1. Pipline(管道传输)

（1）解释：可以将*多个命令*发送到服务器，而根本不用等待答复，最后一步即可读取答复。
（2）与普通模式的对比：

一般情况下，使用 redis 去 put/get 都是先拿到一个 jedis 实例，然后操作，

然后释放连接；这种模式是请求-响应，请求-响应.

这种模式，下一次请求必须得等第一次请求响应回来之后才可以，因为 redis

是单线程的，按部就班，一步一步来。而 pipeline 管道改变了这种请求模式，客户端可以一次发送多个命令，无须等待服务器的返回，请求，请求，请求，响应，响应，响应.

（3）注意事项：
使用管道发送命令时，服务器将被迫回复一个队列答复，占用很多内存。所以，如果你需要发送大量的命令，最好是把他们按照合理数量分批次的处理，例如 10K 的命令，读回复，然后再发送另一个 10k 的命令，等等。这样速度几乎是

相同的，但是在回复这 10k 命令队列需要非常大量的内存用来组织返回数据内容。
（4）使用：

```
Jedis jedis =
poolFactory.getjedisResourcePool().getResource();
Pipeline pl = jedis.pipelined();
```

实现原理：使用队列，而队列的原理是时先进先出，这样就保证数据的顺序性
（5）同步数据的问题：

A）、Pipeline 有与 redis 形同的操作，但是在数据落盘的时候需要在执行

的方法后添加 sync()方法，如果 insert 时有多条数据，

在数据拼接完之后，在执行 sync()方法，这样可以提高效率。

B）、如果在 hget()时没有 sync()时会报，没有在 hget()同步数据

C）、如果在 hset(),hdel(),hget()获取数据时都没有执行 sync()方法，

但是在最后执行了 pl.close()方法，相当于执行了 sync()方法。

（6）适用场景：

（A）有些系统可能对可靠性要求很高，每次操作都需要立马知道这次操作是否成功，是否数据已经写进 redis 了，那这种场景就不适合

(B)有的系统，可能是批量的将数据写入 redis，允许一定比例的写入失败，那么这种场景就可以使用了

2. Pub/sub（发布/订阅）

建议使用专业的 MQ

3. Redis 的原子操作：事务和 lua 脚本

3.1 事务：

（A）简介：

事务中的所有命令都被序列化并顺序执行。在 Redis 事务的执行过程中，永远不会发生另一个客户端发出的请求。这样可以确保将命令作为单个隔离操作执行

（B）基本指令：MULTI，EXEC，DISCARD 和 WATCH

（C）为什么不支持回滚？：（官方回应）

仅当使用错误的语法（并且在命令队列期间无法检测到该问题）或针对持有错误数据类型的键调用 Redis 命令时，该命令才能失败：这实际上意味着失败的命令是编程错误的结果，还有一种很可能在开发过程中而不是生产过程中发现的错误。

Redis 在内部得到了简化和加快，因为它不需要回滚的能力。

（D）使用案例：

```
WATCH mykey
val = GET mykey
val = val + 1
MULTI
SET mykey $val
EXEC
```

（E）上诉案例解析：WATCH 基于乐观锁实现，若监听的 key 值发生变化，事务自动终止执行

3.2lua 脚本：

（A）（官方）脚本是在 Redis 2.6 中引入的，而事务早已存在，在不久的将来，我们会看到整个用户群只是在使用脚本，这并非不可能

（B）Lua 教程：https://www.runoob.com/lua/lua-tutorial.html

（C）在 redis 中调用 lua 的基本命令：

eval 脚本内容 key 个数 key 列表 参数列表

evalsha 脚本 SHA1 值 key 个数 key 列表 参数列表

script load script 将 Lua 脚本加载到 Redis 内存中

script exists sh1 [sha1 …] 判断 sha1 脚本是否在内存中

script flush 清空 Redis 内存中所有的 Lua 脚本

script kill 杀死正在执行的 Lua 脚本。（如果此时 Lua 脚本正在执行写操作，那么 script kill 将不会生效）

Redis 提供了一个 lua-time-limit 参数，默认 5 秒，它是 Lua 脚本的超时时间，如果 Lua 脚本超时，其他执行正常命令的客户端会收到“Busy Redis is busy running a script”错误，但是不会停止脚本运行，此时可以使用 script kill 杀死正在执行的 Lua 脚本

（D）在 lua 中调用 redis 的基本函数：

redis.call()----发生错误终止运行

redis.pcall()----发生错误继续运行

（E）lua 和 redis 的数据转换

redis 类型	**Lua 类型**
整数回复	数字类型
字符串回复	字符串
多行字符串	table
{err="xx"}	redis.error_reply('xx')
{ok="xx"}	redis.status_reply('xx')

(F) 如何保证原子性？

Redis 保证以原子方式执行脚本：执行脚本时不会执行其他脚本或 Redis 命令，这也意味着执行慢速脚本不是一个好主意

（F）脚本非持久：

1. 重新启动 Redis 实例会刷新脚本缓存，这不是持久性的
2. 一种常见的模式是调用 SCRIPT LOAD 来加载将出现在管道中的所有脚本，然后直接在管道内部使用 EVALSHA，而无需检查由于无法识别脚本哈希而导致的错误。

（G）脚本在集群环境下的复制

1. 无控制（**整个脚本复制---*redis 版本 4 或更早版本***）
2. 选择复制（redis 版本 5 以后默认开启，redis3.2 需要显示开启）

3.2 如下：（版本 5 以后无需加第一行）

```
redis.replicate_commands() -- Enable effects replication.
```

```
redis.call('set','A','1')

redis.set_repl(redis.REPL_NONE)

redis.call('set','B','2')

redis.set_repl(redis.REPL_ALL)

redis.call('set','C','3')
```

运行上述脚本后，结果是将仅在副本和 AOF 上创建键 A 和 C。

（H）脚本中禁用全局变量，改用 redis -- key 或者定义 local 变量

（I）脚本中可以调用 select 函数，2.8.12 以后只对脚本本身产生影响，调用者的数据库不发生变化

（J）在 resp3 协议下调用 lua 的变化

1. 从 Redis 版本 6 开始，服务器支持两种不同的协议。一个叫做 RESP2，它是旧协议：与服务器的所有新连接都以这种模式启动。但是，客户端可以使用 HELLO 命令协商新协议：这样，将连接置于 RESP3 模式。在这种模式下，某些命令（例如 HGETALL）以新的数据类型（在这种情况下为 Map 数据类型）回复。RESP3 协议在语义上更强大，但是大多数脚本只使用 RESP2 即可。
2. 数据类型转换发生了变化

（K）Redis Lua 解释器加载以下 Lua 库，其中 struct，CJSON 和 cmsgpack 是外部库，所有其他库都是标准的 Lua 库

（L）*redis.log* 功能可以从 Lua 脚本写入 Redis 日志文件

（M）*lua-time-limit* 默认执行时间是 *5* 秒，当脚本达到超时时，Redis 不会自动终止它，接收客户端的命令，但回复 busy 错误回复

（N）Lua 调试：从 3.2 版开始

./redis-cli --ldb --eval xx.lua(按步执行，step 命令运行下一行)

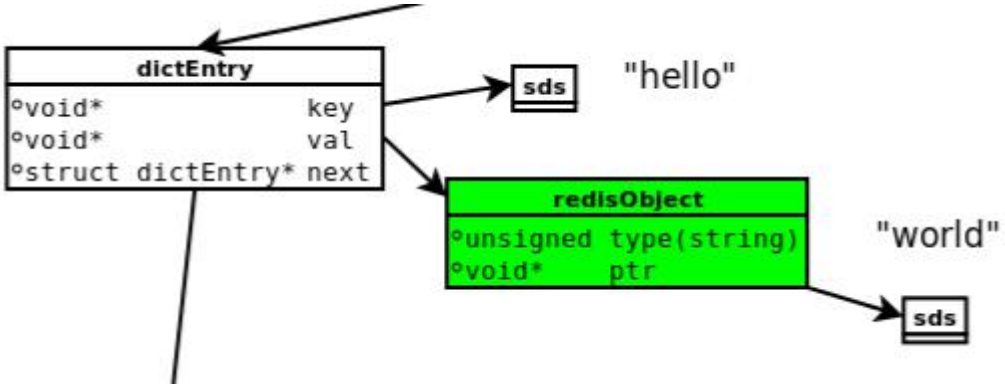
4. 内存优化：（3.0 为例）

（A）查看内存使用情况：info memory

（B）监测工具 treeDMS

（C）key-value 占用内存分析：

（1）如：set hello world，所涉及到的数据模型：



Key-value 存储的数据结构：dictEntry（64 位系统分配 32 字节内存）
Key--hello 存储的数据结构：SDS（实际大小 9+key 的长度）
Value--world 存储的数据结构：redisObject（16 个字节）+value 的大小
jemalloc 为每种数据结构分配内存，分配的规律是：

Category	Spacing	Size
Small	8	[8]
	16	[16, 32, 48, ..., 128]
	32	[160, 192, 224, 256]
	64	[320, 384, 448, 512]
	128	[640, 768, 896, 1024]
	256	[1280, 1536, 1792, 2048]
	512	[2560, 3072, 3584]
Large	4 KiB	[4 KiB, 8 KiB, 12 KiB, ..., 4072 KiB]
Huge	4 MiB	[4 MiB, 8 MiB, 12 MiB, ...]

例如，如果需要存储大小为 130 字节的对象，jemalloc 会将其放入 160 字节的内存单元中。
如果将该对象的大小控制在 128 字节以内，每个对象将减少 32 个字节
•数据量不大的情况下无需过多关注内存大小的优化，否则只会增加开发难度和运维难度
• 数据量较大上亿级的情况下，优化对节省内存相对可观
（2）Str 优化：
字符串类型的内部编码有 3 种：int、embstr（<39 时）、raw
• int 超过 long 范围时自动转换 raw 编码，整数占用 8 字节
• embstr 编码适用于只读场景
（3）list 优化：
内部编码：压缩列表（ziplist）或双端链表（linkedlist）
• ziplist 适用小数据量　512 个 64byte 大小（更节省空间，集中存储）
• linkedlist 大数据量较划算

（4）hash 优化：
内部编码：hashtable 或 ziplist
• ziplist 适用小数据量　512 个 64byte 大小（更节省空间，集中存储）
（5）set 优化：
内部编码：整数集合（intset）或哈希表（hashtable）
• intset 适用 512 个以内的整数 value
（6）zset 优化：
内部编码：ziplist 或跳跃表（skiplist）
• ziplist 适用小数据量　128 个 64byte 大小（更节省空间，集中存储）
（D）优化建议：
（1）由于 jemalloc 分配内存时数值是不连续的，因此 key/value 字符串变化一个字节，可能会引起占用内存很大的变动；在设计时可以利用这一点。
（2）使用整型/长整型可以节省更多空间
（3）共享对象设置：REDIS_SHARED_INTEGERS
（4）避免过度设计
（5）关注内存碎片率：
• 小于 1 时说明使用虚拟内存（即 swap），此时访问速度较慢
• 正常在 1.03 左右相对正常，太大内存碎片较多，考虑重启

5. 过期策略选择：

Lru：最早使用清空
Lfu：最少使用清空
结合场景使用：
（1）低一致性业务建议配置最大内存和淘汰策略的方式使用。
（2）强一致性可以结合使用超时剔除（ex）和主动更新

6. 客户端缓存（6.0）

7. 分布式锁

（1）单机版：

上锁：`SET resource_name my_random_value NX PX 30000`

解锁：

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
```

```
return 0
```

```
End
```

（2）集群版：

```
public interface DistributedLock {
    /**
     * 获取锁
     * @author zhi.li
     * @return 锁标识
     */
    String acquire();

    /**
     * 释放锁
     * @author zhi.li
     * @param indentifier
     * @return
     */
    boolean release(String indentifier);
}

public class RedisDistributedRedLock implements DistributedLock {

    /**
     * redis 客户端
     */
    private RedissonClient redissonClient;

    /**
     * 分布式锁的键值
     */
    private String lockKey;

    private RLock redLock;

    /**
     * 锁的有效时间 10s
     */
    int expireTime = 10 * 1000;

    /**
     * 获取锁的超时时间
```

```
     */
    int acquireTimeout    = 500;

    public RedisDistributedRedLock(RedissonClient redissonClient, String lockKey) {
        this.redissonClient = redissonClient;
        this.lockKey = lockKey;
    }

    @Override
    public String acquire() {
        redLock = redissonClient.getLock(lockKey);
        boolean isLock;
        try{
            isLock        =        redLock.tryLock(acquireTimeout,        expireTime,
TimeUnit.MILLISECONDS);
            if(isLock){
                System.out.println(Thread.currentThread().getName()   +   "   "   +
lockKey + "获得了锁");
                return null;
            }
        }catch (Exception e){
            e.printStackTrace();
        }
        return null;
    }

    @Override
    public boolean release(String indentifier) {
        if(null != redLock){
            redLock.unlock();
            return true;
        }

        return false;
    }
}
public class RedisDistributedRedLockTest {
    static int n = 5;
    public static void secskill() {
        if(n <= 0) {
            System.out.println("抢购完成");
            return;
        }
```

```
        System.out.println(--n);
    }
    public static void main(String[] args) {

        Config config = new Config();
        //支持单机，主从，哨兵，集群等模式
        //此为哨兵模式
        config.useSentinelServers()
                .setMasterName("mymaster")
                .addSentinelAddress("127.0.0.1:26369","127.0.0.1:26379","127.0.
0.1:26389")
                .setDatabase(0);
        Runnable runnable = () -> {
            RedisDistributedRedLock redisDistributedRedLock = null;
            RedissonClient redissonClient = null;
            try {
                redissonClient = Redisson.create(config);
                redisDistributedRedLock                =                new
RedisDistributedRedLock(redissonClient, "stock_lock");
                redisDistributedRedLock.acquire();
                secskill();
                System.out.println(Thread.currentThread().getName() + "正在运
行");
            } finally {
                if (redisDistributedRedLock != null) {
                    redisDistributedRedLock.release(null);
                }

                redissonClient.shutdown();
            }
        };

        for (int i = 0; i < 10; i++) {
            Thread t = new Thread(runnable);
            t.start();
        }
    }
```