**Redis**

# Redis 发布订阅

## 简介

Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

Redis 客户端可以订阅任意数量的频道。

## 示例

下图展示了频道 channel1 ，

以及订阅这个频道的三个客户端 —— client2 、 client5 和 client1 之间的关系：



当有新消息通过 PUBLISH 命令发送给频道 channel1 时，这个消息就会被发送给订阅它的三个客户端：



## 命令

**订阅频道：**

SUBSCRIBE channel [channel ...]订阅给定的一个或多个频道的信息

PSUBSCRIBE pattern [pattern ...]订阅一个或多个符合给定模式的频道。

**发布频道：**

PUBLISH channel message 将信息发送到指定的频道。

**退订频道：**

UNSUBSCRIBE [channel [channel ...]] 指退订给定的频道。

PUNSUBSCRIBE [pattern [pattern ...]]退订所有给定模式的频道。

## 应用场景

**这一功能最明显的用法就是构建实时消息系统，比如普通的即时聊天，群聊等功能**

1在一个博客网站中，有100个粉丝订阅了你，当你发布新文章，就可以推送消息给粉丝们。

2微信公众号模式

# Redis多数据库

Redis下，数据库是由一个整数索引标识，而不是由一个数据库名称。默认情况下，一个客户端连接到数据库0。

redis配置文件中下面的参数来控制数据库总数：

**database 16 //(从0开始 1 2 3 …15)**

select 数据库//数据库的切换

移动数据（将当前key移动另个库)

move key名称 数据库

数据库清空：

flushdb //清除当前数据库的所有key

flushall //清除整个Redis的数据库所有key

# Redis事务

Redis 事务可以一次执行多个命令，（按顺序地串行化执行，执行中不会被其它命令插入，不许加塞）

## 简介

Redis 事务可以一次执行多个命令（允许在一次单独的步骤中执行一组命令）， 并且带有以下两个重要的保证：

批量操作在发送 EXEC 命令前被放入队列缓存。

收到 EXEC 命令后进入事务执行，事务中任意命令执行失败，其余的命令依然被执行。

在事务执行过程，其他客户端提交的命令请求不会插入到事务执行命令序列中。

1、Redis会将一个事务中的所有命令序列化，然后按顺序执行

2、执行中不会被其它命令插入，不许出现加赛行为

**一个事务从开始到执行会经历以下三个阶段：**

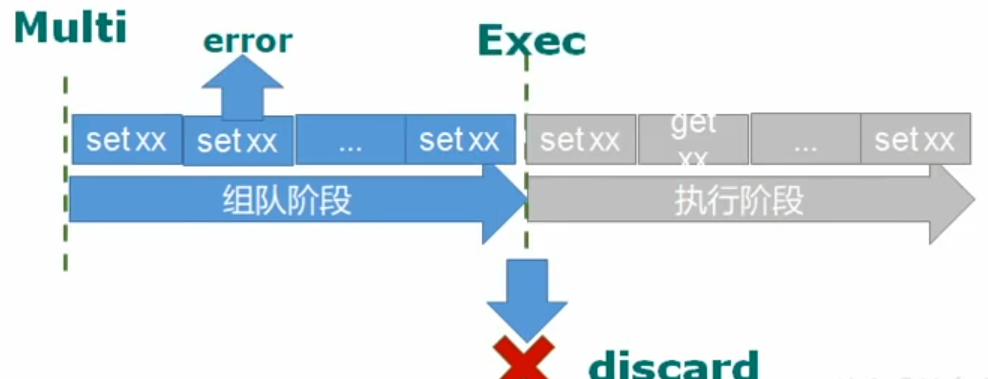
开始事务。

命令入队。

执行事务。

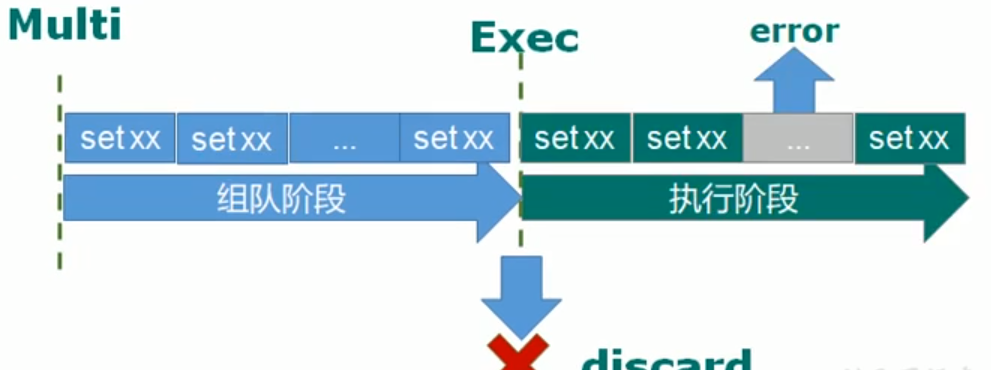
事务的错误处理：

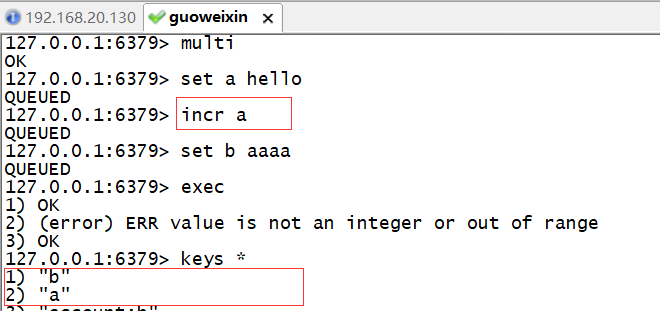
队列中的某个命令出现了报告错误，执行时整个的所有队列都会被取消。



事务的错误处理：

如果执行的某个命令报出了错误，则只有报错的命令不会被执行，而其它的命令都会执行，不会回滚。





## 命令



**Redis 不支持回滚（roll back）**

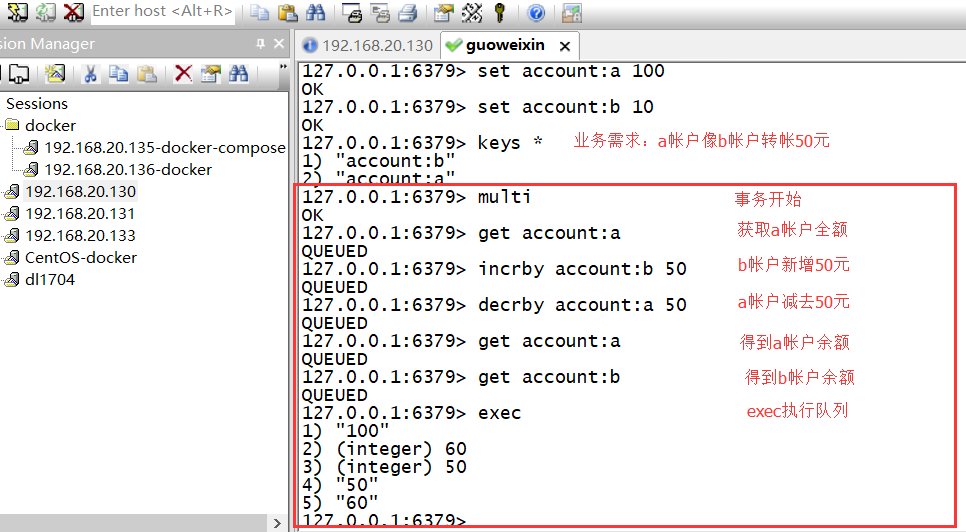
正因为redis不支持回滚功能，才使得redis在事务上可以保持简洁和快速。

## 示例

### 示例1 MULTI EXEC

**转帐功能，A向B帐号转帐50元**

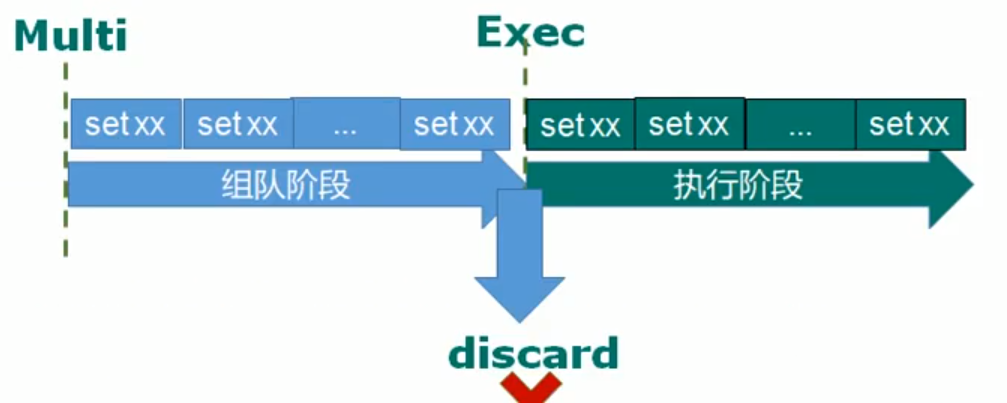
一个事务的例子，它先以 **MULTI** 开始一个事务，然后将多个命令入队到事务中，最后由 **EXEC** 命令触发事务



1输入Multi命令开始，输入的命令都会依次进入命令队列中，但不会执行

2直到输入Exec后，Redis会将之前的命令队列中的命令依次执行

### 示例2 DISCARD放弃队列运行



1输入Multi命令开始，输入的命令都会依次进入命令队列中，但不会执行

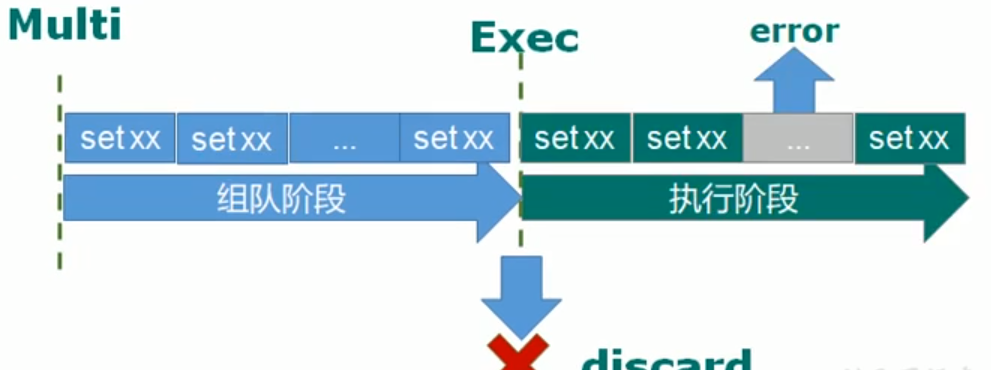
2直到输入Exec后，Redis会将之前的命令队列中的命令依次执行。

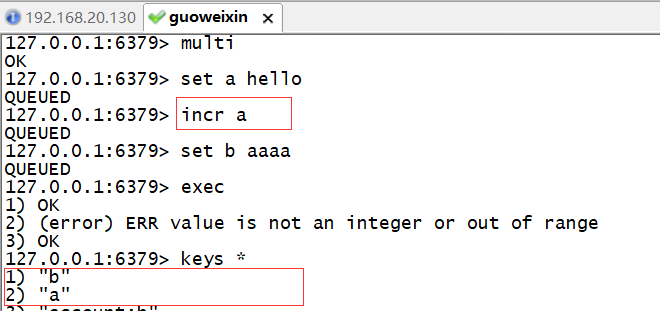
3命令队列的过程中可以通过discard来放弃队列运行

### 示例3事务的错误处理

事务的错误处理：

如果执行的某个命令报出了错误，则只有报错的命令不会被执行，而其它的命令都会执行，不会回滚。

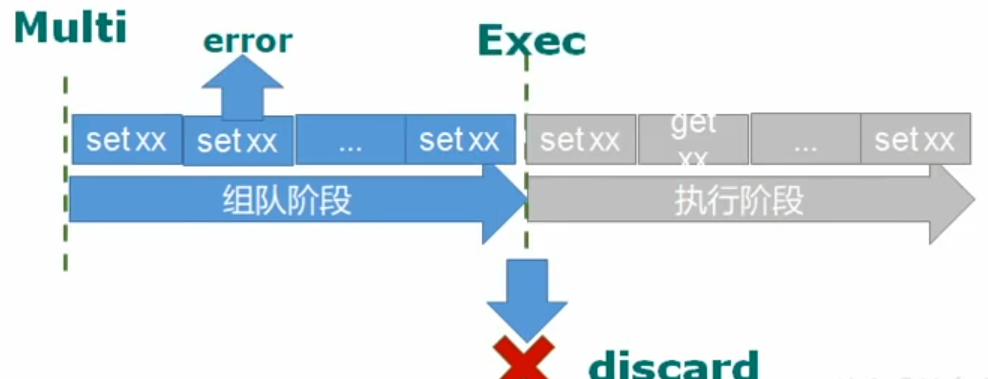


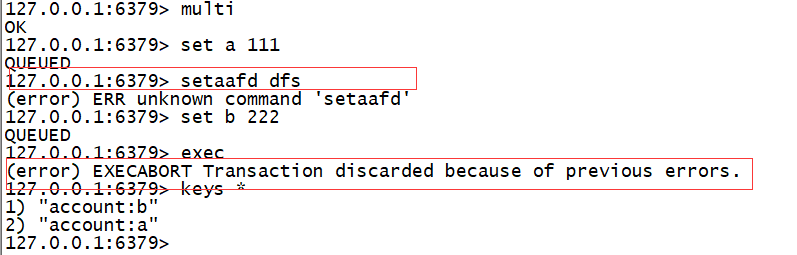


### 示例4事务的错误处理

事务的错误处理：

队列中的某个命令出现了报告错误，执行时整个的所有队列都会被取消。



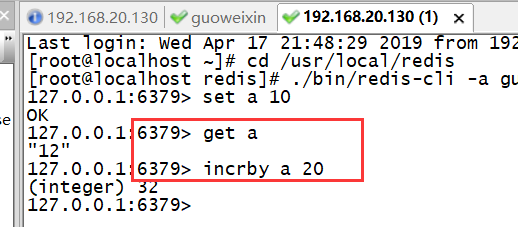
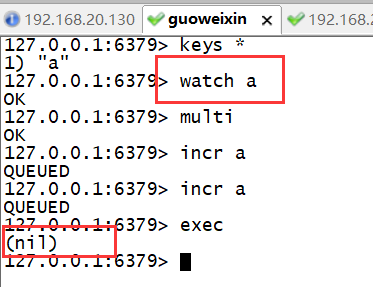


由于之前的错误，事务执行失败

### 示例5事务的WATCH

**[WATCH key [key ...]](http://www.runoob.com/redis/transactions-watch.html)**监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。

**需求：某一帐户在一事务内进行操作，在提交事务前，另一个进程对该帐户进行操作。**



### 示例5 UNWATCH

Redis Unwatch 命令用于取消 WATCH 命令对所有 key 的监视。

如果在执行WATCH命令之后，EXEC命令或DISCARD命令先被执行的话，那就不需要再执行UNWATCH了

## 应用场景

**一组命令必须同时都执行，或者都不执行。**

**我们想要保证一组命令在执行的过程之中不被其它命令插入。**

商品秒杀（活动）。

# Redis 脚本

Redis 脚本使用 Lua 解释器来执行脚本。 Redis 2.6 版本通过内嵌支持 Lua 环境。执行脚本的常用命令为 EVAL。

# Redis数据淘汰策略redis.conf

Redis官方给的警告，当内存不足时，Redis会根据配置的缓存策略淘汰部分Keys，以保证写入成功。当无淘汰策略时或没有找到适合淘汰的Key时，Redis直接返回out of memory错误。

**最大缓存配置**

在 redis 中，允许用户设置最大使用内存大小

maxmemory 512G

**redis 提供6种数据淘汰策略：**

* volatile-lru：从已设置过期时间的数据集中挑选最近最少使用的数据淘汰
* volatile-lfu：从已设置过期的Keys中，删除一段时间内使用次数最少使用的
* volatile-ttl：从已设置过期时间的数据集中挑选最近将要过期的数据淘汰
* volatile-random：从已设置过期时间的数据集中随机选择数据淘汰
* allkeys-lru：从数据集中挑选最近最少使用的数据淘汰
* allkeys-lfu：从所有Keys中，删除一段时间内使用次数最少使用的
* allkeys-random：从数据集中随机选择数据淘汰
* no-enviction（驱逐）：禁止驱逐数据(不采用任何淘汰策略。默认即为此配置),针对写操作，返回错误信息

建议：了解了Redis的淘汰策略之后，在平时使用时应尽量主动设置/更新key的expire时间，主动剔除不活跃的旧数据，有助于提升查询性能

# Redis持久化

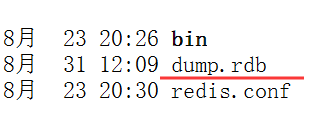
## 简介

**数据存放于：**

**内存：高效、断电（关机）内存数据会丢失**

**硬盘：读写速度慢于内存，断电数据不会丢失**

## RDB



**RDB：是redis的默认持久化机制。 RDB相当于照快照，保存的是一种状态。**

**几十G数据--》几KB快照**

快照是默认的持久化方式。这种方式是就是将内存中数据以快照的方式写入到二进制文件中,默认的文件名为dump.rdb。

**优点：**

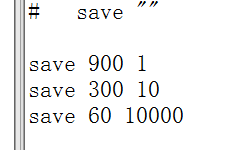
**快照保存数据极快、还原数据极快**

**适用于灾难备份**

**缺点：小内存机器不适合使用,RDB机制符合要求就会照快照**

快照条件：

1、服务器正常关闭时 ./bin/redis-cli shutdown

2、key满足一定条件，会进行快照

vim redis.conf 搜索save

:/save

save 900 1 //每900秒（15分钟）至少1个key发生变化，产生快照

save 300 10 //每300秒（5分钟）至少10个key发生变化，产生快照

save 60 10000 //每60秒（1分钟）至少10000个key发生变化，产生快照

## AOF

由于快照方式是在一定间隔时间做一次的，所以如果redis 意外down 掉的话，就会丢失最后一次快照后的所有修改。如果应用要求不能丢失任何修改的话，可以采用aof 持久化方式。

Append-only file:aof 比快照方式有更好的持久化性，是由于在使用aof 持久化方式时,redis 会将每一个收到的写命令都通过write 函数追加到文件中(默认是appendonly.aof)。当redis 重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。

**有三种方式如下（默认是：每秒 fsync 一次）**

* appendonly yes //启用 aof 持久化方式
* # appendfsync always //收到写命令就立即写入磁盘，最慢，但是保证完全的持久化
* appendfsync everysec //每秒钟写入磁盘一次，在性能和持久化方面做了很好的折中
* # appendfsync no //完全依赖 os，性能最好,持久化没保证

**产生的问题：**

 aof 的方式也同时带来了另一个问题。持久化文件会变的越来越大。例如我们调用 incr test命令 100 次，文件中必须保存全部的 100 条命令，其实有 99 条都是多余的。

# redis缓存与数据库一致性

## 解决方案

### 一、实时同步

对强一致要求比较高的，应采用实时同步方案，即查询缓存查询不到再从DB查询，保存到缓存；更新缓存时，先更新数据库，再将缓存的设置过期(建议不要去更新缓存内容，直接设置缓存过期)。

@Cacheable：查询时使用，注意Long类型需转换为Sting类型，否则会抛异常

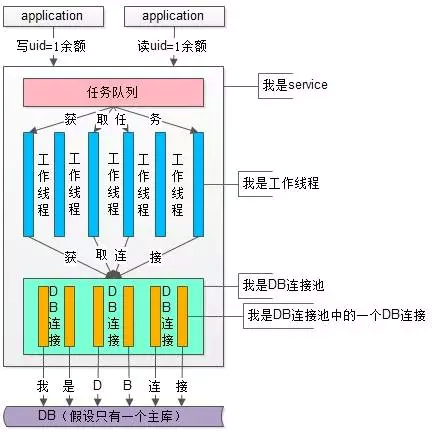
@CachePut：更新时使用，使用此注解，一定会从DB上查询数据

@CacheEvict：删除时使用；

@Caching：组合用法

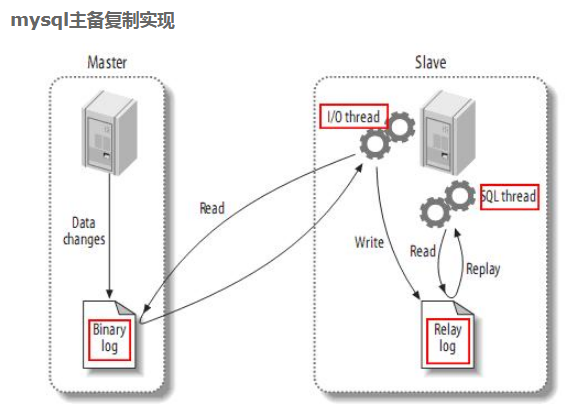
### 二、异步队列

对于并发程度较高的，可采用异步队列的方式同步，可采用kafka等消息中间件处理消息生产和消费。

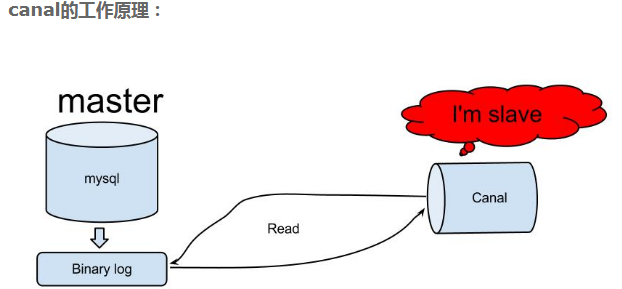


### 三、使用阿里的同步工具canal

canal实现方式是模拟mysql slave和master的同步机制，监控DB bitlog的日志更新来触发缓存的更新，此种方法可以解放程序员双手，减少工作量，但在使用时有些局限性。



1. master将改变记录到二进制日志(binary log)中（这些记录叫做二进制日志事件，binary log events，可以通过show binlog events进行查看）；
2. slave将master的binary log events拷贝到它的中继日志(relay log)；
3. slave重做中继日志中的事件，将改变反映它自己的数据。



1. canal模拟mysql slave的交互协议，伪装自己为mysql slave，向mysql master发送dump协议
2. mysql master收到dump请求，开始推送binary log给slave(也就是canal)
3. canal解析binary log对象(原始为byte流)

### 四、采用UDF自定义函数的方式

面对mysql的API进行编程，利用触发器进行缓存同步，但UDF主要是c/c++语言实现，学习成本高。

# 总结：

## 穿透

缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时需要从[数据库](http://lib.csdn.net/base/mysql" \t "_blank" \o "MySQL知识库)查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

**解决办法：** 持久层查询不到就缓存空结果，查询时先判断缓存中是否exists(key) ,如果有直接返回空，没有则查询后返回，

注意insert时需清除查询的key，否则即便DB中有值也查询不到(当然也可以设置空缓存的过期时间）

## 雪崩

雪崩：缓存大量失效的时候，引发大量查询数据库。  
解决办法：①用锁/分布式锁或者队列串行访问

      ②缓存失效时间均匀分布

## 热点key

热点key:某个key访问非常频繁，当key失效的时候有大量线程来构建缓存，导致负载增加，系统崩溃。

解决办法：

①使用锁，单机用synchronized,lock等，分布式用分布式锁。

②缓存过期时间不设置，而是设置在key对应的value里。如果检测到存的时间超过过期时间则异步更新缓存。

③在value设置一个比过期时间t0小的过期时间值t1，当t1过期的时候，延长t1并做更新缓存操作。

4设置标签缓存，标签缓存设置过期时间，标签缓存过期后，需异步地更新实际缓存