Redis - 单线程的工作模式(44~46)

一、Redis 数据类型的底层编码与演进（全面展开）

1. 编码方式概览（从“逻辑类型”到“物理实现”）

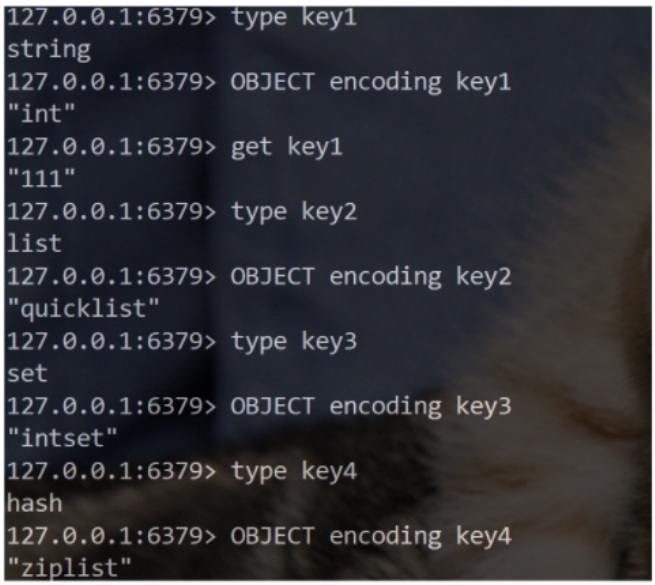
Redis 对外暴露的是 **五大逻辑数据类型**，但在内部，它真正关心的是：

> **如何用最少的内存 + 足够快的速度，表达当前数据形态**

因此，每种逻辑类型都并非只有一种底层实现，而是**多种编码并存，按需切换**。

1.1 string 的三种编码（最经典、最常考）

object encoding key
查看 key 对应的 value 的实际编码方式.



redis会自动根据当前的实际情况选择内部的编码方式.自动适应的

（1） `int` 编码

- 使用场景：value 是**可被解析为 long long 的整数**
- 特点：
  - 不使用 SDS
  - 直接将整数存储在 redisObject 中
- 优点：
  - **极致省内存**
  - `incr / decr` 操作非常快
- 编码切换：
  - 一旦写入非整数值 → 立即转为 `raw`

📌 设计思想：

> **能不用字符串，就不用字符串**

（2） `embstr` 编码

- 使用场景：字符串长度 ≤ 44 字节
- 内部结构特点：
  - redisObject + SDS **一次性连续分配**
- 优点：

    - 少一次内存分配
    - 少一次指针跳转
    - 非常 cache-friendly
- 缺点：
    - **不可修改**
    - 一旦发生 append / set 新值 → 转为 `raw`

📌 这是 Redis 非常“工程化”的优化点：

> **为小对象量身定制编码**

（3） `raw` 编码

- 使用场景：
    - 字符串较长
    - 或由 embstr/int 转换而来
- 底层：
    - SDS + 单独内存分配
- 优点：
    - 支持动态扩容
    - 功能最完整

📌 `raw` 是 string 的“最终形态”

1.2 hash 的编码选择

`ziplist`

- 适用条件（必须同时满足）：
    - field-value 数量少
    - 每个 field / value 较短
- 特点：
    - 连续内存
    - 无指针
    - 极省内存
- 缺点：

- 查找 O(n)
- 插入/删除需要整体移动内存

`hashtable`

- 适用场景：
  - hash 规模变大
- 特点：
  - dict 实现
  - 查找 O(1)
- 编码切换：
  - 一旦超过阈值 → ziplist → hashtable
  - **不可逆**

📌 Redis 再次体现：

> **先省内存，再保性能**

1.3 list 的编码演进（考点密集区）

早期：

- `ziplist` + `linkedlist`

Redis 3.2 之后：

- **统一使用 `quicklist`**

1.4 set 的编码

`intset`

- 元素全是整数
- 数量少
- 内部自动升级：
  - int16 → int32 → int64
- 极致省内存

`hashtable`

- 一旦插入非整数
- 或元素数量过多
- 立刻升级

1.5 zset 的编码（最复杂）

`ziplist`

- 小规模、有序数据
- score + member 连续存储

`skiplist + dict`

- skiplist：范围查询、有序遍历
- dict：member → score 快速定位
- **两套结构协同工作**

📌 这是 Redis 典型的：

> **时间效率优先设计**

---

2. 核心编码演进（quicklist 深入）

2.1 quicklist 的诞生背景

Redis 团队发现：

- linkedlist：
  - 指针太多
  - 内存浪费严重
- ziplist：
  - 插入删除成本高
  - 不适合大 list

于是 quicklist 出现。

2.2 quicklist 的结构本质

代码块

```
quicklist
  ↓
双向链表
  ↓
每个节点是一个 ziplist
```

- 插入删除：链表级别 O(1)
- 存储：ziplist 连续内存
- 兼顾：
  - 性能
  - 内存

2.3 编码动态切换机制

Redis 在以下情况下会触发编码切换：

- 元素数量超过阈值
- 单个元素大小超过限制
- 数据类型发生变化（如 set 插入非整数）

📌 重要原则：

> **只升级，不降级**

这是为了：

- 避免频繁转换
- 保证性能稳定

---

3. 面试考点与技巧（升维）

高频问法

- ziplist 为什么省内存?
- quicklist 解决了什么问题?
- zset 为什么要用 skiplist + dict?
- 编码切换是自动的吗?

回答关键点

> **不要背规则，要讲“设计权衡”**

二、Redis 单线程模型与并发安全（全面展开）

1. 单线程的真实含义（非常容易被误解）

Redis 的“单线程”指的是：

> **命令执行是单线程**

而不是：

- 所有模块都单线程

redis只使用一个线程，处理所有的命令请求.不是说一个redis服务器进程内部真的就只有一个线程.其实也有多个线程，多个线程是在处理网络I0

---

2. Redis 实际线程模型

- 主线程：
  - 命令解析
  - 命令执行
  - 内存操作
- 辅助线程：
  - 网络 IO
  - AOF 重写
  - lazy free（大对象释放）

📌 核心数据结构只被主线程操作

---

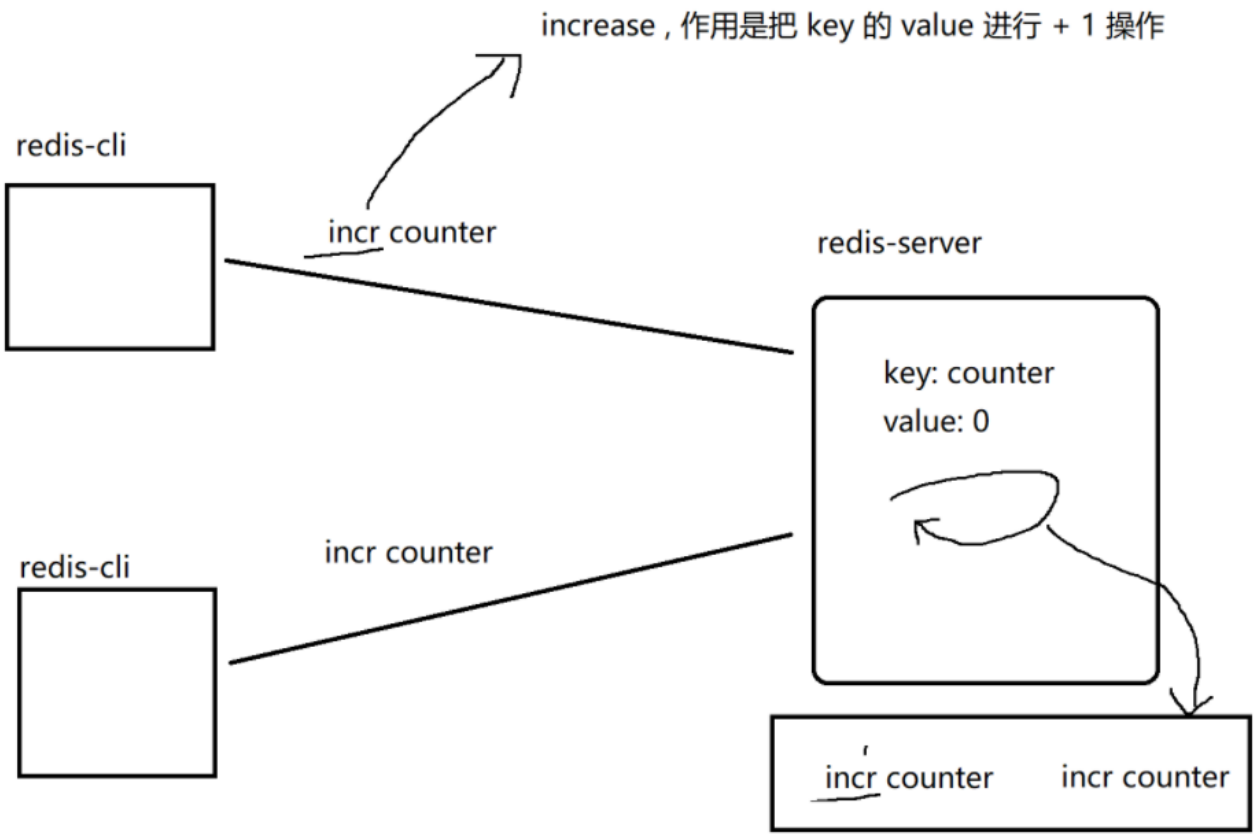
3. 命令执行流程（细化）

代码块

```
客户端发送命令
  ↓
网络 IO 接收
  ↓
```

```
请求进入队列
  ↓
主线程事件循环
  ↓
逐条执行命令
  ↓
返回结果
```

4. 并发安全的本质原因



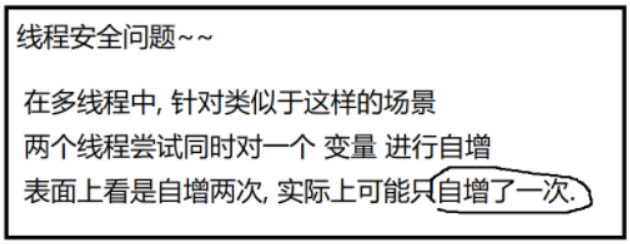
> 当前这两个客户端，也相当于"并发"的发起了上述的请求~~此时就意味着是否服务器这边也会存在类似的线程安全问题呢??
>
> 幸运的是，并不会. redis服务器实际上是单线程模型.保证了当前收到的这多个请求是串行执行的!!!
>
> 多个请求同时到达redis服务器，也是要先在队列中排队.
>
> 再等待redis服务器一个一个的取出里面的命令再执行.
>
> 微观上讲，redis服务器是串行/顺序执行这多个命令的.

- 没有共享写
- 没有并行修改
- 没有锁竞争

所以：

```
代码块
1   incr counter
```

在 Redis 中天然是线程安全的。



---

5. 单线程的代价与限制

阻塞问题

以下命令会严重阻塞：

- `keys *`
- `hgetall`
- `smembers`
- 大 value 的 `del`

redis必须要特别小心，某个操作占用时间长，就会阻塞其他命令的执行!!

解决方案

- 使用 `scan / hscan / sscan`
- 使用分页
- 控制单 key 体积

---

6. Redis 为什么仍然这么快?

- 内存存储
- 单线程无锁

- epoll / kqueue
- 高效数据结构
- 极简执行路径

📌 **单线程不是弱点，而是优势**

三、学习与实践建议（再展开）

1. 编码学习的正确方式

不要问：

> “这个类型用什么编码？”

而要问：

> “在什么数据形态下，用什么编码最合适？”

---

2. 并发安全的实践验证

你可以做这些实验：

- 多客户端同时 `incr`
- 模拟大 key 阻塞
- 对比 `keys` vs `scan`

这些实验比背书更有价值。

---

3. 生产环境经验总结

- 控制单 key 大小
- 避免全量命令
- 使用 pipeline
- 合理拆分数据结构

结尾总结

> **Redis 的强大，不在于命令多，而在于它用极其克制的设计，把性能、内存和复杂度控制在了一个极优平衡点上。**