Redis - hash编码方式及其应用（76~80）

一、压缩算法与 ziplist 编码原理（深度展开）

1. 压缩的本质：重新编码，而不是“省略信息”

从计算机科学角度看，**压缩的核心目标并不是删除数据，而是用“更短的符号”表达“同样的信息量”**。

（1）信息论视角

- 数据中往往存在 **冗余性**
  - 字符重复
  - 模式重复
  - 结构重复
- 压缩算法的本质：
  👉 **利用统计特征，把“高频信息”映射为“短编码”**

这正是哈夫曼编码、LZ77/LZ78、LZW 等算法的理论基础。

（2）你给的示例进一步解释

原始字符串：

```
代码块
1    abccccccddddeee
```

如果直接存储：

- 每个字符 1 字节
- 总长度 = 1 × 15 = 15 字节

如果使用**运行长度编码（RLE）思想**：

```
代码块
1    a1b1c5d5e3
```

- 通过「字符 + 次数」重新编码

- 信息没有减少，但**表达方式更紧凑**

⚠ 关键点：

> **压缩并不追求“最短字符串”，而是追求“整体最省空间 + 可高效解码”**

（3）压缩一定是“可逆”的

- 如果无法还原 → 不是压缩，而是“丢失”
- Redis 使用的 ziplist 并不是传统意义的压缩算法
  👉 而是 **“结构层面的紧凑编码”**

---

2. ziplist 编码设计：用结构换空间

ziplist 是 Redis 为小数据量场景专门设计的一种 **内存紧凑结构**。

（1）为什么 Redis 需要 ziplist?

Redis 的一个核心设计目标是：

> **在保证 O(1) 或接近 O(1) 操作复杂度的前提下，尽可能节省内存**

而传统数据结构的问题在于：

- 链表：
  - 每个节点都要 `prev / next` 指针
  - 64 位系统上指针通常 8 字节
- 哈希表：
  - bucket 数组
  - rehash 过程
  - 指针 + 元数据开销明显

👉 **当数据量很小的时候，这些“结构性开销”反而比数据本身还大**

（2）ziplist 的核心结构特点（重点）

ziplist 是：

- **一整块连续内存**
- 没有任何指针
- 所有元素顺序排列

每个 entry 内部包含：

- `prev_entry_length`：前一个元素的长度
- `encoding`：当前元素的编码方式（int / string）
- `content`：真实数据

📌 这带来几个直接好处：

1. **极低的内存碎片**
2. **没有指针开销**
3. 数据局部性好，CPU cache 命中率高

（3）为什么 ziplist 只适合“小数据”？

ziplist 的代价是：

- 插入 / 删除 **不是 O(1)**
- 中间插入可能触发：
  - 大块内存移动（memmove）
  - 连锁更新 `prev_entry_length`

👉 当元素数量一多，**时间复杂度和内存拷贝成本都会急剧上升**

所以 Redis 明确限制使用条件。

（4）自动切换规则的深层含义

你提到的两个阈值非常关键：

```
代码块
1   hash-max-ziplist-entries = 512
2   hash-max-ziplist-value   = 64
```

这背后是一个**空间-时间权衡模型**：

维度	ziplist	hashtable
内存占用	极小	较大
访问复杂度	O(n)	O(1)
插入删除	慢	快
扩容成本	无	rehash

Redis 的策略是：

> **当“结构性开销 < 数据规模收益”时，用 ziplist**
> **否则果断切换为 hashtable**

（5）频繁切换为什么是“灾难”？

从 ziplist → hashtable：

- 需要：
  - 遍历所有 entry
  - 重新 hash
  - 分配新结构
  - 释放旧内存

如果业务中：

- 数据量在阈值附近反复横跳
  👉 会导致 **频繁重编码 + 内存抖动**

✔️ 这也是你强调“合理设置阈值”的工程意义所在。

二、Redis Hash vs String 存储结构化数据（深度展开）

1. 本质差异：结构化 vs 序列化

（1）String 存对象 = “黑盒”

代码块

```
SET user:1 '{"name":"James","age":28}'
```

在 Redis 看来：

- value 只是一个字节数组
- 内部字段完全不可感知

👉 所有结构语义都在 **客户端**

（2）Hash 存对象 = “白盒”

```
代码块
1   HSET user:1 name James age 28
```

Redis 层面：

- 每个字段都是独立的 `field-value`
- 可单独操作、单独编码、单独更新

---

2. 更新效率为什么差距这么大？

String 更新一个字段的真实流程

1. GET 整个 JSON
2. 反序列化为对象
3. 修改字段
4. 重新序列化
5. SET 覆盖原值

📌 这是 **CPU + 内存 + 网络** 三重开销

Hash 更新一个字段

```
代码块
1   HSET user:1 age 29
```

- 定位 field
- 覆盖 value
- 完成

👉 **O(1)，无序列化**

---

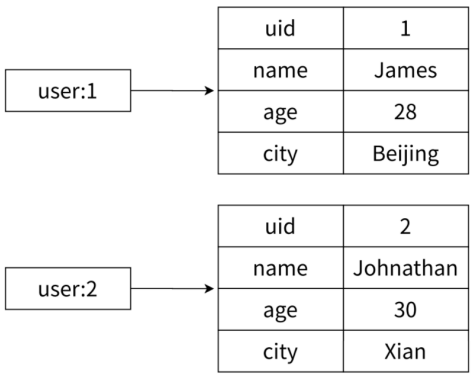
3. 内存占用对比的深层原因

String(JSON) 的隐藏成本

- 重复的字段名
- 引号、冒号、逗号
- 字符串格式

这些在 Redis 看来：

> **全是“业务无关的冗余字符”**



uid	name	age	city
1	James	28	Beijing
2	Johnathan	30	Xian

如果使用 string (Json)的格式来表示Userlnfo万一只想获取其中的某个field，或者修改某个field就需要把整个json都读出来，解析成对象，操作field，再重写转成json字符串，再写回去

如果使用hash的方式来表示Userlnfo就可以使用field表示对象的每个属性(数据表的每个列)此时就可以非常方便的修改/获取任何一个属性的值了~~

Hash 的优势

- 字段名只存一次
- 小 Hash 可使用 ziplist
- 整体结构可动态升级

👉 所以在**字段多、修改频繁**的场景下，Hash 是绝对优势。

---

4. 场景选择的工程视角

你给的总结本质是：

- **Hash：数据库思维**
- **String：消息 / 快照思维**

工程中的常见组合策略

- Redis Hash：
  👉 作为 **热数据 / 状态数据**
- MySQL / JSON：
  👉 作为 **持久化与对外接口格式**

---

5. 冗余存 uid 的“工程智慧”

这点非常重要，也非常真实。

在 Hash 中额外存：

```
代码块
1   HSET user:1 uid 1
```

目的不是“多此一举”，而是：

- 避免 key 丢失语义
- 支持 value 级别的反向引用
- 降低业务代码复杂度

👉 **这是典型的：用一点空间换开发效率**

三、高内聚 & 低耦合（结合 Redis 再深化）

1. 为什么这是“系统级原则”？

当系统规模变大时，复杂度主要来源于：

- 模块之间的依赖关系
- 修改的“连锁反应”

高内聚 + 低耦合的本质目标是：

> **把变化控制在最小范围内**

---

2. 高内聚：让一个模块“只干一件事”

Redis 中的体现

- ziplist：

- 只关心内存布局
- 不关心命令、不关心业务

- hashtable：
  - 只关心 key → value 映射

👉 **每个模块的复杂度被严格限制**

---

3. 低耦合：模块之间“通过契约交互”

Redis 的设计是：

- 命令层：
  - 不关心底层是 ziplist 还是 hashtable
- 编码层：
  - 对外暴露统一接口

所以：

- 改编码 ≠ 改命令
- 改存储 ≠ 改协议

这正是 Redis 可以不断演进编码结构的原因。

四、整体设计原则的再提炼（升维总结）

你这四点可以抽象成一句话：

> **Redis 的设计不是“为了快”，而是“在不同规模下始终保持最优平衡”**

1️⃣ 动态编码 = 自适应系统

2️⃣ Hash 优先 = 结构化优先

3️⃣ 高内聚低耦合 = 可长期演进

4️⃣ 工程冗余 = 用空间买确定性
