Redis 八股

MLE 算法指北

2025 年 2 月 23 日

Redis 是一个开源的内存数据结构存储系统，支持多种数据类型，包括字符串、哈希、列表、集合、有序集合等。它可以作为数据库、缓存、消息中间件等。

Redis 的核心原理基于内存管理，所有数据都存储在内存中，提供了高速的数据存取能力。Redis 使用单线程模型，虽然是单线程，但由于其 I/O 多路复用机制，能够有效地处理大量并发请求。

Redis 支持持久化，能够将内存数据保存到磁盘中，确保数据的可靠性。支持两种持久化机制：RDB 快照和 AOF 日志。

1 Redis 架构及原理

1.1 Redis 整体架构

Redis 采用 **C/S（Client-Server）架构 **，其主要组成部分包括：

- **客户端（Client）**：向 Redis 服务器发送命令，获取或存储数据。
- **服务器（Server）**：负责处理客户端请求，执行命令，并返回结果。
- **存储引擎（Storage Engine）**：Redis 主要基于内存存储，并提供 RDB 和 AOF 持久化机制。
- **网络处理（Networking）**：基于 **I/O 多路复用 ** 技术，实现高并发处理能力。

1.2 Redis 单线程架构

1.2.1 为什么 Redis 采用单线程？

Redis 是一个单线程的数据库，即它使用单个线程处理所有客户端请求。Redis 选择单线程架构的主要原因如下：

- **内存操作快**：Redis 主要执行内存操作，避免了磁盘 I/O 瓶颈，CPU 通常不是性能瓶颈。
- **避免多线程的锁竞争**：单线程架构避免了加锁带来的性能开销，提高执行效率。
- **I/O 多路复用**：Redis 采用 I/O 多路复用机制（如 `epoll`），同时处理多个请求，提高并发性能。

Redis 事件处理模型 Redis 的事件处理模型包括：

- **文件事件（File Event）**：用于处理网络 I/O，监听客户端请求。
- **时间事件（Time Event）**：用于执行定时任务，如删除过期键、AOF 持久化。

文件事件 Redis 采用基于 **Reactor 模型**的 I/O 多路复用技术：

1. Redis 监听多个客户端的 socket 连接，并注册读写事件。
2. 当客户端请求到达时，Redis 通过 `epoll` 或 `select` 监听事件。
3. Redis 通过事件循环依次处理每个请求（读取数据、执行命令、返回响应）。

核心代码（简化版）：

```
while (!server.shutdown) {
    int numevents = aeProcessEvents(server.el, AE_ALL_EVENTS);
    if (numevents > 0) {
        processCommand();
    }
}
```

- `aeProcessEvents()` 处理所有 I/O 事件。- `processCommand()` 解析并执行 Redis 命令。

时间事件 Redis 通过 **最小堆（min-heap）** 维护时间事件，每次循环只检查最早的定时任务，提高执行效率。

- 定期删除过期键
- AOF 持久化
- 主从同步检查

1.3 Redis 单线程的 I/O 多路复用

Redis 能在单线程下仍然支持高并发，主要依赖 **I/O 多路复用 ** 技术。I/O 多路复用可以让一个线程监听多个 I/O 事件，避免多个线程同时阻塞等待 I/O。Redis 主要使用 epoll（Linux）或 select（Mac）处理多个客户端请求。

1.3.1 I/O 方式对比

方式	机制	适用场景
阻塞 I/O（Blocking I/O）	单个 I/O 操作会阻塞进程	低并发
非阻塞 I/O（Non-blocking I/O）	进程不停轮询 I/O 状态	高 CPU 消耗
I/O 多路复用（Multiplexing）	通过 `epoll/select` 监听多个 I/O 事件	高并发，低 CPU

表 1: 不同 I/O 方式对比

1.3.2 `epoll`

- `epoll` 是 Linux 的高效 I/O 复用方式，比 `select` 和 `poll` 速度更快。
- Redis 通过 `epoll` 同时监听多个 socket 连接，在事件发生时一次性处理，避免轮询带来的 CPU 开销。

1.3.3 Redis 单线程的优势与劣势

优势

- **避免线程切换开销**：多线程会有上下文切换，而 Redis 单线程避免了这个问题。
- **避免锁竞争**：多线程需要加锁，而 Redis 直接单线程处理，保证数据一致性。
- **I/O 多路复用**：采用 `epoll`，即使单线程也能支持高并发请求。

劣势

- **不能利用多核 CPU**：Redis 只用一个线程处理请求，无法利用多核 CPU 并行计算。
- **复杂计算任务会阻塞请求**：某些命令（如 `keys *`）执行时间过长，会阻塞所有请求。
- **内存受限**：单个 Redis 进程可能无法管理所有数据，需要分片存储。

1.4 Redis 单线程的优化方案

1.4.1 使用多个 Redis 实例

在多核 CPU 机器上，运行多个 Redis 进程，提高吞吐量：

```
redis-server --port 6379
redis-server --port 6380
```

1.4.2 使用 Redis Cluster（集群）

Redis Cluster 采用 **分片存储**，多个节点分担数据，提高性能。

1.4.3 避免大数据查询

- **不要使用 `keys *`**，改用 `SCAN` 进行分批查询，避免阻塞线程。

1.4.4 使用 Pipeline 提高吞吐量

Redis 支持 Pipeline 批量执行命令，减少网络 I/O：

```
pipe = redis_client.pipeline()
pipe.set("key1", "value1")
pipe.set("key2", "value2")
pipe.execute()
```

1.5 Redis 存储原理

Redis 主要使用 ** 字典（dict）** 结构存储数据，每个 'Key' 通过 ** 哈希表 ** 进行管理：

$$\text{Index} = \text{hash}(key) \mod \text{size}$$

Redis 使用 ** 渐进式 rehash** 技术优化哈希表扩展过程，以减少一次性 rehash 造成的性能开销。

1.6 Redis 持久化机制

- **RDB (Redis Database)**：周期性地将数据快照保存到磁盘，适合备份，但可能会丢失最近的修改。
- **AOF（Append Only File)**：记录所有写操作，提供更高的数据安全性，但磁盘 I/O 频率较高。

1.7 Redis 高可用架构

- **主从复制（Master-Slave)**：通过 'replicaof' 命令同步数据，实现高可用性。
- **Sentinel 哨兵模式**：提供故障检测和自动主从切换，保证系统稳定性。
- **Redis Cluster**：采用数据分片技术，实现分布式存储和高可用架构。

1.8 Redis 特点

- **高性能**: Redis 是基于内存的数据库，读写速度极快。
- **持久化**: 支持 RDB（快照）和 AOF（日志）两种持久化方式，确保数据安全。
- **数据结构丰富**: Redis 支持多种数据类型，如字符串、哈希、列表、集合、有序集合等。
- **原子性操作**: Redis 提供事务支持，保证数据一致性。
- **分布式支持**: 通过 Redis Cluster 进行数据分片，实现高可用和负载均衡。

2 Redis 实现分布式服务

Redis 可以在多个分布式服务中扮演重要角色，以下是几种常见的 Redis 分布式应用场景：

2.1 分布式锁

Redis 可以通过 `SETNX` 命令实现分布式锁。具体流程为：客户端使用 `SETNX` 命令设置一个唯一键值对，如果设置成功则获得锁，执行完任务后删除锁。

示例代码：

```
import redis
import time

def acquire_lock(redis_client, lock_key, timeout=10):
    lock_value = str(time.time())  # 锁的唯一标识
    if redis_client.setnx(lock_key, lock_value):  # 如果 lock_key 不存在，则设置成功
        redis_client.expire(lock_key, timeout)  # 设置锁的过期时间
        return True
    return False

def release_lock(redis_client, lock_key, lock_value):
    if redis_client.get(lock_key) == lock_value:
        redis_client.delete(lock_key)

# 示例使用
r = redis.StrictRedis(host='localhost', port=6379, db=0)
if acquire_lock(r, 'my_lock_key'):
    try:
        # 执行关键操作
        print("Lock acquired, performing task...")
    finally:
        release_lock(r, 'my_lock_key', str(time.time()))
```

2.2 分布式缓存

Redis 可以用作分布式缓存层，结合 Redis 集群可以存储大规模数据。常用命令如 `SET` 和 `GET`，通过设置过期时间，避免缓存击穿。

示例代码:

```
def cache_data(redis_client, key, value, expiration=3600):
    redis_client.setex(key, expiration, value)  # 设置带过期时间的缓存

def get_cached_data(redis_client, key):
    return redis_client.get(key)
```

2.3 分布式队列

Redis 提供的 `LPUSH` 和 `BRPOP` 命令，可以用于实现生产者-消费者模式，完成分布式队列的功能。

示例代码:

```
def producer(redis_client, queue_key, task_data):
    redis_client.lpush(queue_key, task_data)  # 向队列头插入任务

def consumer(redis_client, queue_key):
    task_data = redis_client.brpop(queue_key)  # 阻塞方式获取队列任务
    return task_data
```

2.4 分布式计数器

Redis 提供的 `INCR` 命令能够原子性地实现自增计数器，适用于高并发场景。

示例代码:

```
def increment_counter(redis_client, counter_key):
    return redis_client.incr(counter_key)  # 自增计数器
```

3 总结

Redis 作为一种高效的内存数据库，广泛应用于分布式服务中，尤其在实现分布式锁、缓存、队列和计数器等场景下有着重要作用。通过合理的使用 Redis 的各种功能，可以大大提高系统的性能和可靠性。在分布式环境下，Redis 的集群、主从复制等机制确保了高可用性和数据的一致性。