则默认失败）。

在2PC的准备阶段和提交阶段之间，插入预提交阶段，使3PC拥有CanCommit、PreCommit、DoCommit三个阶段。

PreCommit是一个缓冲，保证了在最后提交阶段之前各参与节点的状态是一致的。

#### 三阶段提交协议的缺点

如果进入PreCommit后，Coordinator发出的是abort请求，假设只有一个Cohort收到并进行了abort操作，而其他对于系统状态未知的Cohort会根据3PC选择继续Commit，此时系统状态发生不一致性。

## 分布式锁

### 分布式锁的实现

#### 基于数据库实现分布式锁

要实现分布式锁，最简单的方式可能就是直接创建一张锁表，然后通过操作该表中的数据来实现了。

当我们要锁住某个方法或资源时，我们就在该表中增加一条记录，想要释放锁的时候就删除这条记录。

这种简单的实现有以下几个问题：

这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。

数据库是单点？搞两个数据库，数据之前双向同步。一旦挂掉快速切换到备库上。

这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得到锁。

没有失效时间？只要做一个定时任务，每隔一定时间把数据库中的超时数据清理一遍。

这把锁只能是非阻塞的，因为数据的insert操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作。

非阻塞的？搞一个while循环，直到insert成功再返回成功。

这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。

非重入的？在数据库表中加个字段，记录当前获得锁的机器的主机信息和线程信息，那么下次再获取锁的时候先查询数据库，如果当前机器的主机信息和线程信息在数据库可以查到的话，直接把锁分配给他就可以了。

基于MySql的InnoDB引擎，在查询语句后面增加for update，数据库会在查询过程中给数据库表增加排他锁。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。

我们可以认为获得排它锁的线程即可获得分布式锁，当获取到锁之后，可以执行方法的业务逻辑，执行完方法之后，可以通过commit方法解锁。

#### 基于缓存（redis，memcached，tair）实现分布式锁

#### 基于Zookeeper实现分布式锁

### 分布式锁应该是怎么样的

这里以方法锁为例，资源锁同理

可以保证在分布式部署的应用集群中，同一个方法在同一时间只能被一台机器上的一个线程执行。

这把锁要是一把可重入锁（避免死锁）。

这把锁最好是一把阻塞锁（根据业务需求考虑要不要这条）。

有高可用的获取锁和释放锁功能。

获取锁和释放锁的性能要好。

## [Session共享](https://blog.csdn.net/qq_34666857/article/details/77112985)

### 粘性session

粘性session是指Ngnix每次都将同一用户的所有请求转发至同一台服务器上，即将用户与服务器绑定。

### 服务器session复制

即每次session发生变化时，创建或者修改，就广播给所有集群中的服务器，使所有的服务器上的session相同。

### session共享

缓存session，使用redis， memcached。

### session持久化

将session存储至数据库中，像操作数据一样操做session。

## Zookeeper

## Dubbo

Dubbo 是一个分布式服务框架，致力于提供高性能和透明化的 **RPC 远程服务调用方案**，以及 **SOA 服务治理方案**。简单的说，Dubbo 就是个服务框架，说白了就是个**远程服务调用的分布式框架**。

* **远程通讯**: 提供对多种基于长连接的 NIO 框架抽象封装（非阻塞 I/O 的通信方式，Mina/Netty/Grizzly），包括多种线程模型，序列化（Hessian2/ProtoBuf），以及“请求-响应”模式的信息交换方式。
* **集群容错**: 提供基于接口方法的透明远程过程调用（RPC），包括多协议支持（自定义 RPC 协议），以及软负载均衡（Random/RoundRobin），失败容错（Failover/Failback），地址路由，动态配置等集群支持。
* **自动发现**: 基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

### 背景

随着互联网的发展，网站应用的规模不断扩大，常规的垂直应用架构已无法应对，分布式服务架构以及流动计算架构势在必行，亟需一个治理系统确保架构有条不紊的演进。



#### 单一应用架构

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。

此时，用于简化增删改查工作量的 数据访问框架(ORM) 是关键。

#### 垂直应用架构

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。

此时，用于加速前端页面开发的 Web框架(MVC) 是关键。

#### 分布式服务架构

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。

此时，用于提高业务复用及整合的 分布式服务框架(RPC) 是关键。

#### 流动计算架构

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。

此时，用于提高机器利用率的 资源调度和治理中心(SOA) 是关键。实际上这样是一种云架构，实现了服务的动态扩展，在高并发情况下，服务端可以快速部署机器，而对于应用端没有任何其他影响。

### 工作原理



#### Provider

暴露服务方称之为“服务提供者”。

#### Consumer

调用[远程服务](https://baike.baidu.com/item/%E8%BF%9C%E7%A8%8B%E6%9C%8D%E5%8A%A1)方称之为“服务消费者”。

#### Registry

服务注册与发现的中心目录服务称之为“服务注册中心”。

#### Monitor

统计服务的调用次数和调用时间的日志服务称之为“服务监控中心”。

### 服务治理



在大规模服务化之前，应用可能只是通过 RMI 或 Hessian 等工具，简单的暴露和引用远程服务，通过配置服务的URL地址进行调用，通过 F5 等硬件进行负载均衡。

当服务越来越多时，服务 URL 配置管理变得非常困难，F5 硬件负载均衡器的单点压力也越来越大。 此时需要一个服务注册中心，动态的注册和发现服务，使服务的位置透明。并通过在消费方获取服务提供方地址列表，实现软负载均衡和 [failover](#_failover(故障切换）)，降低对 F5 硬件负载均衡器的依赖，也能减少部分成本。

当进一步发展，服务间依赖关系变得错踪复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完整的描述应用的架构关系。 这时，需要自动画出应用间的依赖关系图，以帮助架构师理清理关系。

接着，服务的调用量越来越大，服务的容量问题就暴露出来，这个服务需要多少机器支撑？什么时候该加机器？ 为了解决这些问题，第一步，要将服务现在每天的调用量，响应时间，都统计出来，作为容量规划的参考指标。其次，要可以动态调整权重，在线上，将某台机器的权重一直加大，并在加大的过程中记录响应时间的变化，直到响应时间到达阀值，记录此时的访问量，再以此访问量乘以机器数反推总容量。

### 连通性

1. **注册中心**负责服务地址的注册与查找，相当于[目录服务](https://baike.baidu.com/item/%E7%9B%AE%E5%BD%95%E6%9C%8D%E5%8A%A1)，**服务提供者**和**服务消费者**只在启动时与注册中心交互，**注册中心**不转发请求，压力较小。
2. **监控中心**负责统计各服务调用次数，调用时间等，统计先在内存汇总后每分钟一次发送到**监控中心**服务器，并以报表展示。
3. **服务提供者**向**注册中心**注册其提供的服务，并汇报调用时间到**监控中心**，此时间不包含网络开销。
4. **服务消费者**向**注册中心**获取服务提供者地址列表，并根据负载算法直接调用**提供者**，同时汇报调用时间到**监控中心**，此时间包含网络开销。
5. **注册中心**，**服务提供者**，**服务消费者**三者之间均为长连接，监控中心除外。
6. **注册中心**通过[长连接](https://baike.baidu.com/item/%E9%95%BF%E8%BF%9E%E6%8E%A5)感知**服务提供者**的存在，**服务提供者**宕机，**注册中心**将立即推送事件通知**消费者**。
7. **注册中心**和**监控中心**全部宕机，不影响已运行的**提供者**和**消费者**，**消费者**在[本地缓存](https://baike.baidu.com/item/%E6%9C%AC%E5%9C%B0%E7%BC%93%E5%AD%98)了**提供者**列表。
8. **注册中心**和**监控中心**都是可选的，**服务消费者**可以直连**服务提供者**。

### 健壮性

1. **监控中心**宕掉不影响使用，只是丢失部分[采样数据](https://baike.baidu.com/item/%E9%87%87%E6%A0%B7%E6%95%B0%E6%8D%AE)。
2. 数据库宕掉后，**注册中心**仍能通过[缓存](https://baike.baidu.com/item/%E7%BC%93%E5%AD%98)提供服务列表查询，但不能注册新服务。
3. **注册中心**对等[集群](https://baike.baidu.com/item/%E9%9B%86%E7%BE%A4)，任意一台宕掉后，将自动切换到另一台。
4. **注册中心**全部宕掉后，**服务提供者**和**服务消费者**仍能通过本地缓存通讯。
5. **服务提供者**无状态，任意一台宕掉后，不影响使用。
6. **服务提供者**全部宕掉后，**服务消费者**应用将无法使用，并无限次重连等待**服务提供者**恢复。

### 伸缩性

1. **注册中心**为对等集群，可动态增加机器部署实例，所有客户端将自动发现新的**注册中心**
2. **服务提供者**无状态，可动态增加机器部署实例，**注册中心**将推送新的服务提供者信息给**消费者**

### 升级性

当服务集群规模进一步扩大，带动IT治理结构进一步升级，需要实现动态部署，进行流动计算，现有分布式服务架构不会带来阻力：



* Deployer: 自动部署服务的本地代理。（没有实现）
* Repository: 仓库用于存储服务应用发布包。（没有实现）
* Scheduler: 调度中心基于访问压力自动增减服务提供者。（没有实现）
* Admin: 统一管理控制台。（没有实现）

### Dubbo整体设计



各层说明：

* config，配置层，对外配置接口，以ServiceConfig, ReferenceConfig为中心，可以直接new配置类，也可以通过spring解析配置生成配置类。
* proxy，服务代理层，服务接口透明代理，生成服务的客户端Stub和服务器端Skeleton，以ServiceProxy为中心，扩展接口为ProxyFactory。
* registry，注册中心层，封装服务地址的注册与发现，以服务URL为中心，扩展接口为RegistryFactory, Registry, RegistryService。
* cluster，路由层，封装多个提供者的路由及负载均衡，并桥接注册中心，以Invoker为中心，扩展接口为Cluster, Directory, Router, LoadBalance。
* monitor，监控层，RPC调用次数和调用时间监控，以Statistics为中心，扩展接口为MonitorFactory, Monitor, MonitorService。
* protocol，远程调用层，封将RPC调用，以Invocation, Result为中心，扩展接口为Protocol, Invoker, Exporter。
* exchange，信息交换层，封装请求响应模式，同步转异步，以Request, Response为中心，扩展接口为Exchanger, ExchangeChannel, ExchangeClient, ExchangeServer。
* transport，网络传输层，抽象mina和netty为统一接口，以Message为中心，扩展接口为Channel, Transporter, Client, Server, Codec。
* serialize，数据序列化层，可复用的一些工具，扩展接口为Serialization, ObjectInput, ObjectOutput, ThreadPool。

## Redis

### Redis安装

$ wget http://download.redis.io/releases/[redis-3.2.11.tar.gz](http://download.redis.io/releases/redis-3.2.11.tar.gz)

$ tar xzf redis-3.2.11.tar.gz

$ cd redis-3.2.11

$ make

### 单线程架构

Redis使用了**单线程架构**和**I/O多路复用模型**来实现高性能的内存数据库服务。

每次客户端调用都经历了发送命令、执行命令、返回结果三个过程。

其中第2步是重点要讨论的，因为Redis是单线程来处理命令的，所以一条命令从客户端达到服务端不会立刻被执行，所有命令都会进入一个队列中，然后逐个被执行。所以客户端命令的执行顺序是不确定的，但是可以确定不会有两条命令被同时执行，所以两条incr命令无论怎么执行最终结果都是2，不会产生并发问题，这就是Redis单线程的基本模型。但是像发送命令、返回结果、命令排队肯定不像描述的这么简单，Redis使用了I/O多路复用技术来解决I/O的问题。

那么为什么Redis使用单线程模型会达到每秒万级别的处理能力呢？可以将其归结为三点：

第一，纯内存访问，Redis将所有数据放在内存中，内存的响应时长大约为100纳秒，这是Redis达到每秒万级别访问的重要基础。

第二，**非阻塞I/O**，Redis使用**epoll**作为**I/O多路复用**技术的实现，再加上Redis自身的事件处理模型将**epoll**中的连接、读写、关闭都转换为事件，不在网络I/O上浪费过多的时间。

第三，单线程避免了线程切换和竞态产生的消耗。

既然采用单线程就能达到如此高的性能，那么也不失为一种不错的选择，因为单线程能带来几个好处：第一，单线程可以简化数据结构和算法的实现。如果对高级编程语言熟悉的读者应该了解并发数据结构实现不但困难而且开发测试比较麻烦。第二，单线程避免了线程切换和竞态产生的消耗，对于服务端开发来说，锁和线程切换通常是性能杀手。

但是单线程会有一个问题：对于每个命令的执行时间是有要求的。如果某个命令执行过长，会造成其他命令的阻塞，对于Redis这种高性能的服务来说是致命的，所以Redis是面向快速执行场景的数据库。

### 服务端

#### 启动运动端

下面启动redis服务.

$ cd src

$ ./redis-server &

注意这种方式启动redis 使用的是默认配置。也可以通过启动参数告诉redis使用指定配置文件使用下面命令启动。

$ cd src

$ ./redis-server redis.conf &

### 客户端

#### 启动客户端

启动redis服务进程后，就可以使用测试客户端程序redis-cli和redis服务交互了。 比如：

$ cd src

$ ./redis-cli

redis> set foo bar

OK

redis> get foo

"bar"

#### 连接远程服务端

$ redis-cli -h host -p port -a password

#### 关闭服务端

127.0.0.1:6379> shutdown

### 数据结构

#### 全局命令

##### key \*

查看所有键

##### dbsize

键总数，返回当前数据库中键的总数。

##### exists key

检查键是否存在，如果键存在返回1，如果不存在返回0。

##### del key [key …]

删除键，无论值是什么数据结构类型，del命令都可以将其删除。

##### expire key seconds

Redis支持对键添加过期时间，当超过过期时间后，会自动删除键，例如为键hello设置了10秒过期时间：

127.0.0.1:6379> set hello world

OK

127.0.0.1:6379> expire hello 10

(integer) 1

##### ttl key

返回键的剩余过期时间，它有3种返回值：

大于等于0的整数：键剩余的过期时间。

-1：键没设置过期时间。

-2：键不存在

##### type key

返回键的数据结构类型:string,hash,list,set,zset。

#### 内部编码



实际上每种数据结构都有自己底层的内部编码实现，而且是多种实现，这样Redis会在合适的场景选择合适的内部编码。

可以通过object encoding命令查询内部编码：

127.0.0.1:6379> object encoding hello

"embstr"

127.0.0.1:6379> object encoding mylist

"ziplist"

#### string（字符串）

##### set key value [EX seconds] [PX milliseconds] [NX|XX]

set命令有几个选项：

**ex** seconds：为键设置秒级过期时间。

**px** milliseconds：为键设置毫秒级过期时间。

**nx**：键必须不存在，才可以设置成功，用于添加。

**xx**：与nx相反，键必须存在，才可以设置成功，用于更新。

##### mset key value [key value ...]

批量设置值。

##### setex key seconds value

为键设置秒级过期时间。

##### setnx key value

键必须不存在，才可以设置成功，用于添加。

##### get key

获取值。

##### mget key [key ...]

批量获取值。

##### incr key

incr命令用于对值做自增操作，返回结果分为三种情况：

1. 值不是整数，返回错误。
2. 值是整数，返回自增后的结果。
3. 键不存在，按照值为0自增，返回结果为1。

##### decr key

自减

##### incrby key increment

自增指定数字

##### decrby key decrement

自减指定数字

##### incrbyfloat key increment

自增浮点数

#### hash（哈希）

#### list（列表）

#### set（集合）

#### zset（有序集合）

### 持久化

Redis支持RDB和AOF两种持久化机制，持久化功能有效地避免因进程退出造成的数据丢失问题，当下次重启时利用之前持久化的文件即可实现数据恢复。

#### RDB持久化

RDB持久化是把当前进程数据生成快照保存到硬盘的过程，触发RDB持久化过程分为**手动触发**和**自动触发**。

手动触发分别对应**save**和**bgsave**命令：

##### save命令

阻塞当前Redis服务器，直到RDB过程完成为止，对于内存比较大的实例会造成长时间阻塞，线上环境不建议使用。运行save命令对应的Redis日志如下：

\* DB saved on disk

##### bgsave命令

Redis进程执行fork操作创建子进程，RDB持久化过程由子进程负责，完成后自动结束。阻塞只发生在fork阶段，一般时间很短。运行bgsave命令对应的Redis日志如下：

\* Background saving started by pid 3151

\* DB saved on disk

\* RDB: 0 MB of memory used by copy-on-write

\* Background saving terminated with success

显然bgsave命令是针对save阻塞问题做的优化。因此Redis内部所有的涉及RDB的操作都采用bgsave的方式，而save命令已经废弃。

##### 自动触发

除了执行命令手动触发之外，Redis内部还存在自动触发RDB的持久化机制，例如以下场景：

1. 使用save相关配置(redis.conf)，如“save m n”。表示m秒内数据集存在n次修改时，自动触发bgsave。
2. 如果从节点执行全量复制操作，主节点自动执行bgsave生成RDB文件并发送给从节点。
3. 执行debug reload命令重新加载Redis时，也会自动触发save操作。
4. 默认情况下执行shutdown命令时，如果没有开启AOF持久化功能则自动执行bgsave。

##### RDB文件的处理

1. 保存

RDB文件保存在**dir**配置指定的目录下，文件名通过**dbfilename**配置指定。可以通过执行config set dir {newDir}和config set dbfilename {newFileName}运行期动态执行，当下次运行时RDB文件会保存到新目录。

当遇到坏盘或磁盘写满等情况时，可以通过config set dir {newDir}在线修改文件路径到可用的磁盘路径，之后执行bgsave进行磁盘切换，同样适用于AOF持久化文件。

压缩：Redis默认采用LZF算法对生成的RDB文件做压缩处理，压缩后的文件远远小于内存大小，默认开启，可以通过参数config set rdbcompression {yes|no}动态修改。

##### RDB的优缺点

1. **RDB的优点**

RDB是一个紧凑压缩的二进制文件，代表Redis在某个时间点上的数据快照。

非常适用于备份，全量复制等场景。比如每6小时执行bgsave备份，并把RDB文件拷贝到远程机器或者文件系统中（如hdfs），用于灾难恢复。

Redis加载RDB恢复数据远远快于AOF的方式。

1. **RDB的缺点**

RDB方式数据**没办法做到实时持久化/秒级持久化**。因为bgsave每次运行都要执行fork操作创建子进程，属于**重量级操作，频繁执行成本过高**。

RDB文件使用特定二进制格式保存，Redis版本演进过程中有多个格式的**RDB版本**，**存在**老版本Redis服务**无法兼容**新版RDB格式的**问题**。

#### AOF持久化(append only file)

AOF（append only file）持久化：以独立日志的方式记录每次写命令，重启时再重新执行AOF文件中的命令达到恢复数据的目的。AOF的主要作用是解决了数据持久化的实时性，目前已经是Redis持久化的主流方式。理解掌握好AOF持久化机制对我们兼顾数据安全性和性能非常有帮助。

开启AOF功能需要设置配置：**appendonly yes**，默认不开启。AOF文件名通过**appendfilename**配置设置，默认文件名是appendonly.aof。保存路径同RDB持久化方式一致，通过dir配置指定。AOF的工作流程操作：**命令写入（append）**、**文件同步（sync）**、**文件重写（rewrite）**、**重启加载（load）**。

流程如下：

1）所有的写入命令会追加到**aof\_buf（缓冲区）**中。

2）**AOF缓冲区**根据对应的策略向硬盘做同步操作。

3）随着AOF文件越来越大，需要定期对**AOF文件**进行重写，达到压缩的目的。

4）当Redis服务器重启时，可以加载**AOF文件**进行数据恢复。

##### 文件同步

Redis提供了多种AOF缓冲区同步文件策略，由参数appendfsync控制：

|  |  |
| --- | --- |
| 可配置值 | 说明 |
| always | 命令写入aof\_buf后调用系统**fsync**操作同步到AOF文件，**fsync**完成后线程返回。 |
| everysec | 命令写入aof\_buf后调用系统**write**操作，write完成后线程返回。**fsync**同步文件操作由专门线程每秒调用一次。 |
| no | 命令写入aof\_buf后调用系统**write**操作，**不**对AOF文件**做fsync**同步，同步硬盘操作由操作系统负责，通常同步周期最长30秒。 |

系统调用write和fsync说明：

**write**操作会触发延迟写（delayed write）机制。Linux在内核提供页缓冲区用来提高硬盘IO性能。write操作在写入系统缓冲区后直接返回。同步硬盘操作依赖于系统调度机制，例如：缓冲区页空间写满或达到特定时间周期。同步文件之前，如果此时系统故障宕机，缓冲区内数据将丢失。

**fsync**针对单个文件操作（比如AOF文件），做强制硬盘同步，fsync将阻塞直到写入硬盘完成后返回，保证了数据持久化。

除了write、fsync，Linux还提供了sync、fdatasync操作，具体API说明参见：

<http://linux.die.net/man/2/write>，

<http://linux.die.net/man/2/fsync>，

<http://linux.die.net/man/2/sync>，

<http://linux.die.net/man/2/fdatasync>。

* 配置为always时，每次写入都要同步AOF文件，在一般的SATA硬盘上，Redis只能支持大约几百TPS写入，显然跟Redis高性能特性背道而驰，不建议配置。
* 配置为no，由于操作系统每次同步AOF文件的周期不可控，而且会加大每次同步硬盘的数据量，虽然提升了性能，但数据安全性无法保证。
* 配置为everysec，是建议的同步策略，也是默认配置，做到兼顾性能和数据安全性。理论上只有在系统突然宕机的情况下丢失1秒的数据。

##### 重写机制（rewrite）

随着命令不断写入AOF，文件会越来越大，为了解决这个问题，Redis引入AOF重写机制压缩文件体积。AOF文件重写是把Redis进程内的数据转化为写命令同步到新AOF文件的过程。

重写后的AOF文件为什么可以变小？有如下原因：

* 1. 进程内已经超时的数据不再写入文件。
  2. 旧的AOF文件含有无效命令，如del key1、hdel key2、srem keys、set a111、set a222等。重写使用进程内数据直接生成，这样新的AOF文件只保留最终数据的写入命令。
  3. 多条写命令可以合并为一个，如：lpush list a、lpush list b、lpush list c可以转化为：lpush list a b c。为了防止单条命令过大造成客户端缓冲区溢出，对于list、set、hash、zset等类型操作，以64个元素为界拆分为多条。

AOF重写降低了文件占用空间，除此之外，另一个目的是：更小的AOF文件可以更快地被Redis加载。

AOF重写过程可以手动触发和自动触发：

**手动触发**：直接调用**bgrewriteaof**命令。

**自动触发**：根据**auto-aof-rewrite-min-size**和**auto-aof-rewrite-percentage**参数确定自动触发时机。

**auto-aof-rewrite-min-size**：表示运行AOF重写时文件最小体积，默认为64MB。

**auto-aof-rewrite-percentage**：代表当前AOF文件空间（aof\_current\_size）和上一次重写后AOF文件空间（aof\_base\_size）的比值。

自动触发时机=aof\_current\_size>auto-aof-rewrite-min-size&&（aof\_current\_size-aof\_base\_size）/aof\_base\_size>=auto-aof-rewrite-percentage  
其中aof\_current\_size和aof\_base\_size可以在info Persistence统计信息中查看。

### 命令

#### debug

##### debug reload

save当前的rdb文件，并清空当前数据库，重新加载rdb，加载与启动时加载类似，加载过程中只能服务部分只读请求（比如info、ping等）

# 消息队列框架

# 版本管理(版本控制)

## 常用术语

### 仓库（Repository）

受版本控制的所有文件修订历史的共享数据库。

### 工作空间（Workspace)

本地硬盘或Unix 用户帐户上编辑的文件副本。

### 工作树/区（Working tree）

工作区中包含了仓库的工作文件。您可以修改的内容和提交更改作为新的提交到仓库。

### 暂存区（Staging area）

暂存区是工作区用来提交更改（commit）前可以暂存工作区的变化。

### 索引（Index）

索引是暂存区的另一种术语。

### 签入（Checkin）

将新版本复制回仓库。

### 签出（Checkout）

从仓库中将文件的最新修订版本复制到工作空间。

### 提交（Commit）

对各自文件的工作副本做了更改，并将这些更改提交到仓库。

### 冲突（Conflict）

多人对同一文件的工作副本进行更改，并将这些更改提交到仓库。

### 合并（Merge）

将某分支上的更改联接到此主干或同为主干的另一个分支。

### 分支（Branch）

从主线上分离开的副本，默认分支叫master。

### 锁（Lock）

获得修改文件的专有权限。

### 头（HEAD）

头是一个象征性的参考，最常用以指向当前选择的分支。

### 修订（Revision）

表示代码的一个版本状态。Git通过用SHA1 hash算法表示的ID来标识不同的版本。

### 标记（Tags）

标记指的是某个分支某个特定时间点的状态。通过标记，可以很方便的切换到标记时的状态。

## Git

### 创建本地ssh

ssh-keygen -t rsa -C "13322808776@189.cn

### 验证是否配置成功

ssh -T git@github.com

### 本地git仓库关联GitHub仓库

git remote add origin git@github.com:bsc2012/documents.git

### 初始化[git](http://lib.csdn.net/base/28)仓库

git init

### 添加文件

git add ./\*

### 删除文件

git rm filename

### 提交缓存

git commit -m '提交'

### 提交到远程GitHub仓库

git push -u origin master

### ****与GitHub远程仓库同步****

git pull origin master

## github

<https://github.com/bsc2012>

账号:bsc2012 密码:bsc\*\*\*\*\*\*\*213