Redis Hash 类型：设计原理、底层实现与生产实践（深度展开）（70~75）

一、Redis Hash 类型的核心概念与设计

1. 结构本质：为什么 Hash 适合存对象

（1）Hash 在 Redis 中“到底是什么”

Redis 的 Hash 本质上是：

> **一个 key → 映射到多个 field-value 的容器**

形式上是两层结构：

```
key
  ├── field1 -> value1
  ├── field2 -> value2
  └── field3 -> value3
```

这与 `string` 不同，`string` 是：

```
key -> value
```

Hash 是 **“key 中再嵌一层 key-value”**，因此也被称为**嵌套型数据结构**。

（2）为什么 Hash 特别适合“对象数据”

以用户对象为例：

代码块

```
{
  "id": 1,
  "name": "Tom",
  "age": 18,
  "score": 90
}
```

如果用 string 存：

- `user:1:name`
- `user:1:age`
- `user:1:score`

问题在于：

- key 数量暴增
- key 命名冗余
- 管理复杂

而用 Hash：

代码块

```
key = user:1
field = name / age / score
```

优势非常明显：

- **逻辑聚合**：一个对象一个 key
- **语义清晰**：天然表达“对象”
- **内存友好**：减少 key 的重复字符串

📌 **一句话总结：**

> Hash 是 Redis 中最接近“对象模型”的数据结构。

（3）Hash vs JSON string 的本质区别

很多新手会问：

> “我用 JSON 存 string 不也能存对象吗？”

可以，但差别巨大：

维度	Hash	JSON string
局部更新	支持（HSET 单字段）	不支持（要整体覆盖）
读写粒度	字段级	整体
并发安全	天然原子	需额外处理
内存结构	结构化	纯字符串

📌 **结论：**

> **需要频繁修改对象某个字段 → 用 Hash**
> **只读或整体替换 → string 也可以**

---

2. 底层编码：为什么有 ziplist 和 dict

Redis 的设计哲学非常明确：

> **小数据，省内存；大数据，保性能**

（1）ziplist（压缩列表）的设计初衷

ziplist 的核心特点：

- 连续内存
- 无额外指针
- entry 紧凑排列

非常适合：

- field 少
- value 短
- 小对象

典型场景：

- 用户基础信息
- 配置项
- 轻量级状态

⚠ 代价是：

- 查找是 O(N)
- 插入可能触发整体内存移动

（2）dict（字典）的使用场景

当 Hash 满足以下任一条件：

- field 数量变多
- 单个 field/value 变大

Redis 会自动切换为 dict：

- 底层是哈希表
- 查找、修改接近 O(1)
- 内存占用更高

📌 **这是典型的“空间换时间”。**

（3）动态切换的意义

这一点非常重要：

- **开发者不用管**
- Redis 自动决定
- 数据结构对用户透明

但你必须：

- **知道什么时候会切**
- **知道切完会发生什么**

否则就会在生产环境踩坑（后面会讲）。

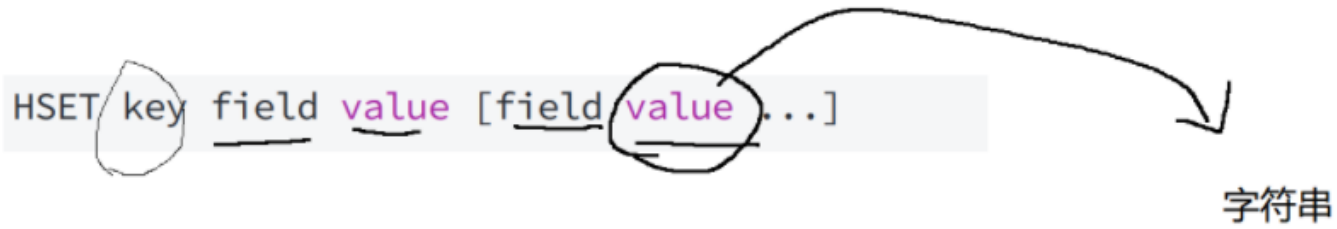
二、Hash 核心命令详解（结合底层与使用场景）

---

1. 基础增删改查：为什么这些命令这样设计

`HSET`

```
代码块
1   HSET user:1 name Tom age 18 score 90
```



返回值, 是设置成功的 键值对(field - value) 的个数

设计亮点：

- 支持一次写多个字段
- 单条命令，原子执行
- 网络 IO 次数更少

📌 **最佳实践：**

> 永远优先用批量 HSET，而不是多个单字段写。

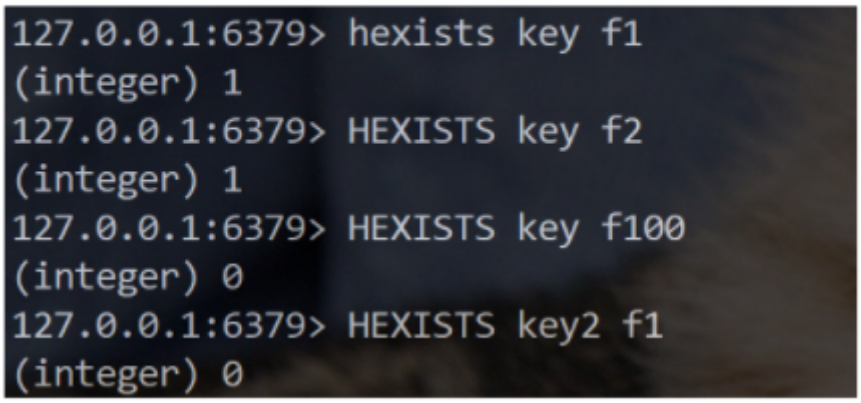
`HGET / HEXISTS`

- `HGET`：字段级读取


```
127.0.0.1:6379> hget key f100
(nil)
127.0.0.1:6379> hget key2 f1
(nil)
```

- `HEXISTS`：字段级存在性判断

```
HEXISTS key field
```



这两个命令的意义在于：

- **避免取全量**
- **避免无效读**

在大 Hash 中尤为重要。

`HDEL`

```
HDEL key field [field ...]
```

注意一个容易忽略的点：

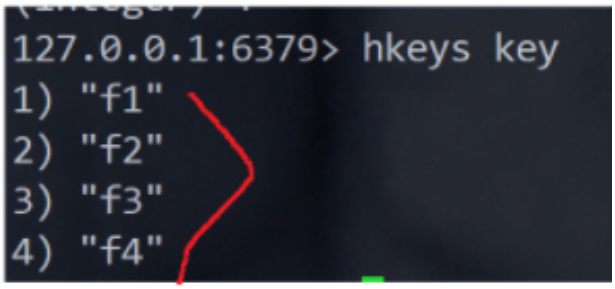
> 删除的是 **field**，不是 **key**
>
> **del** 删除的是 **key**
>
> **hdel** 删除的是 **field**

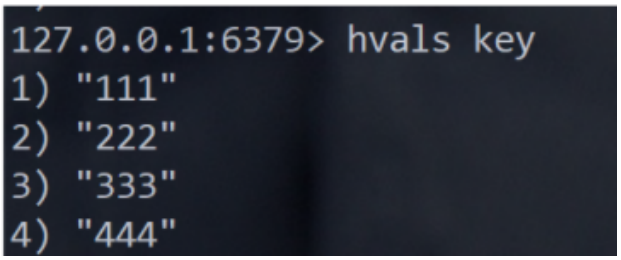
- Hash 被删空后：
  - Redis 会自动删除整个 key
- 不会留下“空壳”

---

2. 批量与全量操作：为什么它们“危险”

`HGETALL / HKEYS / HVALS`





这些命令的共同特征：

- **一次性遍历所有 field**
- 时间复杂度 O(N)
- 在单线程 Redis 中会阻塞

危险并不在于：

- “慢”

而在于：

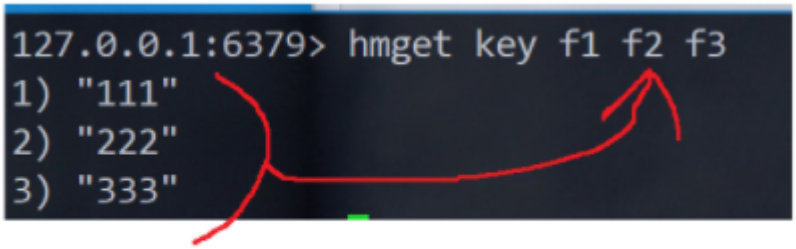
- **阻塞所有其他请求**

📌 **生产环境共识：**

> 只要 Hash 可能变大，就禁止使用这些命令。

HMGET

> 类似于之前的MGET，可以一次查询多个field
>
> HGET一次只能查一个field



注意, 多个 value 的顺序和 field 的顺序是匹配的.

`HSCAN` 的设计哲学

`HSCAN` 的本质是：

- 把一次性遍历
- 拆成多次小遍历

特点：

- 非阻塞
- 可中断
- 允许近似结果
- 敲一次命令，遍历一小部分

📌 **这和 `SCAN` 替代 `KEYS` 是同一套思想。**

> 有没有hmset,一次设置多个field和value 呢??
>
> 有但是，并不需要使用.hset 已经支持一次设置多个field和value了.

3. 统计与数值操作：Hash 的“隐形杀手锏”

`HLEN` 为什么是 O(1)

因为：

- Hash 内部维护了字段数量
- 不需要遍历

这也是 Redis 设计中：

> **“为高频统计场景预留优化点”** 的体现。

`HINCRBY / HINCRBYFLOAT`

这些命令非常重要：

- 支持字段级计数
- 原子操作
- 无并发问题

典型场景：

- 用户积分
- 商品库存
- 行为统计

📌 **相比 string 的 incr：**

> Hash 可以在一个 key 内维护多个独立计数器。

三、性能风险与优化建议（生产级重点）

1. 大 Hash 的真实风险

大 Hash 的问题不在“存不下”，而在：

- 访问模式错误
- 全量操作失控
- 单线程被拖死

真实事故常见于：

- 运维执行 `HGETALL`
- 程序误用 `HKEYS`
- 调试命令误入生产

---

2. 正确的使用原则

（1）Hash 的字段要“有限”

经验法则：

- Hash 适合：
  - **几十到几百字段**
- 超大规模：
  - 应拆 key

- 或改用其他结构

（2）明确访问路径

- 你是不是**只需要几个字段**
- 你是不是**经常全量读**

如果经常全量读：

- Hash 可能不是最优选择

（3）编码意识（高级）

虽然 Redis 会自动切换编码，但你要知道：

- ziplist → dict 是**不可逆的**
- 一旦变大，内存占用立刻上升

这在：

- 内存敏感场景
- 大规模缓存集群

非常关键。

四、核心设计思想与实践（升维总结）

1. Hash 的真正定位

> **Redis Hash = 高性能、轻量级的对象存储结构**

它不是：

- 数据库表
- ORM 对象

而是：

- **缓存层的对象表达方式**

---

2. 动态编码的设计哲学

Redis 的底层设计非常“务实”：

- 不追求理论最优

- 追求真实业务场景最优

📌 **这也是 Redis 能长期占据缓存领域的原因之一。**

---

3. 关于类比 ConcurrentHashMap

这个类比是**理解层面的类比**，不是实现层面的：

- 相同点：
  - 哈希结构
  - 高效查找
- 不同点：
  - Redis 单线程 → 原子性来自串行
  - Java 多线程 → 原子性来自锁/并发控制

---

4. 给你的一个“架构级判断标准”

以后你可以这样判断是否用 Hash：

> **“这是一个对象吗？**
> **它的字段会频繁单独更新吗？**
> **字段数量是否可控？”**

三个都是 **Yes** → Hash
否则 → 重新考虑
