前言

大家都知道，Redis是现在很热门的一个NoSQL数据库，也是最常见的缓存服务，很多原因都是因为它功能多而且嘎嘎快。

可是仔细想想，在Java中其实Map也是能够充当缓存的，由于是Java程序内置的，因此Map中缓存的数据也是基于内存的同样嘎嘎快，那么为什么还要快男Redis呢？下面将细细道来。

1、Map自问

- Redis可以存储**几十个G**的数据，Map行吗？
- Redis的缓存可以进行**本地持久化**，Map行吗？
- Redis可以作为**分布式缓存**，Map只能在同一个JVM中进行缓存；
- Redis支持**每秒百万级的并发**，Map行吗？
- Redis有**过期机制**，Map有吗？
- Redis有丰富的API，支持**非常多的应用场景**，Map行吗？

> 🌈
>
> Map：这波输麻了……

2、Map和Redis对比

通过下面表格分析，在大部分情况下，Redis都能够碾压Map：

特点	Redis	HashMap
数据存储	内存数据库，数据存储在内存中，支持持久化	Java集合，数据存储在Java内存中
数据结构	支持多种数据结构，如String、List、Set等	仅支持键值对数据结构，键和值可以是任意对象
持久化	支持RDB持久化和AOF持久化	不支持持久化

特点	Redis	HashMap
查询速度	由于数据存储在内存中，查询速度非常快	查询速度与数据量有关
网络传输	支持网络传输，可以用于分布式缓存和消息队列	仅用于本地内存数据存储
线程安全	单线程，不会出现并发导致的线程安全问题	线程不安全，易出现并发导致的线程安全问题
数据安全	可设置密码进行访问控制	无访问控制
高可用性	支持主从复制和哨兵模式保障高可用性	仅用于本地内存数据存储
分布式支持	支持分布式部署和数据分片	仅用于单机环境
功能扩展	支持插件和Lua脚本扩展功能	仅能使用Java提供的集合操作

但是Map并非一无是处的，在某些情况下，使用Map会更轻量，成本更低：

1. **单机环境**：当应用程序只在单机上运行，无需进行数据分片和分布式部署时，可以优先选择使用Java的Map。
2. **简单数据结构**：当数据需求较为简单，只需要使用键值对数据结构，不需要复杂的数据类型和操作时，Map是一个简单而高效的选择。
3. **本地内存数据存储**：当数据量相对较小，且不需要进行持久化存储时，Map可以直接将数据存储在JVM内存中，提供较快的访问速度。
4. **不需要分布式支持**：当应用程序不需要分布式部署，不需要进行数据分片和分布式缓存时，Map是一个合适的选择。
5. **数据安全性要求较低**：Map的访问控制相对简单，不支持复杂的密码设置和访问权限控制，适用于对数据安全性要求不高的场景。

3、ConcurrentHashMap和Redis对比

前面提及到普通的Map是存在线程安全问题的，那么如果是ConcurrentHashMap呢？

特点	ConcurrentHashMap	Redis
类型	Java本地线程安全哈希表	分布式内存数据库
数据存储	JVM内存中，属于本地数据结构	内存中，支持持久化到磁盘
分布式特性	不支持分布式部署	支持分布式部署，多台服务器
功能	通用哈希表，适用于本地多线程环境	提供丰富的数据结构和功能
性能	本地数据结构，读写操作性能高	优秀的性能，适用于大规模数据和高并发访问

特点	ConcurrentHashMap	Redis
线程安全性	线程安全	线程安全
持久化	不支持持久化	支持持久化
数据复制和分片	不支持数据复制和分片	支持数据复制和分片

所以总的来说，当只是单机并且需求简单的情况下，可以优先考虑使用Map进行控制成本和轻量化代码，如果具备线程安全的需求，那么可以上ConcurrentHashMap进行控制线程安全问题。

但是这些的前提都是需要在单机的情况下，因为Map是存储在JVM中的，而JVM并非是分布式数据共享的而是单机的，因此当涉及到分布式集群的场景下，Redis无疑是更优解。

4、Redis为什么是单线程的

前面有提及到，Redis在线程安全这一块是因为单线程的原因所以线程安全，那么为什么是单线程的呢？官方是这么给出解释的：

How can Redis use multiple CPUs or cores?

It's not very frequent that CPU becomes your bottleneck with Redis, as usually Redis is either memory or network bound. For instance, when using pipelining a Redis instance running on an average Linux system can deliver 1 million requests per second, so if your application mainly uses O(N) or O(log(N)) commands, it is hardly going to use too much CPU.

However, to maximize CPU usage you can start multiple instances of Redis in the same box and treat them as different servers. At some point a single box may not be enough anyway, so if you want to use multiple CPUs you can start thinking of some way to shard earlier.

核心意思是：**CPU 并不是制约 Redis 性能表现的瓶颈所在**，更多情况下是受到内存大小和网络I/O的限制，所以 Redis 核心网络模型使用单线程并没有什么问题，如果你想要使用服务的多核CPU，可以在一台服务器上启动多个节点或者采用分片集群的方式。

除了上面的官方回答，选择单线程的原因也有下面的考虑。

使用了单线程后，可维护性高，多线程模型虽然在某些方面表现优异，但是它却引入了程序执行顺序的不确定性，带来了并发读写的一系列问题，**增加了系统复杂度、同时可能存在线程切换、甚至加锁解锁、死锁造成的性能损耗**。

5、Redis真的是单线程吗

是，但不完全是。

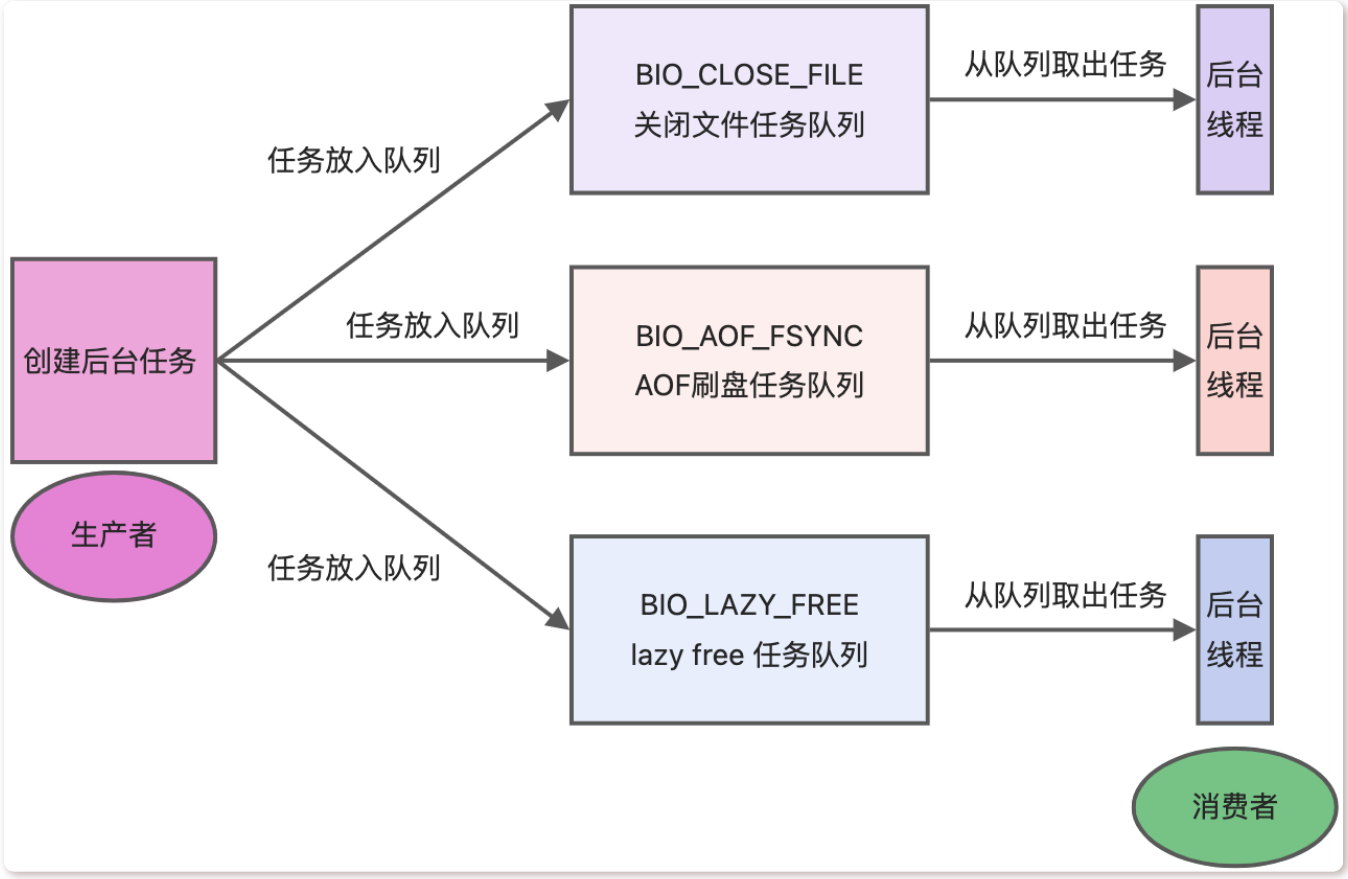
Redis 单线程指的是「接收客户端请求→解析请求 →进行数据读写等操作→发送数据给客户端」这个过程是由一个线程（主线程）来完成的，这也是我们常说 Redis 是单线程的原因。

但是，**Redis 程序并不是单线程的**，Redis 在启动的时候，是会**启动后台线程**（BIO）的：

- **Redis 在 2.6 版本**，会启动 2 个后台线程，分别处理关闭文件、AOF 刷盘这两个任务；
- **Redis 在 4.0 版本之后**，新增了一个新的后台线程，用来异步释放 Redis 内存，也就是 `lazyfree` 线程。例如执行 `unlink key / flushdb async / flushall async` 等命令，会把这些删除操作交给后台线程来执行，好处是不会导致 Redis 主线程卡顿。因此，当我们要删除一个大 key 的时候，不要使用 del 命令删除，因为 del 是在主线程处理的，这样会导致 Redis 主线程卡顿，因此我们应该使用 unlink 命令来异步删除大key。

之所以 Redis 为「关闭文件、AOF 刷盘、释放内存」这些任务创建单独的线程来处理，是因为这些任务的操作都是**很耗时**的，如果把这些任务都放在主线程来处理，那么 Redis 主线程就很容易发生阻塞，这样就无法处理后续的请求了。

后台线程相当于一个消费者，生产者把耗时任务丢到任务队列中，消费者（BIO）不停轮询这个队列，拿出任务就去执行对应的方法即可。



关闭文件、AOF 刷盘、释放内存这三个任务都有各自的任务队列：

- `BIO_CLOSE_FILE`，关闭文件任务队列：当队列有任务后，后台线程会调用 close(fd) ，将文件关闭；
- `BIO_AOF_FSYNC`，AOF刷盘任务队列：当 AOF 日志配置成 everysec 选项后，主线程会把 AOF 写日志操作封装成一个任务，也放到队列中。当发现队列有任务后，后台线程会调用 fsync(fd)，将 AOF 文件刷盘，
- `BIO_LAZY_FREE`，lazy free任务队列：当队列有任务后，后台线程会 free(obj) 释放对象 / free(dict) 删除数据库所有对象 / free(skiplist) 释放跳表对象；

6、为什么又引入了多线程呢

虽然 Redis 的主要工作（网络 I/O 和执行命令）一直是单线程模型，但是**在 Redis 6.0 版本之后，也采用了多个 I/O 线程来处理网络请求，这是因为随着网络硬件的性能提升，Redis 的性能瓶颈有时会出现在网络 I/O 的处理上**。

所以为了提高网络 I/O 的并行度，Redis 6.0 对于网络 I/O 采用多线程来处理。**但是对于命令的执行，Redis 仍然使用单线程来处理**，所以大家不要误解 Redis 有多线程同时执行命令。

Redis 官方表示，**Redis 6.0 版本引入的多线程 I/O 特性对性能提升至少是一倍以上**。

Redis 6.0 版本支持的 I/O 多线程特性，默认情况下 I/O 多线程只针对发送响应数据（write client socket），并不会以多线程的方式处理读请求（read client socket）。要想开启多线程处理客户端读请求，就需要把 `Redis.conf` 配置文件中的 `io-threads-do-reads` 配置项设为 yes。

```
//读请求也使用io多线程
io-threads-do-reads yes
```

同时， Redis.conf 配置文件中提供了 IO 多线程个数的配置项。

```
// io-threads N，表示启用 N-1 个 I/O 多线程（主线程也算一个 I/O 线程）
io-threads 4
```

关于线程数的设置，官方的建议是如果为4核的CPU，建议线程数设置为2或3，如果为8核 CPU 建议线程数设置为6，**线程数一定要小于机器核数，线程数并不是越大越好**。

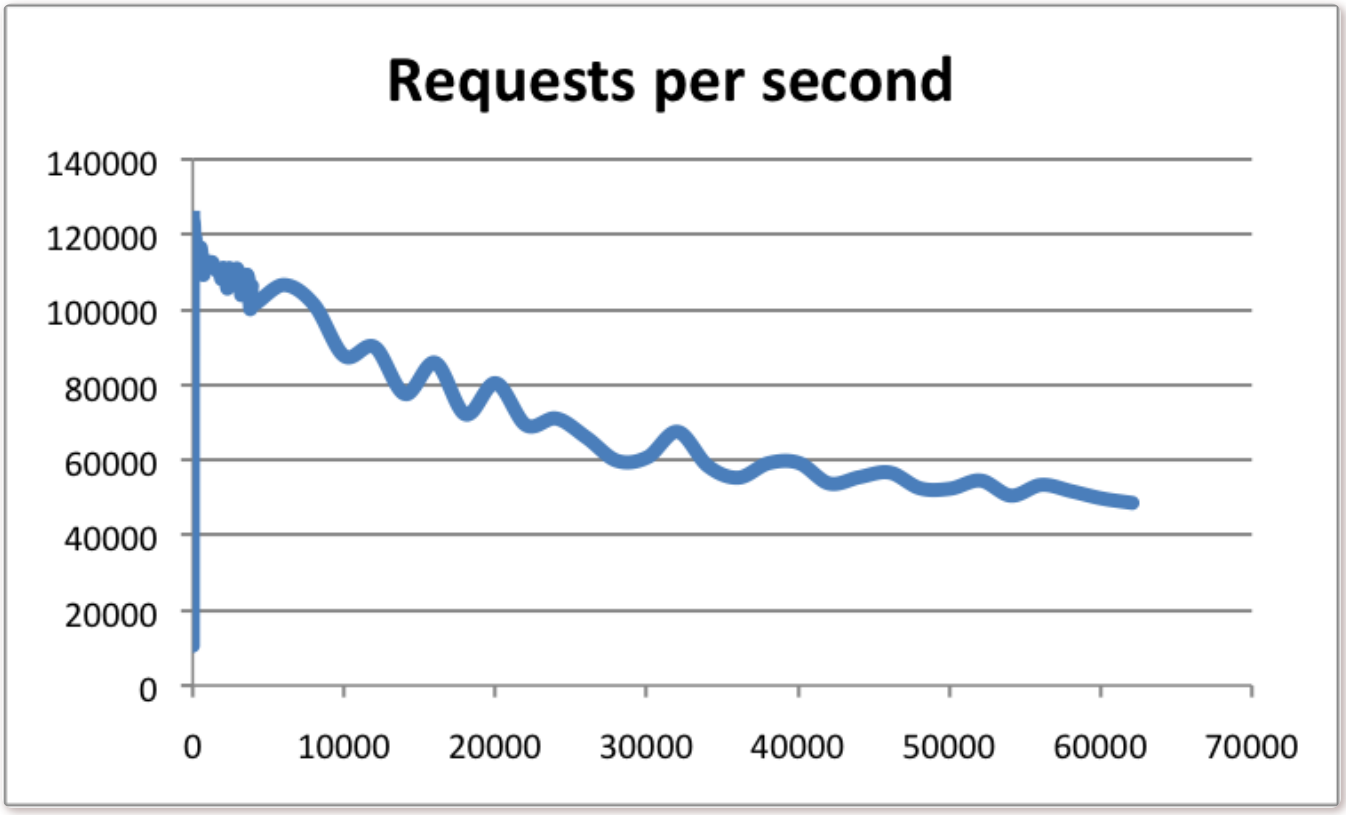
因此，Redis 6.0版本之后，Redis在启动的时候，默认情况下会创建一个主线程，并**额外创建6个线程**：

- `Redis-server`：Redis的主线程，主要负责执行命令；
- `bio_close_file`、`bio_aof_fsync`、`bio_lazy_free`：三个后台线程，分别异步处理关闭文件任务、AOF刷盘任务、释放内存任务；
- `io_thd_1`、`io_thd_2`、`io_thd_3`：三个I/O线程，io-threads默认是4，所以会启动 `3（4-1）` 个 I/O 多线程，用来分担Redis网络I/O的压力。

7、Redis单线程为什么还这么快

Redis都是单线程了，为什么还能当快男性能嘎嘎好呢?

官方使用基准测试的结果是，**单线程的 Redis 吞吐量可以达到 10W/每秒**，细分的话读的速度是110000次/秒，写的速度是81000次/秒。如下图所示：



之所以 Redis 采用单线程（网络 I/O 和执行命令）那么快，有如下几个原因：

- Redis 的大部分操作**都在内存中完成**，并且采用了高效的数据结构，因此 Redis 瓶颈可能是机器的内存或者网络带宽，而并非CPU，既然CPU不是瓶颈，那么自然就采用单线程的解决方案了；
- Redis 采用单线程模型可以**避免了多线程之间的竞争**，省去了多线程切换带来的时间和性能上的开销，而且也不会导致死锁问题。
- Redis 采用了**I/O 多路复用机制**处理大量的客户端 Socket 请求，**IO 多路复用机制是指一个线程处理多个 IO 流**，就是我们经常听到的 *select/epoll* 机制。简单来说，**在 Redis 只运行单线程的情况下，该机制允许内核中，同时存在多个监听 Socket 和已连接 Socket**。内核会一直监听这些 Socket 上的连接请求或数据请求。一旦有请求到达，就会交给Redis线程处理，这就实现了一个Redis线程处理多个IO流的效果。