Apusic 金蝶天燕

生成时间：2026-05-07

开发手册

金蝶Apusic分布式缓存 V2.0.4

版权声明

本文档所涉及的软件著作权、版权等知识产权已依法进行了注册，由金蝶天燕云计算股份有限公司合法拥有。受《中华人民共和国著作权法》《计算机软件保护条例》《知识产权保护条例》和相关国际版权条约、法律、法规以及其它知识产权法律和条约的保护。未经授权许可，不得非法使用。

免责声明

本文档包含的版权信息由金蝶天燕云计算股份有限公司合法拥有，受法律的保护，金蝶天燕云计算股份有限公司对本文档可能涉及到的非金蝶天燕云计算股份有限公司的信息不承担任何责任。在法律允许的范围内，您可以查阅并仅能够在《中华人民共和国著作权法》规定的合法范围内复制和打印本文档。任何单位和个人未经金蝶天燕云计算股份有限公司书面授权许可，不得使用、修改、再发布本文档的任何部分和内容，否则将被视为侵权，金蝶天燕云计算股份有限公司有依法追究其责任的权利。

本文档如有更新，不另行通知。对本文档中的问题您可向金蝶天燕云计算股份有限公司告知或查询。未经本公司明确授予的任何权利均予保留。

商标声明

Apusic是深圳市金蝶天燕云计算股份有限公司向中华人民共和国国家商标局申请注册的注册商标，注册商标专用权由金蝶天燕合法拥有，受法律保护。未经金蝶天燕的书面许可，任何单位及个人不得以任何方式或理由对该商标的任何部分进行使用、复制、修改、传播、抄录或与其它产品捆绑使用销售。凡侵犯金蝶天燕商标权的，金蝶天燕将依法追究其法律责任。本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

目录

1 前言

本文档为金蝶Apusic分布式缓存（AMDC）V2.0.4的开发使用说明，帮助用户快速学习如何使用金蝶Apusic分布式缓存进行开发。

1.1 适用对象

本文档适用于开发工程师、软件架构师及研发经理等人员。

1.2 相关文档

了解更多AMDC V2.0.4产品相关的信息，请参阅以下AMDC V2.0.4产品手册文档集：

序号	手册文档	说明
1	金蝶Apusic分布式缓存 V2.0.4 快速使用手册	简单介绍了如何快速上手使用AMDC 。
2	金蝶Apusic分布式缓存 V2.0.4 安装手册	详细介绍如何在各操作系统上安装AMDC，以及AMDC服务启停操作，产品的注册过程。
3	金蝶Apusic分布式缓存 V2.0.4 缓存核心用户手册	详细介绍 AMDC 相关功能的使用、配置、管理及配套工具的使用方法。
4	金蝶Apusic分布式缓存 V2.0.4 管控台用户手册	详细介绍AMDC管控台相关功能的使用和操作说明。
5	金蝶Apusic分布式缓存 V2.0.4 开发手册	详细介绍基于各开发语言进行AMDC客户端应用开发的说明。
6	金蝶Apusic分布式缓存 V2.0.4 迁移手册	详细介绍AMDC历史版本迁移升级到V2.0.4版本的说明，以及Redis迁移到AMDC的说明。
7	金蝶Apusic分布式缓存 V2.0.4 运维手册	详细介绍AMDC的监控、运维、安全加固等运维说明。
8	金蝶Apusic分布式缓存 V2.0.4 性能优化手册	详细介绍AMDC性能调优的说明。

1.3 技术支持

AMDC产品提供全面的技术支持服务，您可以通过以下方式获得技术支持：

- 网址：www.apusic.com
- 电话：400-855-5800
- 邮箱：support@apusic.com
- 金蝶云社区：https://vip.kingdee.com/?productId=73&productLineId=14&lang=zh-CN

您在取得技术支持时，请提供如下信息：

1. 您的姓名
2. 公司信息与联系方式
3. 操作系统及其版本
4. 产品版本号
5. 出现异常及错误的日志、截图等详细信息

2 客户端应用开发

我们将使用 `Java` 、 `Python` 、 `Go` 三种比较常用的语言去展示AMDC对各种语言Redis客户端的兼容性。

2.1 Java应用开发

java开发Redis客户端应用推荐使用Jedis。Jedis 是Redis 官方推荐的、轻量级且高性能的同步 Java 客户端，它以接近原生命令的方式让 Java 开发者能够方便、高效地连接、操作和管理Redis数据库。

以下通过使用Jedis连接AMDC的单机、哨兵和集群模式部署的缓存核心服务，并进行数据读写操作提供开发示例。

- 单节点连接与使用

```
Jedis amdc = new Jedis("172.24.4.212", 6359);
System.out.println(amdc.ping());
System.out.println(amdc.set("key11", "value11"));
System.out.println(amdc.get("key11"));
```

- 哨兵模式连接与使用

```
Set<String> sentinels = new HashSet<String>(Arrays.asList(
    "172.24.6.110:27000",
    "172.24.6.110:27001",
    "172.24.6.110:27002"
));
JedisSentinelPool pool = new JedisSentinelPool("mymaster",
sentinels, jedisPoolConfig);
Jedis jedis = pool.getResource();
System.out.println(jedis.set("key12", "value12"));
System.out.println(jedis.get("key12"));
```

- 集群模式连接与使用

```
Set<HostAndPort> clusterNodes = new HashSet<>();
clusterNodes.add(new HostAndPort("172.24.6.110", 7000));
clusterNodes.add(new HostAndPort("172.24.6.110", 7001));
```

```
clusterNodes.add(new HostAndPort("172.24.6.110", 7002));
clusterNodes.add(new HostAndPort("172.24.6.110", 7003));
clusterNodes.add(new HostAndPort("172.24.6.110", 7004));
clusterNodes.add(new HostAndPort("172.24.6.110", 7005));
JedisCluster ac = new JedisCluster(clusterNodes,
genericObjectPoolConfig);
System.out.println(ac.set("test1", "a"));
System.out.println(ac.set("1test", "b"));
System.out.println(ac.set("aaatest", "c"));
System.out.println("-------------------------");
System.out.println(ac.get("test1"));
System.out.println(ac.get("1test"));
System.out.println(ac.get("aaatest"));
```

2.1.1 Spring Boot项目集成

AMDC兼容Redis数据存储协议，外部客户端可无缝切换使用AMDC产品。

2.1.1.1 创建项目并添加配置

```
# application.yml
spring:
  redis:
    host: localhost       # Redis地址
    port: 6379            # 端口
    timeout: 2000ms       # 连接超时时间
    database: 0           # 使用0号数据库（0-15）

    # 连接池配置（重要！）
    lettuce:
      pool:
        max-active: 16    # 最大连接数
        max-idle: 8       # 最大空闲连接
        min-idle: 4       # 最小空闲连接
        max-wait: 1000ms # 获取连接最大等待时间
```

2.1.1.2 配置RedisTemplate

```
@Configuration
public class RedisConfig {

    @Bean
    public RedisTemplate<String, Object> redisTemplate(
            RedisConnectionFactory factory) {

        RedisTemplate<String, Object> template = new RedisTemplate<>
();
        template.setConnectionFactory(factory);

        // 使用String序列化Key（重要！）
        template.setKeySerializer(new StringRedisSerializer());
        template.setHashKeySerializer(new StringRedisSerializer());

        // 使用JSON序列化Value
        template.setValueSerializer(new
Jackson2JsonRedisSerializer<>(Object.class));
        template.setHashValueSerializer(new
Jackson2JsonRedisSerializer<>(Object.class));

        return template;
    }
}
```

2.1.1.3 使用示例

下面是缓存参与审批流程状态管理的一个实例。

```
@Service
public class ApprovalService {
    @Autowired
    private RedisTemplate<String, Object> redisTemplate;
```

```
    /**
     * 存储审批状态
     */
    public void saveApprovalStatus(String approvalId,
                                   String status,
                                   String approver) {
        String key = "approval:status:" + approvalId;

        Map<String, String> data = new HashMap<>();
        data.put("status", status);
        data.put("approver", approver);
        data.put("updateTime", LocalDateTime.now().toString());
        data.put("expireAt", LocalDateTime.now()
            .plusDays(7).toString()); // 审批数据保留7天

        // 使用Hash存储
        redisTemplate.opsForHash().putAll(key, data);

        // 设置过期时间
        redisTemplate.expire(key, 7, TimeUnit.DAYS);
    }

    /**
     * 获取审批进度
     */
    public Map<Object, Object> getApprovalProgress(String
approvalId) {
        String key = "approval:status:" + approvalId;
        return redisTemplate.opsForHash().entries(key);
    }

    /**
     * 批量获取待办事项
     */
    public List<String> getPendingApprovals(String approver) {
        String key = "approval:pending:" + approver;
```

```
        List<Object> list = redisTemplate.opsForList().range(key, 0,
-1);

        return list != null ?
            list.stream()
                .map(Object::toString)
                .collect(Collectors.toList()) :
            Collections.emptyList();
    }
}
```

2.1.2 Redisson 使用缓存示例

Redisson 是Redis 的 Java 客户端与实时数据平台，为开发者提供最方便、最易用的使用方式。通过 Redisson 对象，它在 Java 代码与Redis 之间建立了优雅的抽象层，让开发者专注于业务逻辑与数据模型。同时，Redisson 极大地扩展了Redis的功能，实现了大量原生所不具备的分布式特性，包括分布式集合、分布式锁、分布式对象、分布式服务等。

本节演示AMDC对Redission框架的开发支持和兼容特性，提供相关代码示例供参考。

2.1.2.1 快速开始

我们需要先进行依赖配置

```
<!-- pom.xml -->
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson-spring-boot-starter</artifactId>
    <version>3.27.1</version>
</dependency>
```

```
# application.yml - 单机模式
spring:
  redis:
    redisson:
      config: |
        singleServerConfig:
          address: "redis://127.0.0.1:6379"
```

```
    password: null
    database: 0
    # 连接池设置
    connectionPoolSize: 64
    connectionMinimumIdleSize: 24
    # 超时设置（毫秒）
    connectTimeout: 10000
    timeout: 3000
    retryAttempts: 3
  # 使用JSON序列化
  codec: !<org.redisson.codec.JsonJacksonCodec> {}
```

以下是在代码中集成的不同模式的缓存的使用方式和部分参数的配置。

```
@Configuration
public class RedissonConfig {

    @Bean(destroyMethod = "shutdown")
    public RedissonClient redissonClient() {
        Config config = new Config();

        // 单机模式
        config.useSingleServer()
              .setAddress("redis://127.0.0.1:6379")
              .setPassword("yourPassword")
              .setDatabase(0)
              .setConnectionPoolSize(64)
              .setConnectionMinimumIdleSize(10);

        // 集群模式（金融政务生产环境推荐）
        // config.useClusterServers()
        //       .addNodeAddress(
        //           "redis://node1:6379",
        //           "redis://node2:6379",
        //           "redis://node3:6379"
        //       )
```

```
        //        .setScanInterval(2000); // 集群状态扫描间隔

        // 哨兵模式
        // config.useSentinelServers()
        //        .setMasterName("mymaster")
        //        .addSentinelAddress(
        //            "redis://sentinel1:26379",
        //            "redis://sentinel2:26379"
        //        );

        // 序列化配置
        config.setCodec(new JsonJacksonCodec());

        // 线程配置
        config.setThreads(16);  // 处理Redis响应的线程数
        config.setNettyThreads(32); // Netty I/O线程数

        return Redisson.create(config);
    }
}
```

同样的，我们使用审批流的例子：

```
@Service
public class ApprovalWorkflowService {

    @Autowired
    private RedissonClient redisson;

    /**
     * 分布式审批流程
     */
    public void processApproval(String applicationId,
                                String approver,
                                boolean approved) {
```

```
        // 1. 获取流程锁
        RLock flowLock = redisson.getLock("approval:flow:" +
applicationId);

        if (flowLock.tryLock()) {
            try {
                // 2. 获取当前流程状态
                RMap<String, String> flowState =
                    redisson.getMap("approval:state:" +
applicationId);

                String currentStep = flowState.get("currentStep");
                String status = flowState.get("status");

                // 3. 检查是否可以审批
                if (!"PENDING".equals(status)) {
                    throw new IllegalStateException("申请已处理完成");
                }

                // 4. 更新审批状态
                flowState.put("approver", approver);
                flowState.put("approved", String.valueOf(approved));
                flowState.put("approvalTime",
LocalDateTime.now().toString());
                flowState.put("status", approved ? "APPROVED" :
"REJECTED");

                // 5. 发布审批完成事件
                RTopic topic =
redisson.getTopic("approval:completed");
                topic.publishAsync(new ApprovalEvent(applicationId,
approved));

                // 6. 记录审批历史
                RList<ApprovalRecord> history =
                    redisson.getList("approval:history:" +
```

```
applicationId);
                history.add(new ApprovalRecord(approver, approved,
LocalDateTime.now()));

            } finally {
                flowLock.unlock();
            }
        }
    }

    /**
     * 多级会签（需要所有审批人同意）
     */
    public boolean multiSignApproval(String applicationId,
List<String> approvers) {
        String latchKey = "approval:latch:" + applicationId;
        RCountDownLatch latch =
redisson.getCountDownLatch(latchKey);

        // 初始化计数器
        latch.trySetCount(approvers.size());

        // 每个审批人独立处理
        for (String approver : approvers) {
            // 异步提交给审批人
            submitToApprover(applicationId, approver, latchKey);
        }

        try {
            // 等待所有审批人完成（最多1小时）
            return latch.await(1, TimeUnit.HOURS);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return false;
        }
```

```
    }
}s
```

2.2 Python应用开发

- 单节点连接与使用

```
import redis
import time

r = redis.Redis(host='172.24.4.212', port=6359, db=0,
decode_responses=True)
print(r.set("key1", "value1"))
print(r.get("key1"))
r.expire("key1", 10)
print(r.ttl("key1"))
time.sleep(1)
print(r.ttl("key1"))
```

- 哨兵模式连接与使用

```
from redis.sentinel import Sentinel

SENTINELADDRS = [("172.21.33.69", "26359"), ("172.21.33.69",
"26360"), ("172.21.33.69", "26361")]
def sentinel():
    amdc = redis.Sentinel(sentinels=SENTINELADDRS)
    # 获取主实例
    master = amdc.master_for("mymaster")
    # 获取从实例
    slave = amdc.slave_for("mymaster")
    # 主实例中写入
    print(master.set("key2"))
    time.sleep(2)
```

```
    # 从实例中获取
    print(slave.get("key2"))
```

- 集群模式连接与使用

```
from redis.cluster import RedisCluster

CLUSTERADDRS = [{"host":"172.21.33.69", "port":"6359"},
{"host":"172.21.33.69", "port":"6360"}, {"host":"172.21.33.69",
"port":"6361"}]
def cluster():

amdc_cluster=RedisCluster(startup_nodes=CLUSTERADDRS,decode_responses=T
   print(amdc_cluster.set("key3","value3"))
   print(amdc_cluster.get("key3"))
```

2.3 Go应用开发

- 单节点连接与使用

```
package main

import (
    "fmt"
    "github.com/go-redis/redis"
)

func main() {
    rdb := redis.NewClient(&redis.Options{
        Addr: "172.21.33.69:6359",
    })
    if err := rdb.Ping().Err(); err != nil {
        fmt.Println("Ping error:", err)
    } else {
        fmt.Println("Ping:", rdb.Ping().Val())
```

```
    }
}
```

- 哨兵模式连接与使用

```
package main

import (
    "fmt"
    "github.com/go-redis/redis/v8"
)

func main() {
    rdb := redis.NewFailoverClient(&redis.FailoverOptions{
        MasterName: "mymaster",
        SentinelAddrs: []string{"172.21.33.69:26359",
"172.21.33.69:26360", "172.21.33.69:26361"},
    })
    defer rdb.Close()
    _, err := rdb.Set("hahakey", "hahavalue", 0).Result()
    if err != nil {
        fmt.Println("Set error:", err)
    } else {
        fmt.Println("Set hahakey: hahavalue")
    }
}
```

- 集群模式连接与使用

```
package main

import (
    "fmt"
    "github.com/go-redis/redis/v8"
)
```

```
func main() {
    addrs := []string{"172.21.33.69:6359", "172.21.33.69:6360",
"172.21.33.69:6361"}
    rdb := redis.NewClusterClient(&redis.ClusterOptions{
        Addrs: addrs,
    })
    if err := rdb.Ping().Err(); err != nil {
        fmt.Println("Ping error:", err)
    } else {
        fmt.Println("Ping:", rdb.Ping().Val())
    }
}
```

2.4 AMDC操作命令

命令	说明
ACL LOAD	从配置的ACL文件中重新加载ACL
ACL SAVE	在已配置的ACL文件中保存当前的ACL规则
ACL LIST	列出ACL配置文件格式的当前ACL规则
ACL USERS	列出所有已配置的ACL规则的用户名
ACL GETUSER username	获取特定ACL用户的规则
ACL SETUSER username [rule [rule ...]]	修改或创建特定ACL用户的规则
ACL DELUSER username [username ...]	删除指定的ACL用户和相关规则
ACL CAT [categoryname]	列出类别内的ACL类别或命令
ACL GENPASS [bits]	为ACL用户生成伪谐像安全密码
ACL WHOAMI	返回与当前连接关联的用户的名称
ACL LOG [count or RESET]	列出最新的ACL安全事件
ACL HELP	显示有关不同的子命令的有用文本
APPEND key value	将值附加到键

ASKING	在-ask重定向后由群集客户端发送
AUTH [username] password	对服务器进行身份验证
BGREWRITEAOF	异步重写仅附加文件
BGSAVE [SCHEDULE]	异步将数据集保存到磁盘
BITCOUNT key [start end [BYTE|BIT]]	计数字符串中的设置位
BITFIELD key [GET encoding offset] [SET encoding offset value] [INCRBY encoding offset increment] [OVERFLOW WRAP|SAT|FAIL]	在字符串上执行任意位字段整数操作
BITFIELD_RO key GET encoding offset	在字符串上执行任意位字段整数操作，禁区的只读变体
BITOP operation destkey key [key ...]	在字符串之间执行按位操作
BITPOS key bit [start [end [BYTE|BIT]]]	在字符串中找到第一个位设置或清除
BLPOP key [key ...] timeout	阻塞式删除并返回第一个元素
BRPOP key [key ...] timeout	阻塞式删除并返回最后一个元素
BRPOPLPUSH srouce target timeout	从列表中取出最后一个元素，并插入到另外一个列表的头部； 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。
BLMOVE source destination FROM-TOP|FROM-BOTTOM TO-TOP| TO-BOTTOM timeout	阻塞式返回并删除存储在源列表的第一个或最后一个元素（头尾取决于wherefrom参数），并将该元素压入到存储目标列表的第一个或最后一个元素（头尾取决于whereto参数）
LMPOP numkeys key [key ...] LEFT|RIGHT [COUNT count]	从提供的键名列表中弹出第一个非空列表键中的一个或多个元素
BLMPOP timeout numkeys key [key ...] LEFT|RIGHT [COUNT count]	阻塞式从提供的键名列表中弹出第一个非空列表键中的一个或多个元素；或阻塞连接直到另一个客户端推送到它或直到超时
BZPOPMIN key [key ...] timeout	阻塞式删除并将成员从一个或多个有序集中返回最低分数，或阻塞连接直到另一个客户端推送到它或直到超时

BZPOPMAX key [key ...] timeout	阻塞式删除并将成员从一个或多个有序集中返回最高分数，或阻塞连接直到另一个客户端推送到它或直到超时
CLIENT CACHING YES|NO	指示服务器在下一个请求中跟踪或不键
CLIENT ID	返回当前连接的客户端ID
CLIENT INFO	返回有关当前客户端连接的信息
CLIENT KILL [ip:port] [ID client-id] [TYPE normal|master|slave|pubsub] [USER username] [ADDR ip:port] [LADDR ip:port] [SKIPME yes/no]	杀死客户的连接
CLIENT LIST [TYPE normal|master|replica|pubsub] [ID client-id [client-id ...]]	获取客户端连接列表
CLIENT GETNAME	获取当前的连接名称
CLIENT GETREDIR	获取跟踪通知重定向客户端ID（如果有）
CLIENT UNPAUSE	恢复暂停的客户的处理
CLIENT PAUSE timeout [WRITE|ALL]	停止客户端的处理命令一段时间
CLIENT REPLY ON|OFF|SKIP	指示服务器是否回复命令
CLIENT SETNAME connection-name	设置当前连接名称
CLIENT TRACKING ON|OFF [REDIRECT client-id] [PREFIX prefix [PREFIX prefix ...]] [BCAST] [OPTIN] [OPTOUT] [NOLOOP]	启用或禁用服务器辅助客户端缓存支持
CLIENT TRACKINGINFO	返回关于当前连接的服务器辅助客户端缓存的信息
CLIENT UNBLOCK client-id [TIMEOUT|ERROR]	取消阻止从不同连接中阻止阻止命令的客户端
COMMAND	获取 Redis 命令详细信息数组
COMMAND COUNT	获取Redis命令的总数
COMMAND GETKEYS	给定完整的 Redis 命令提取键
COMMAND INFO command-name [command-name ...]	获取特定Redis命令详细信息的数组
CONFIG GET parameter [parameter ...]	获取配置参数的值
CONFIG REWRITE	用内存配置重写配置文件

CONFIG SET parameter value [parameter value ...]	将配置参数设置为给定的值
CONFIG RESETSTAT	重置 INFO 返回的统计信息
COPY source destination [DB destination-db] [REPLACE]	复制一个键
DBSIZE	返回所选数据库中的键数
DEBUG OBJECT key	获取有关键的调试信息
DEBUG SEGFAULT	使服务器崩溃
DECR key	一个键的整数值减一
DECRBY key decrement	按给定的数字减少一个键的整数值
DEL key [key ...]	删除一个键
DISCARD	放弃在多次执行后发出的所有命令
DUMP key	返回存储在指定键中的值的序列化版本
ECHO message	回显给定的字符串
EVAL script numkeys [key [key ...]] [arg [arg ...]]	执行一个Lua脚本服务器端
EVALSHA sha1 numkeys [key [key ...]] [arg [arg ...]]	执行 Lua 脚本服务器端
EXEC	执行 MULTI
EXISTS key [key ...]	判断key是否存在
EXPIRE key seconds [NX|XX|GT|LT]	以秒为单位设置键的生存时间
EXPIREAT key timestamp [NX|XX|GT|LT]	将键的到期时间设置为 UNIX 时间戳
EXPIRETIME key	获取键的到期 Unix 时间戳
FAILOVER [TO host port [FORCE]] [ABORT] [TIMEOUT milliseconds]	在此服务器与其副本之一之间启动协调故障转移
FLUSHALL [ASYNC|SYNC]	从所有数据库中删除所有键
FLUSHDB [ASYNC|SYNC]	从当前数据库中删除所有键
GEOADD key [NX|XX] [CH] longitude latitude member [longitude latitude member ...]	在使用有序集表示的地理空间索引中添加一个或多个地理空间项目
GEOHASH key member [member ...]	将地理空间索引的成员作为标准 geohash 字符串返回

GEOPOS key member [member ...]	返回地理空间索引成员的经度和纬度
GEODIST key member1 member2 [m|km|ft|mi]	返回地理空间索引的两个成员之间的距离
GEORADIUS key longitude latitude radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count [ANY]] [ASC|DESC] [STORE key] [STOREDIST key]	查询表示地理空间索引的有序集，以获取与点的给定最大距离匹配的成员
GEORADIUSBYMEMBER key member radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count [ANY]] [ASC|DESC] [STORE key] [STOREDIST key]	查询表示地理空间索引的有序集以获取与成员的给定最大距离匹配的成员
GEOSEARCH key [FROMMEMBER member] [FROMLONLAT longitude latitude] [BYRADIUS radius m|km|ft|mi] [BYBOX width height m|km|ft|mi] [ASC|DESC] [COUNT count [ANY]] [WITHCOORD] [WITHDIST] [WITHHASH]	查询表示地理空间索引的有序集以获取框或圆区域内的成员
GEOSEARCHSTORE destination source [FROMMEMBER member] [FROMLONLAT longitude latitude] [BYRADIUS radius m|km|ft|mi] [BYBOX width height m|km|ft|mi] [ASC|DESC] [COUNT count [ANY]] [STOREDIST]	查询表示地理空间索引的有序集以获取框或圆区域内的成员，并将结果存储在另一个键中
GET key	获取一个键的值
GETBIT key offset	返回存储在 key
GETDEL key	的字符串值中偏移量处的位值，获取某个key的值并删除key
GETRANGE key start end	获取存储在键中的字符串的子字符串
GETSET key value	设置一个键的字符串值并返回它的旧值
HDEL key field [field ...]	删除一个或多个哈希字段
HELLO [protover [AUTH username password] [SETNAME clientname]]	与 Redis 握手
HEXISTS key field	判断一个hash字段是否存在
HGET key field	获取哈希字段的值
HGETALL key	获取哈希中的所有字段和值
HINCRBY key field increment	将散列字段的整数值增加给定的数字
HINCRBYFLOAT key field increment	将哈希字段的浮点值增加给定的数量

HKEYS key	获取散列中的所有字段
HLEN key	获取哈希中的字段数
HMGET key field [field ...]	获取所有给定哈希字段的值
HMSET key field value [field value ...]	将多个哈希字段设置为多个值
HSET key field value [field value ...]	设置一个哈希字段的字符串值
HSETNX key field value	设置一个哈希字段的值，仅当该字段不存在时
HRANDFIELD key [count [WITHVALUES]]	从散列中获取一个或多个随机字段
HSTRLEN key field	获取哈希字段值的长度
HVALS key	获取散列中的所有值
INCR key	键的整数值加一
INCRBY key increment	将键的整数值增加给定的数量
INCRBYFLOAT key increment	将键的浮点值增加给定的数量
INFO [section]	获取有关服务器的信息和统计信息
LOLWUT [VERSION version]	展示一些计算机艺术和 Redis 版本
KEYS pattern	找到所有匹配给定模式的键
LASTSAVE	获取上次成功保存到磁盘的 UNIX 时间戳
LINDEX key index	通过索引从列表中获取元素
LINSERT key BEFORE|AFTER pivot element	在列表中的另一个元素之前或之后插入一个元素
LLEN key	获取列表的长度
LPOP key [count]	删除并获取列表中的第一个元素
LPOS key element [RANK rank] [COUNT num-matches] [MAXLEN len]	返回列表中匹配元素的索引
LPUSH key element [element ...]	将一个或多个元素添加到列表中
LPUSHX key element [element ...]	将元素添加到列表中，仅当列表存在时
LRANGE key start stop	从列表中获取一系列元素

LREM key count element	从列表中删除元素
LSET key index element	通过索引设置列表中元素的值
LTRIM key start stop	将列表修剪到指定范围
MEMORY DOCTOR	输出内存问题报告
MEMORY HELP	显示有关不同子命令的有用文本
MEMORY MALLOC-STATS	显示分配器内部统计信息
MEMORY PURGE	要求分配器释放内存
MEMORY STATS	显示内存使用详情
MEMORY USAGE key [SAMPLES count]	估计一个key的内存占用
MGET key [key ...]	获取所有给定键的值
MIGRATE host port key| destination-db timeout [COPY] [REPLACE] [AUTH password] [AUTH2 username password] [KEYS key [key ...]]	以原子方式将键从 Redis 实例传输到另一个实例
MONITOR	实时监听服务器收到的所有请求
MOVE key db	移动一个键到另一个数据库
MSET key value [key value ...]	将多个键设置为多个值
MSETNX key value [key value ...]	仅当不存在任何键时才将多个键设置为多个值
MULTI	标记一个事务块的开始
OBJECT ENCODING key	检查 Redis 对象的内部编码
OBJECT FREQ key	获取Redis对象的对数访问频率计数器
OBJECT IDLETIME key	获取自上次访问 Redis 对象以来的时间
OBJECT REFCOUNT key	获取键值的引用次数
OBJECT HELP	显示有关不同子命令的有用文本
PERSIST key	从键中删除过期时间
PEXPIRE key milliseconds [NX|XX|GT|LT]	以毫秒为单位设置键的生存时间

PEXPIREAT key milliseconds-timestamp [NX|XX|GT|LT]	将键的到期时间设置为以毫秒为单位指定的 UNIX 时间戳
PEXPIRETIME key	以毫秒为单位获取键的到期 Unix 时间戳
PFADD key [element [element ...]]	将指定的元素添加到指定的 HyperLogLog.
PFCOUNT key [key ...]	返回 HyperLogLog 在 key(s) 处观察到的集合的近似基数
PFMERGE destkey sourcekey [sourcekey ...]	将 N 个不同的 HyperLogLog 合并为一个
PING [message]	Ping 服务器
PSETEX key milliseconds value	以毫秒为单位设置键值和过期时间
PSUBSCRIBE pattern [pattern ...]	侦听发布到与给定模式匹配的频道的消息
PUBSUB CHANNELS [pattern]	列出活动频道
PUBSUB NUMPAT	获取独特模式模式订阅的计数
PUBSUB NUMSUB [channel [channel ...]]	获取频道的订阅者数量
PUBSUB HELP	显示有用的文字 ab输出不同的子命令
PTTL key	以毫秒为单位获取键的生存时间
PUBLISH channel message	向频道发布消息
PUNSUBSCRIBE [pattern [pattern ...]]	停止收听发布到与给定模式匹配的频道的消息
QUIT	关闭连接
RANDOMKEY	从键空间返回一个随机键
READONLY	启用对集群副本节点连接的读取查询
READWRITE	禁用对集群副本节点连接的读取查询
RENAME key newkey	重命名一个键
RENAMENX key newkey	重命名键，仅当新键不存在时
RESET	重置连接
RESTORE key ttl serialized-value [REPLACE] [ABSTTL] [IDLETIME seconds] [FREQ frequency]	使用提供的序列化值创建一个键，之前使用 DUMP 获得

ROLE	返回实例在复制上下文中的角色
RPOP key [count]	删除并获取列表中的最后一个元素
RPOPLPUSH source destination	删除列表中的最后一个元素，将其添加到另一个列表中并返回它
LMOVE source destination LEFT|RIGHT LEFT|RIGHT	从列表中弹出一个元素，将其推送到另一个列表并返回它
RPUSH key element [element ...]	将一个或多个元素附加到列表中
RPUSHX key element [element ...]	仅当列表存在时才将元素附加到列表中
SADD key member [member ...]	将一个或多个成员添加到集合中
SAVE	将数据集同步保存到磁盘
SCARD key	获取集合中的成员数量
SCRIPT DEBUG YES|SYNC|NO	为执行的脚本设置调试模式
SCRIPT EXISTS sha1 [sha1 ...]	检查脚本缓存中是否存在脚本
SCRIPT FLUSH [ASYNC|SYNC]	从脚本缓存中删除所有脚本
SCRIPT KILL	终止当前正在执行的脚本
SCRIPT LOAD script	将指定的 Lua 脚本加载到脚本缓存中
SDIFF key [key ...]	减去多组
SDIFFSTORE destination key [key ...]	减去多个集合并将结果集合存储在一个键中
SELECT index	更改当前连接选择的数据库
SET key value [EX seconds|PX milliseconds|EXAT timestamp|PXAT milliseconds-timestamp|KEEPTTL] [NX|XX] [GET]	设置一个键的字符串值
SETBIT key offset value	设置或清除存储在 key
SETEX key seconds value	设置一个键的值和过期时间
SETNX key value	设置键的值，仅当键不存在时
SETRANGE key offset value	从指定的偏移量开始覆盖键处的部分字符串
SHUTDOWN [NOSAVE|SAVE]	将数据集同步保存到磁盘，然后关闭服务器

SINTER key [key ...]	返回由所有给定集合的交集产生的集合的成员
SINTERCARD numkeys key [key ...] [LIMIT limit]	将多个集合相交并返回结果的基数
SINTERSTORE destination key [key ...]	将多个集合相交并将结果集存储在一个键中
SISMEMBER key member [member ...]	返回每个成员是否为存储在key处的集合的成员
REPLICAOF host port	使服务器成为另一个实例的副本，或将其提升为主服务器
SLOWLOG GET [count]	获取慢日志的条目
SLOWLOG LEN	获取慢日志的长度
SLOWLOG RESET	清除慢日志中的所有条目
SLOWLOG HELP	显示有关不同子命令的有用文本
SMEMBERS key	集齐所有成员
SMOVE source destination member	将成员从一组移动到另一组
SORT key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]] [ASC|DESC] [ALPHA] [STORE destination]	对列表、集合或有序集合中的元素进行排序
SORT_RO key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]] [ASC|DESC] [ALPHA]	对列表、集合或有序集合中的元素进行排序， SORT 的只读变体
SPOP key [count]	从集合中删除并返回一个或多个随机成员
SRANDMEMBER key [count]	从集合中获取一个或多个随机成员
SREM key member [member ...]	从集合中删除一个或多个成员
STRLEN key	获取存储在键中的值的长度
SUBSCRIBE channel [channel ...]	收听发布到给定频道的消息
SUNION key [key ...]	返回所有给定集合的并集所产生的集合的成员
SUNIONSTORE destination key [key ...]	返回所有给定集合的并集并存储到指定集合中
SWAPDB index1 index2	交换两个Redis数据库

SYNC	内部命令，从主站发起复制流
PSYNC replicationid offset	内部命令，从主站发起复制流
TIME	返回当前服务器时间
TOUCH key [key ...]	更改键的最后访问时间，返回指定的现有键的数量
TTL key	在几秒钟内获得一把钥匙的生存时间
TYPE key	确定存储在 key 的类型
UNSUBSCRIBE [channel [channel ...]]	停止收听发布到给定频道的消息
UNLINK key [key ...]	在另一个线程中异步删除一个键，否则它就像 DEL 一样，但非阻塞
UNWATCH	忘记所有观看过的钥匙吧
WAIT numreplicas timeout	等待同步复制当前连接上下文中发送的所有写命令
WATCH key [key ...]	观察给定的键以确定 MULTI/EXEC 块的执行
ZADD key [NX|XX] [GT|LT] [CH] [INCR] score member [score member ...]	将一个或多个成员添加到有序集中，或者如果它已经存在则更新其分数
ZCARD key	获取有序集合中的成员数量
ZCOUNT key min max	用给定值内的分数计算有序集中的成员
ZDIFF numkeys key [key ...] [WITHSCORES]	减去多个有序集
ZDIFFSTORE destination numkeys key [key ...]	减去多个有序集并将结果有序集存储在一个新键中
ZINCRBY key increment member	在有序集中增加成员的分数
ZINTERCARD numkeys key [key ...] [LIMIT limit]	将多个有序集合相交并返回结果的基数
ZINTERSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]	将多个有序集相交并将结果有序集存储在一个新键中
ZLEXCOUNT key min max	计算给定字典范围内有序集中的成员数量
ZPOPMAX key [count]	删除并返回有序集中得分最高的成员
ZPOPMIN key [count]	删除并返回有序集中得分最低的成员

ZMPOP numkeys key [key ...] MIN|MAX [COUNT count]	删除并返回具有有序集中分数的成员
ZRANDMEMBER key [count [WITHSCORES]]	从有序集合中获取一个或多个随机元素
ZRANGESTORE dst src min max [BYSCORE|BYLEX] [REV] [LIMIT offset count]	将有序集合中的一系列成员存储到另一个键中
ZRANGE key min max [BYSCORE|BYLEX] [REV] [LIMIT offset count] [WITHSCORES]	返回有序集合中的一系列成员
ZRANGEBYLEX key min max [LIMIT offset count]	返回成员范围rs 在一个有序的集合中，按字典序排列
ZREVRANGEBYLEX key max min [LIMIT offset count]	返回有序集中的成员范围，按字典序范围，从高到低的字符串排序
ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]	按分数返回有序集中的一系列成员
ZRANK key member	确定有序集合中成员的索引
ZREM key member [member ...]	从有序集中删除一个或多个成员
ZREMRANGEBYLEX key min max	删除给定字典范围之间有序集中的所有成员
ZREMRANGEBYRANK key start stop	删除给定索引内有序集中的所有成员
ZREMRANGEBYSCORE key min max	删除给定分数内有序集中的所有成员
ZREVRANGE key start stop [WITHSCORES]	按索引返回有序集中的一系列成员，分数从高到低排序
ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count]	按分数返回有序集中的一系列成员，分数从高到低排序
ZREVRANK key member	确定一个有序集合中某个成员的索引，分数从高到低排序
ZSCORE key member	在有序集中获取与给定成员关联的分数
ZMSCORE key member [member ...]	获取与有序集中的给定成员相关联的分数
ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]	添加多个有序集并将结果有序集存储在一个新键中
SCAN cursor [MATCH pattern] [COUNT count] [TYPE type]	增量迭代键空间
SSCAN key cursor [MATCH pattern] [COUNT count]	增量迭代 Set 元素

HSCAN key cursor [MATCH pattern] [COUNT count]	增量迭代哈希字段和关联值
ZSCAN key cursor [MATCH pattern] [COUNT count]	增量迭代已排序的集合元素和相关分数
XINFO CONSUMERS key groupname	列出消费者组中的消费者
XINFO GROUPS key	列出流的消费者组
XINFO STREAM key [FULL [COUNT count]]	获取有关流的信息
XINFO HELP	显示有关不同子命令的有用文本
XADD key [NOMKSTREAM] [MAXLEN|MINID [=|~] threshold [LIMIT count]] *|ID field value [field value ...]	将新条目附加到流中
XTRIM key MAXLEN|MINID [=|~] threshold [LIMIT count]	将流修剪到（大约如果 '~' 被传递）特定大小
XDEL key ID [ID ...]	从流中删除指定的条目，返回实际删除的项目数，如果某些 ID 不存在，则可能与传递的 ID 数不同
XRANGE key start end [COUNT count]	返回流中的一系列元素，其 ID 与指定的 ID 间隔相匹配
XREVRANGE key end start [COUNT count]	与 XRANGE 相比，以相反的顺序（从较大到较小的 ID）返回流中的一系列元素，其中 ID 与指定的 ID 间隔匹配
XLEN key	返回流中的条目数
XREAD [COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID ...]	返回多个流中从未见过的元素，其 ID 大于调用者为每个流报告的 ID，可以屏蔽
XGROUP CREATE key groupname id|$ [MKSTREAM]	创建一个消费者组
XGROUP CREATECONSUMER key groupname consumername	在消费者组中创建消费者
XGROUP DELCONSUMER key groupname consumername	从消费者组中删除消费者
XGROUP DESTROY key groupname	销毁一个消费组
XGROUP SETID key groupname id|$	将消费者组设置为任意最后交付的 ID 值
XGROUP HELP	显示有关不同子命令的有用文本
XREADGROUP GROUP group consumer [COUNT count] [BLOCK milliseconds] [NOACK] STREAMS key [key ...] ID [ID ...]	使用消费者组从流中返回新条目，或访问给定消费者的待处理条目的历史记录，可以屏蔽

XACK key group ID [ID ...]	将待处理消息标记为正确处理，有效地将其从消费者组的待处理条目列表中删除，该命令的返回值是成功确认的消息数，即我们在 PEL 中实际能够解析的 ID
XCLAIM key group consumer min-idle-time ID [ID ...] [IDLE ms] [TIME ms-unix-time] [RETRYCOUNT count] [FORCE] [JUSTID]	在流消费者组的上下文中，此命令更改挂起消息的所有权，以便新的所有者是作为命令参数指定的消费者
XAUTOCLAIM key group consumer min-idle-time start [COUNT count] [JUSTID]	在流消费者组的上下文中，此命令自动更改挂起消息的所有权，以便新的所有者是作为命令参数指定的消费者
XPENDING key group [[IDLE min-idle-time] start end count [consumer]]	从流消费者组待处理条目列表中返回信息和条目，即获取但从未确认的消息

全国统一服务热线

4008-555-800

金蝶天燕云计算股份有限公司(简称“金蝶天燕云”)成立于2000年，前身为“金蝶中间件公司”，是金蝶集团旗下新一代软件基础云平台服务商，云计算国家标准制定企业，国家信创产业核心软件企业。金蝶天燕是国家863重点研发计划与核高基重大专项承接企业，也是“两网一站四库十二金”国家重点工程的基础平台提供商，产品广泛应用于政府、军工、金融、能源等关键行业，累计服务客户总数超过10万家。

Apusic 金蝶天燕

云计算国家标准制定企业

金蝶集团旗下基础软件企业

信息技术应用创新核心企业

官网：www.apusic.com