Redis - 单线程为什么这么快？（47~53）

一、Redis 高性能的底层原因（深度展开）

Redis 的“快”并不是单点优化的结果，而是**多层设计叠加后的系统性成果**。可以理解为：

> **每一层都在为“低延迟、高吞吐”服务**

---

1. 纯内存存储（性能的物理基础）

1️⃣ 为什么内存比磁盘快？

- **内存访问**：
  - 纳秒级（ns）
  - CPU 直接寻址
- **磁盘 I/O**：
  - 微秒 / 毫秒级（μs / ms）
  - 涉及：
    - 系统调用
    - 文件系统
    - 磁盘寻址
    - 页缓存

📌 即便是 SSD，也远慢于内存。

2️⃣ Redis 的选择与代价

Redis 把**所有数据放在内存中**，意味着：

✅ 优点：

- 读写几乎都是 CPU + 内存操作
- 延迟极低、稳定可预测

⚠️ 代价：

- 内存成本高

- 数据易丢失

因此 Redis 必须配合：

- RDB
- AOF
- 主从复制

📌 **Redis 的哲学是：**

> 先极致快，再想办法“尽量不丢数据”

3️⃣ 与传统数据库的本质差异

对比项	Redis	MySQL
存储介质	内存	磁盘
设计目标	低延迟	强一致
典型延迟	微秒级	毫秒级

---

2. 精简核心功能（架构层面的克制）

1️⃣ Redis 为何只做 KV?

Redis 从一开始就明确了定位：

> **不是通用数据库，而是高性能数据结构服务器**

因此它**刻意不支持**：

- 复杂事务
- 多表 join
- SQL 解析
- 复杂执行计划

2️⃣ 精简带来的直接收益

- 无 SQL Parser
- 无优化器
- 无执行计划缓存

- 无锁表结构

最终结果是：

> **命令路径极短，几乎“进来就执行”**

3️⃣ 工程上的意义

Redis 的命令执行路径往往是：

```
网络读取 → 查 dict → 操作内存 → 返回
```

这也是 Redis 能做到 **百万级 QPS** 的根本原因之一。

---

3. 单线程模型（性能与安全的统一）

1️⃣ Redis 单线程到底“单”在哪？

Redis 的单线程指的是：

> **核心命令执行线程是单线程**

而不是：

- 所有功能都单线程

2️⃣ Redis 的真实线程模型

- 主线程：
  - 命令解析
  - 命令执行
  - 操作核心数据结构
- 辅助线程：
  - 网络 I/O
  - AOF 重写
  - lazy free（大对象释放）

📌 **关键点**：

> 只有主线程能修改数据

3️⃣ 为什么单线程反而更快？

- 没有锁
- 没有线程切换
- 没有共享资源竞争
- CPU cache 命中率更高

📌 这使 Redis：

- 延迟更稳定
- 行为更可预测

4️⃣ 原子性的自然保障

```
代码块
1   INCR counter
```

在 Redis 中天然是安全的，因为：

- 命令不可被打断
- 不存在并发写

这也是 Redis 非常适合：

- 计数器
- 分布式限流
- 状态控制

---

4. I/O 多路复用（高并发连接的关键）

1️⃣ Redis 面对的真实场景

- 成千上万客户端
- 但任意时刻：
  - 真正“活跃”的连接很少

2️⃣ 不同 I/O 模型的差异

`select`

- 固定大小数组
- 每次遍历所有 FD
- 有数量上限

`poll`

- 动态数组
- 仍需全量遍历

`epoll`

- 事件驱动
- 只返回就绪 FD
- O(1) 级别唤醒

3️⃣ Redis + epoll 的组合效果

> **一个线程 + epoll ≈ 高并发服务器**

- 不阻塞
- 不空轮询
- 高效处理活跃连接

二、Redis 字符串（string）类型核心特性（深度展开）

1. 存储形式（为什么说 Redis string 很“万能”）

1️⃣ 二进制安全的真正含义

Redis string：

- 不关心内容格式
- 不以 `\0` 结尾
- 不做任何解析

因此可以存储：

- 文本
- JSON
- 图片
- 序列化对象

📌 最大值：**512MB**

2️⃣ 与关系型数据库的差异

特性	Redis string	MySQL varchar
是否解析	否	是
二进制安全	是	否
长度限制	512MB	通常 < 64KB

---

2. 核心命令与使用场景（展开）

`SET key value EX seconds`

- 原子操作
- 设置即生效
- 避免额外 `expire`

`SETNX`

- “只在不存在时设置”
- 常用于：
  - 分布式锁（基础版）
  - 初始化标记

⚠ 实际生产需配合：

- 过期时间
- Lua 脚本

`SETXX`

- 只更新已存在的 key
- 常用于：
  - 状态更新
  - 乐观控制

`MSET / MGET`

📌 **非常重要的性能优化点**

- 减少网络 RTT
- 单次请求，多次操作
- 对高并发场景极其友好

`FLUSHDB`

- 清空当前 DB
- 直接删除所有 key
- **生产环境极其危险**

---

3. string 的性能优化要点

- 小值：embstr / int
- 大值：控制 size
- 多 key 操作：优先批量命令
- 避免单 key 过大

三、命令执行效率对比（再展开）

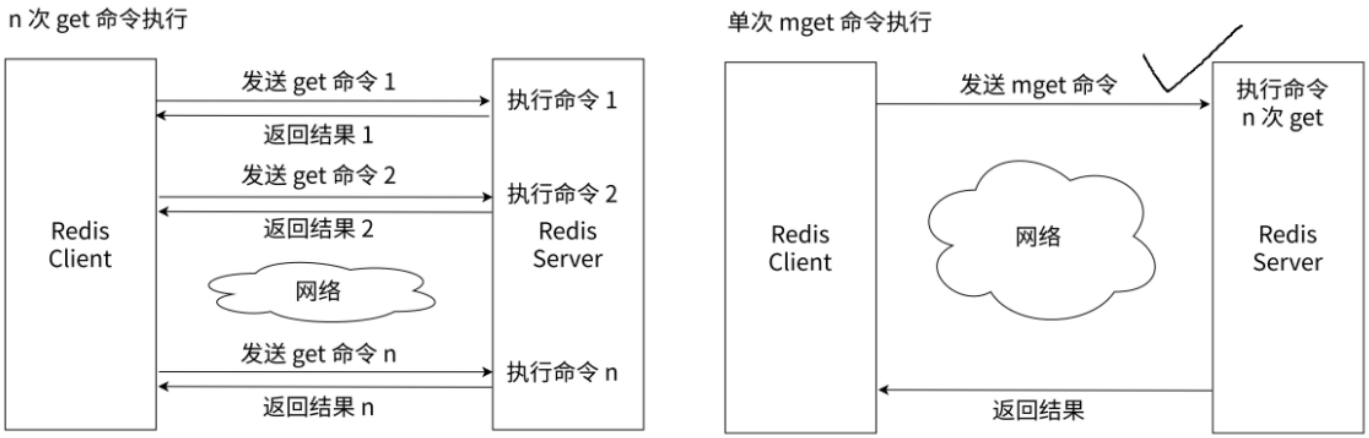
1. 真正的性能瓶颈在哪？

在高并发场景下：

> **瓶颈往往不在 Redis，而在网络**

---

2. GET vs MGET 的本质区别



多次 GET

代码块

```
1    请求 → 响应 × N
```

- RTT × N
- 延迟线性增长

单次 MGET

代码块

```
1    请求 → 批量响应
```

- RTT × 1
- 延迟几乎不变

📌 **这是“网络优化”，不是算法优化**

---

3. 工程建议

- 多 key 操作：

    - MGET / MSET
    - pipeline
- 避免 chatty API

四、学习与实践建议（展开）

1. 学 Redis 要“自顶向下”

先理解：

- Redis 要解决什么问题
- 为什么它能这么快

再学：

- 命令
- 参数
- 细节

---

2. 实践验证建议

你可以亲自验证：

- GET vs MGET 延迟
- keys * 阻塞现象
- 单 key 大 value 的影响

---

3. 生产环境经验总结

- Redis 是缓存，不是数据库替代品
- 控制 key 粒度
- 避免危险命令
- 建立使用规范

最终总结（一句话收尾）

> **Redis 的高性能并非“魔法”，而是对内存、线程模型、I/O 机制和功能边界的极度克制与极致利用。**

五、Redis单线程怎么这么快？[重要的面试题]

> 快慢自然有参照物，Redis的快是参照于数据库（MySQL、Oracle、SQL server）

1. **redis访问内存.数据库则是访问硬盘.**
2. **redis核心功能，比数据库的核心功能更简单.**

   数据库对于数据的插入删除查询....都有更复杂的功能支持.这样的功能势必要花费更多的开销.

   比如，针对插入删除，数据库中的各种约束，都会使数据库做额外的工作.

   redis干的活少，提供的功能相比于mysql也是少了不少
3. **单线程模型，避免了一些不必要的线程竞争开销.**

   redis每个基本操作，都是短平快的就是简单操作一下内存数据，不是什么特别消耗cpu的操作.就算搞多个线程，也提升不大
4. **处理网络IO的时候，使用了epoll这样的IO多路复用机制~**

   一个线程，就可以管理多个socket

   针对TCP来说，服务器这边每次要服务一个客户端，都需要给这个客户端安排一个socket

   一个服务器服务多个客户端，同时就有很多个socket.

   这些socket上都是无事不刻的在传输数据嘛???

   很多情况下，每个客户端和服务器之间的通信也没那么频繁.

   此时这么多socket大部分时间都是静默的，上面是没有数据需要传输的.

   同一时刻，只有少数的Socket是活跃的