

Redis通用命令（三）（38~43）

一、基于优先级队列的过期任务处理（深入展开）

1. 核心思路（Why）

在 Redis 中，过期 key 的管理本质上是一个“按时间排序的任务调度问题”。最直观的做法是：

每个 key 设置一个过期时间，到点就删除。

但问题在于：

- Redis 可能同时维护 **百万级 key**
- 不可能在每一毫秒遍历所有 key 去判断是否过期

因此引入了一个“只关心最早到期者”的思想，这正是**优先级队列（最小堆）**最擅长的场景。

队列中只维护“设置了过期时间的 key”，并按过期时间升序排列

2. 数据结构层面的含义（What）

优先级队列中的每一个元素，逻辑上至少包含：

代码块

```
1  {  
2      key 指针  
3      过期时间（绝对时间戳）  
4  }
```

- 排序依据：过期时间
- 队首元素：**最早会过期的 key**

这意味着：

- 如果队首都还没过期
- 那队列中**所有 key 都不可能过期**

3. 触发机制（When）

Redis 并不是“专门开一个线程盯着时间走”，而是通过事件驱动触发检查，主要来源包括：

1. 定期任务 (serverCron)
2. 客户端访问 key (惰性检查)
3. 事件循环的空闲时间片

当触发过期检查时：

- Redis 只看优先级队列的队首
- 对比：
 - 当前时间
 - key 的过期时间

判断是否过期

4. 处理流程 (How)

一个典型的逻辑过程是：

代码块

```
1 while (队列非空) {
2     取队首 key
3     if (当前时间 >= 过期时间) {
4         删除 key
5         从队列中移除
6     } else {
7         break; // 队首未过期，后面一定也没过期
8     }
9 }
```

🔴 这里的 `break` 非常关键，它直接保证了：

不会无意义地扫描后续元素

5. 优化细节 (Why Redis 能扛住高并发)

你提到的优化点非常重要，我逐条展开：

① 限制单次检查数量

- 每次最多检查 N 个 key
- 防止：

- 大量 key 同时过期
- 主线程被“清空任务”阻塞

这是典型的“**渐进式处理**”思想。

② 新 key 插入策略

- 新增带过期时间的 key：
 - 直接按过期时间插入队列
- 时间复杂度：
 - $O(\log n)$

这是完全可以接受的，因为：

- 设置过期时间本身不是高频操作
 - 远比全量扫描划算
-

③ 队首未过期即停止

这是整个设计的**灵魂点**：

Redis 从来不做“无意义判断”

如果最早的都没过期，其余的就一定安全。

6. 核心优势总结（升维）

这一设计的本质优势是：

- 将“N 个 key 的过期判断”
- 降维为“1 个 key 的时间比较”

也正因为如此，Redis 才能做到：

在高并发 + 大 key 数量的情况下，过期机制几乎不成为性能瓶颈

二、基于时间轮的过期任务处理（深入展开）

1. 时间轮的本质思想

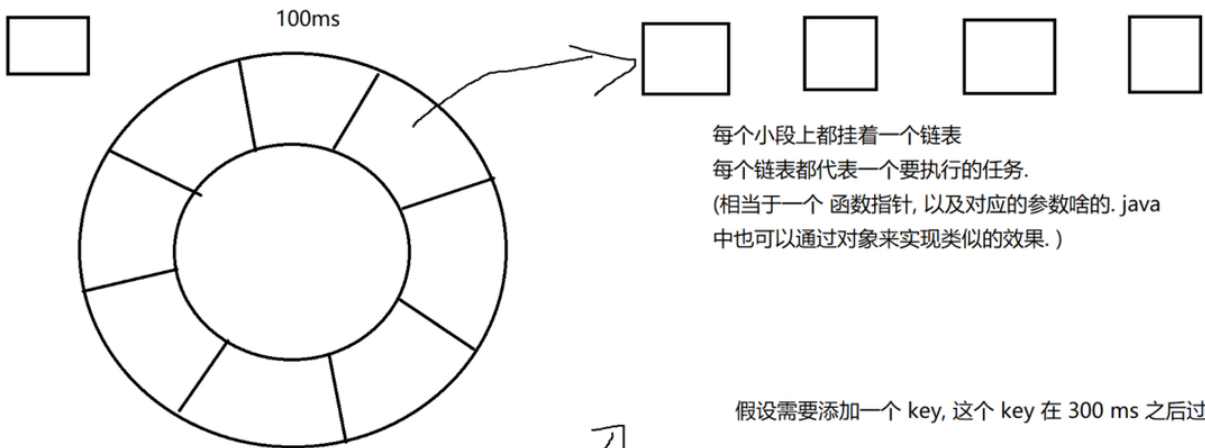
时间轮 (Timing Wheel) 解决的是另一类问题:

“我不需要精确到毫秒, 只要近似正确即可”

它通过 **空间换时间** 的方式, 把时间离散化。

2. 结构拆解 (What)

把时间划分成很多小段~~ (划分的粒度, 看实际需求)



每个小段上都挂着一个链表
每个链表都代表一个要执行的任务。
(相当于一个 函数指针, 以及对应的参数啥的. java
中也可以通过对象来实现类似的效果.)

假设需要添加一个 key, 这个 key 在 300 ms 之后过期

假设指定的过期时间特别特别长

3000ms

此时这个指针, 就会每隔固定的间隔 (此处是约定 100ms)

每次走到一个格子, 就会把这个格子上链表的任务尝试执行一下~~

对于时间轮来说, 每个格子是多少时间, 一共多少个格子, 都是需要根据实际场景, 灵活调配的~~

假设:

- 时间轮有 N 个槽
- 每个槽代表 Δt (如 100ms)

那么整个时间轮能表示的时间范围是:

代码块

```
1 N * Δt
```

每个槽内部是:

- 链表 / 数组
- 存放 “应该在这个时间段触发的任务”

3. 指针驱动模型 (How)

- 系统维护一个“当前指针”
 - 每 Δt :
 - 指针向前移动一格
 - 执行该槽中的所有任务
 - ✚ 执行时通常还会再次校验：
 - 是否真的到期 (防止时间误差)
-

4. 任务添加逻辑 (Example)

你给的例子非常好，我补充完整过程：

- 当前指针：slot 0
- 时间精度：100ms
- key 需要在 300ms 后过期

则：

代码块

```
1  目标槽位 = (当前指针 + 3) % 槽数量
```

然后把 key 放入该槽的任务列表。

5. 时间轮的优势

- 插入：O(1)
 - 触发：O(1)
 - 不需要排序
 - 非常适合：
 - 网络超时
 - 心跳检测
 - 延迟队列
-

6. 为什么 Redis 没用时间轮？

这是一个非常好的“反问点”：

Redis 的场景特点：

- key 数量极大
- TTL 分布非常散
- 对时间精度要求相对较高
- 更关注：
 - 内存占用
 - 实现复杂度
 - 稳定性

时间轮的劣势：

- 时间精度受限
- 槽位数量固定
- 超长 TTL 需要多轮处理
- 实现复杂度高

👉 所以 Redis 选择了：

惰性删除 + 定期抽样
而不是时间轮

三、Redis 核心基础与数据类型（深入展开）

1. 事件循环（Redis 的“心脏”）

Redis 是典型的：

单线程 + 事件驱动模型

事件循环负责：

- 网络 IO
- 命令执行
- 过期 key 处理
- 定期维护任务

优势非常明确：

- 无锁
 - 无上下文切换
 - 逻辑简单
 - 延迟可预测
-

2. 基础命令的真实语义

keys

- 会遍历整个 keyspace
- $O(n)$
- 生产环境禁用

exists

- 字典查找
- $O(1)$
- 不关心 value 内容

del

- 删除 key 的 entry
- 释放 value 内存
- 大对象可能异步释放

expire

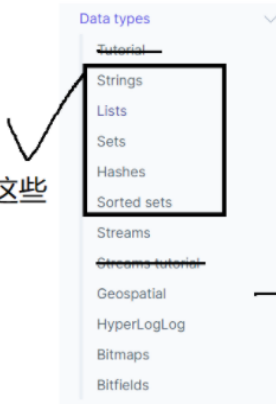
- 只做一件事：
 - 记录过期时间
 - 加入过期管理结构
-

3. 数据类型与编码（Why 这么复杂）

Redis 的核心理念是：

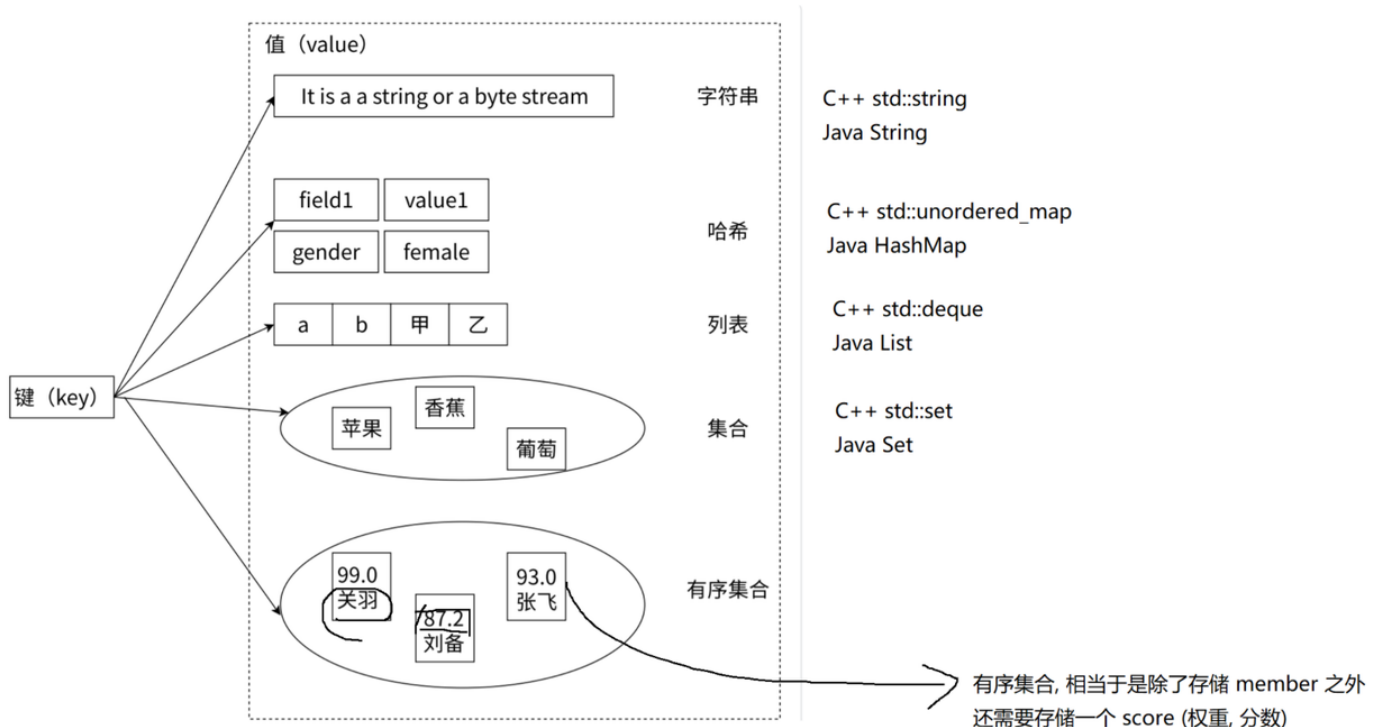
| 同一种逻辑类型 \neq 只有一种物理实现

非常通用的
数据结构
绝大部分都是用这些



当前版本的 redis 支持 10 个数据类型~~

这几个类型都是针对特殊场景下的特殊数据类型/数据结构



Redis底层在实现上述数据结构的时候, 会在源码层面, 针对上述实现进行特定的优化, 来达到节省时间/节省空间效果.

内部的具体实现的数据结构, 还会有变数. 编码方式

redis承诺, 现在我这有个hash表, 你进行查询, 插入, 删除操作, 都保证O(1)

但是, 这个背后的实现, 不一定就是一个标准的hash表.

可能再特定场景下, 使用别的数据结构实现.但是仍然保证时间复杂度符合承诺!!!

string → SDS

- 记录长度
- O(1) 获取 size

- 防止缓冲区溢出

list / hash / zset 的编码切换

这是 Redis 极其重要的一点：

- 小数据 → 紧凑结构 (ziplist)
- 大数据 → 高效结构 (dict / skiplist)

目的只有一个：

用最小的内存，满足当前规模的性能需求

数据结构: redis 承诺给你的, 也可以理解成数据类型。
编码方式: redis 内部底层的实现。

同一个数据类型, 背后可能的编码实现方式是不同的~~ 会根据特定场景优化~~

数据类型	内部编码
string	raw
	int
	embstr
hash	hashtable
	ziplist
list	linkedlist
	ziplist
	hashtable
set	intset
	skiplist
zset	skiplist
	ziplist

redis 会自动适应。
程序员在使用 redis 的时候一般感知不到~~

最基本的字符串。(底层就是持有一个 char 数组(C++), 或者 byte 数组 (Java))

C++ 里的 char 是 1 字节的, 等价于 Java 的 byte

而 Java 的 char 是两个字节的~~

redis 通常也可以用来实现一些“计数”这样的功能。
当 value 就是一个整数的时候
此时可能 redis 会直接使用 int 来保存~~

针对短字符串进行的特殊优化。

在哈希表里面元素比较少的时候
可能就优化成 ziplist 了。

压缩列表, 能够节省空间~~

为啥要压缩??

redis 上有很多很多 key.

可能是某些 key 的 value 是 hash.

此时, 如果 key 特别多, 对应的 hash 也特别多, 但是每个 hash 又不大的情况下, 就尽量去压缩. 压缩之后就可以让整体占用的内存更小了。

最基本的哈希表。
redis 内部的
哈希表的实现~~

在 Java 标准库中有一个东西就叫做
HashTable
(Java 标准库中的 哈希表 的实现)

虽然这里的实现方式可能不太一样
但是整体思想是和之前学过的一致的~~

四、核心设计思想 (升维总结)

1. 过期处理的设计哲学

Redis 不追求：

“key 一过期就立刻删除”

而追求：

系统整体吞吐量最大化

2. 数据结构是“活的”

- 会变
- 会升级
- 会退化

而不是一次选型定终身。

3. 单线程 ≠ 性能低

恰恰相反：

- 单线程让一切变得可控
 - 延迟稳定
 - 更适合内存数据库
-

总结

Redis 的所有设计，都不是为了“理论完美”，而是为了“在真实高并发场景下，稳定、快速、可预测地运行”