京东二面：Redis为什么快？我说Redis是纯内存访问的，然后他对我笑了笑。。。。。。
==========

引言

Redis是一个高性能的开源内存数据库，以其快速的读写速度和丰富的数据结构支持而闻名。作为一个轻量级、灵活的键值存储系统，Redis在各种应用场景下都展现出了惊人的性能优势。无论是作为缓存工具、会话管理组件、消息传递媒介，还是在实时数据处理任务和复杂的分布式系统架构中，Redis均扮演了至关重要的角色。而Redis为什么快的原因也是我们尝尝遇见的高频面试问题。接下来我们就一起探讨一下Redis快的原因。

本文将深入探讨Redis之所以快速处理大规模数据的原因。我们将从Redis基于内存操作的特性、高效的内存数据结构、单线程模型、I/O多路复用技术、底层模型和优化技术、持久化机制以及网络通信协议等多个方面进行分析和讨论。通过深入了解Redis内部机制和性能优化技术，我们可以更好地理解Redis之所以快速的根本原因，以及如何在实际应用中充分发挥其优势。

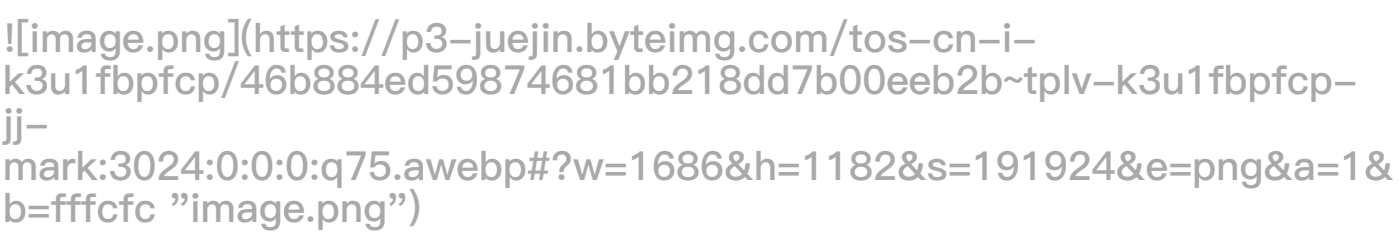


image.png

完全基于内存

Redis作为一种内存导向型数据库系统，其关键特性在于将所有数据实体，包括键值对及其相关的复杂数据结构，完全寄宿于内存之中。相较于依赖磁盘存储的传统数据库系统，Redis巧妙地运用内存的高速读写特性，显著提升了系统的响应速率与整体性能表现。

内存相对于磁盘具备无可比拟的读写速度优势，使得Redis能够即时、高效地处理数据存取。在读取操作层面，Redis无需经过耗时的磁盘I/O过程，只需在内存空间内迅速定位所需数据，从而显著降低了访问延迟；而在写入操作时，Redis同样直接作用于内存区域，新数据能即刻生效，仅在执行持久化策略时

，例如RDB快照或AOF日志记录，数据才会被异步地或按需地同步至磁盘，以确保即使在系统重启后数据仍能得以恢复，但此过程并不会妨碍Redis在常规操作中维持其卓越的性能表现。

说到这，我们就会想到，一台服务器的内存不是无限的，相反的是比较紧张的，Redis基于内存操作，那么Redis究竟是如何在有限内存空间中进行精细且高效的内存管理呢?

过期键删除

Redis支持为键设置过期时间（TTL），并且在键过期后会通过两种方式自动删除它们:

1. **惰性删除（Lazy Expire）**：在访问某个键时，Redis会检查该键是否已经过期，如果已经过期，则在访问时将其删除。这意味着只有当有客户端尝试访问过期的键时，Redis才会执行删除操作。这种方式的优势在于避免了不必要的操作，只有在需要时才进行删除，但缺点是可能会导致过期键在一段时间内仍然占用内存。
2. **定期删除（Active Expire）**：Redis周期性地（默认每秒10次）随机抽取一部分键，并检查它们的过期时间。如果发现某个键已经过期，则立即将其删除。这种方式可以保证过期键在一定时间内被及时删除，避免了过期键长时间占用内存。但定期删除会带来额外的CPU消耗，因为需要在每次抽取时检查键的过期时间。

这两种方式结合起来，可以有效地管理和清理过期键，保证Redis的内存使用在合理范围内。同时，我们在日常开发中可以根据具体业务场景和需求调整过期策略的配置，以达到最佳的性能和内存利用率。

内存淘汰策略

内存淘汰策略是Redis用于释放内存空间的一种机制，当内存空间不足时（达到或超过了配置的`maxmemory`），Redis会根据预先设置的淘汰策略来选择要删除的键，从而释放内存空间。通过合理选择和配置内存淘汰策略，可以有效地管理内存使用，防止内存溢出，并保证系统的稳定性和性能。

常见的内存淘汰策略:

1. **LRU（最近最少使用）**：

LRU策略会删除最近最少被访问的键。Redis会记录每个键最后一次被访问的时间戳，并定期检查这些时间戳，选择最久未被访问的键进行删除。LRU策略适用于缓存场景，通常最久未被访问的键可能是最不常用的，因此删除这些键可以释放更多的内存空间。
2. **LFU（最不经常使用）** ：

LFU策略会删除最不经常被访问的键。Redis会记录每个键被访问的频率，并定期检查这些频率，选择访问频率最低的键进行删除。LFU策略适用于对访问频率较低的键进行淘汰，从而释放内存空间。
3. **TTL（键的过期时间）** ：

TTL策略会删除已经过期的键。Redis会定期检查键的过期时间，并删除已经过期的键。通过设置键的过期时间，可以自动清理不再需要的数据，释放内存空间。
4. **随机删除**：

随机删除策略会随机选择一些键进行删除。虽然这种策略不考虑键的使用频率或过期时间，但在某些情况下可能会是一种简单且有效的淘汰方式，尤其是在内存空间不足时。
5. **淘汰固定数量的键**：

淘汰固定数量的键策略会选择要删除的键的数量，然后按照一定的规则（如LRU或LFU）来选择要删除的键。这种策略可以保证每次淘汰都释放固定数量的内存空间。

当Redis的内存使用达到配置的`maxmemory`限制时，就会触发内存淘汰策略，以释放内存空间。合理选择内存淘汰策略，并根据系统的需求设置`maxmemory`参数，可以有效地管理内存使用，保证系统的稳定性和性能。通过合理配置内存限制和内存淘汰策略，可以有效地管理Redis的内存使用，保证系统在内存空间不足时能够及时释放内存，避免因内存溢出而导致系统性能下降或者崩溃。

> 修改内存`maxmemory`只需要在`redis.conf`配置文件中配置`maxmemory-policy`参数即可。

内存碎片管理

内存碎片整理是指对Redis中的内存空间进行重新排列和整理，以减少内存碎片的数量和大小。内存碎片是指已分配但不再使用的内存块，这些内存块虽然被标记为已分配，但实际上并未被有效利用，造成了内存的浪费。

在Redis中，由于数据的增删改查操作不断进行，会导致内存空间中出现大量的

内存碎片。这些内存碎片虽然单个很小，但如果积累起来会导致内存碎片化，降低内存利用率，影响系统的性能和稳定性。

为了解决内存碎片化的问题，Redis会定期进行内存碎片整理操作。内存碎片整理过程包括以下几个步骤：

1. **遍历内存空间**：Redis会遍历整个内存空间，检查每个内存块的状态，包括已分配和未分配的内存块。
2. **合并相邻的空闲内存块**：Redis会尝试合并相邻的空闲内存块，将它们合并成一个更大的内存块。这样可以减少内存碎片的数量，提高内存利用率。
3. **移动数据**：如果有必要，Redis可能会将数据从一个内存块移动到另一个内存块，以便更好地组织内存空间。这个过程可能会比较耗时，因为需要将数据从一个位置复制到另一个位置。
4. **释放不再使用的内存块**：最后，Redis会释放那些不再使用的内存块，以便它们可以被重新分配给新的数据。

通过定期进行内存碎片整理操作，Redis可以保持内存空间的连续性，减少内存碎片化的程度，提高内存利用率，从而提高系统的性能和稳定性。但是，内存碎片整理过程可能会消耗一定的系统资源，尤其是在内存碎片较多的情况下。所以，通常情况下，Redis会选择在系统负载较低的时候进行碎片整理操作，以避免对系统性能产生不利影响。

高效的内存数据结构

Redis作为一个内存数据库系统，提供了丰富且高效的内存数据结构，包括字符串（`String`）、列表（`List`）、集合（`Set`）、有序集合（`Sorted Set`）、哈希（`Hash`）等。这些数据结构不仅具有简单易用的特点，还能够在内存中高效地存储和操作数据，为Redis的快速性能提供了坚实的基础。

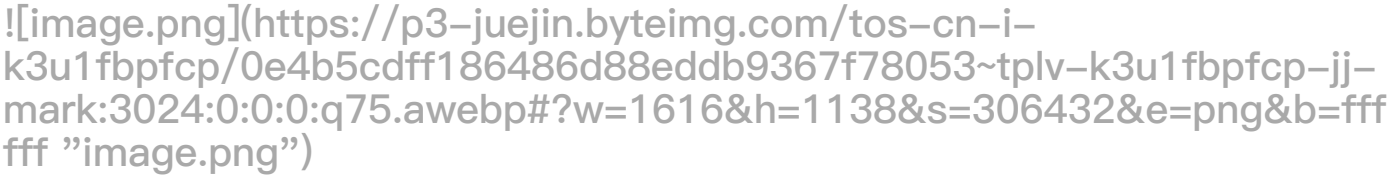


image.png

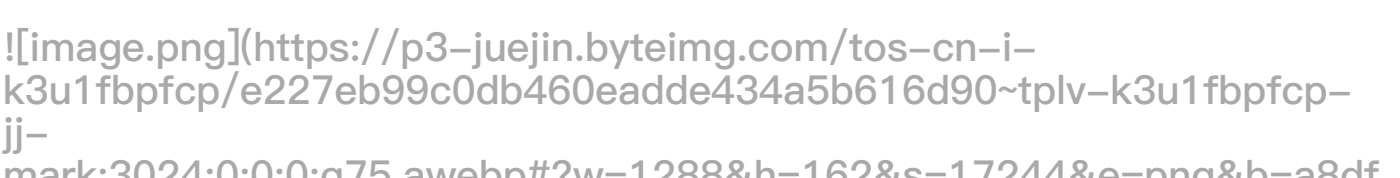
动态字符串

动态字符串是一种能够动态扩展长度的字符串实现方式。在许多编程语言和数

据结构中都有类似的实现，如C语言中的动态数组（dynamic array）。而SDS是Redis中的一种简单动态字符串结构，它是一种动态大小的字节数组，用于存储和操作字符串数据。SDS是Redis内部数据结构的基础，也是字符串数据结构的底层实现。它的结构如下：

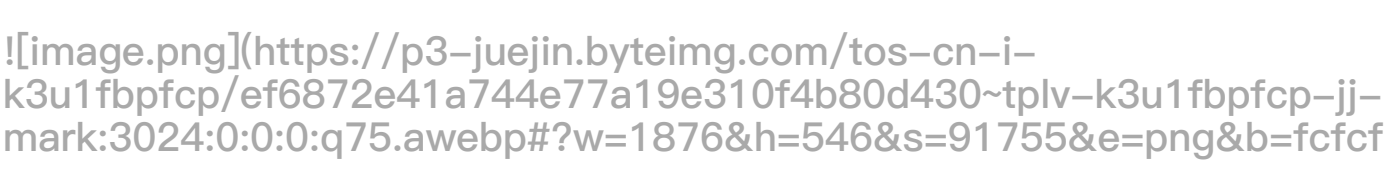
```
/*
 * redis中保存字符串对象的结构
 */
struct sdshdr {
    //用于记录buf数组中使用的字节的数目，和SDS存储的字符串的长度相等
    int len;
    //用于记录buf数组中没有使用的字节的数目
    int free;
    //字节数组，用于储存字符串
    char buf[]; //buf的大小等于len+free+1，其中多余的1个字节是用来存储'\0'的
};
```

在`C`语言中传统字符串是使用长度为N+1的字符数组来表示长度为 的字符串，并且字符串数组的最后一个元素总是空字符`'\0'`。



如果我们想要获取上述`CODERACADEMY`的长度，我们需要从头开始遍历，直到遇到 '\0' 为止。

而Redis的SDS的数据结构使用一个`len`字段记录当前字符串的长度，使用`free`表示空闲的长度。想要获取长度只需要获取`len`字段即可。



c ”image.png”)

image.png

我们可以看出`C`语言获取字符串长度的时间复杂度为`O(N)`，而SDS获取字符串长度的时间复杂度为`O(1)`。除此之外，SDS相对于C语言字符串还有如下区别：

特征	C语言字符串	SDS
类型	静态字符数组	动态字符串结构
内存管理	需手动分配和释放内存	自动扩展和释放内存
存储空间	需要提前预留足够的空间	根据需要动态调整大小
长度计算	需要遍历整个字符串计算长度	O(1)复杂度直接获取字符串长度
二进制安全	不二进制安全	二进制安全
缓冲区溢出保护	不提供缓冲区溢出保护	提供缓冲区溢出保护
操作复杂度	操作复杂度随字符串长度增加而增加	操作复杂度不受字符串长度影响
可拓展性	不易扩展，需要手动处理内存扩展	自动扩展，支持动态调整大小

细说下来，SDS相对于C语言字符串有如下优点：

1. **二进制安全**： SDS可以存储任意二进制数据，而不仅仅是文本字符串。这意味着SDS可以存储包括图片、视频、音频等在内的各种二进制数据，而不会受到特殊字符或者空字符的限制，具有更广泛的适用性。
2. **动态扩展**： SDS的大小可以根据存储的字符串长度动态调整，可以根据实际需要动态分配和释放内存空间。这种动态扩展的能力使得SDS能够处理任意长度的字符串数据，而不受到固定大小的限制。
3. **O(1)复杂度的操作**： SDS支持常数时间复杂度的操作，包括添加字符、删除字符、修改字符等。无论字符串的长度是多少，这些操作的时间开销都是固定的，具有高效的性能。
4. **缓冲区溢出保护**： SDS在存储字符串时，会自动添加一个空字符('\0')作为字符串的结束标志，保证字符串的有效性和安全性。这种缓冲区溢出保护能够防止缓冲区溢出的问题，提高了系统的稳定性和安全性。
5. **惰性空间释放**： 当SDS缩短字符串时，并不会立即释放多余的空间，而是将多余的空间保留下来，以备后续的再利用。这种惰性空间释放的策略可以减少内存分配和释放的开销，提高内存利用率。

这些优点使得SDS在Redis中被广泛应用于存储和操作字符串数据，为Redis的高性能和高可靠性提供了坚实的基础。

双端链表

Redis中的双端链表是一种经过优化的数据结构，用于存储有序的元素集合。它具有双向链接的特性，每个节点都包含指向前一个节点和后一个节点的指针。

image.png

双端链表中的节点是链表的基本构建单元，它存储了链表中的数据元素以及指向前一个节点和后一个节点的指针。在Redis中，双端链表节点的定义通常如下：

```
typedef struct listNode {
    struct listNode *prev;  // 指向前一个节点的指针
    struct listNode *next;  // 指向后一个节点的指针
    void *value;            // 存储的数据元素
} listNode;
```

双端链表中的节点包含了以下几个关键属性：

1. **prev指针**：`prev`指针是指向前一个节点的指针，它指向链表中当前节点的前一个节点。如果当前节点是链表的头节点，则`prev`指针为`NULL`。通过`prev`指针，可以在双端链表中方便地向前遍历节点。
2. **next指针**：`next`指针是指向后一个节点的指针，它指向链表中当前节点的后一个节点。如果当前节点是链表的尾节点，则`next`指针为`NULL`。通过`next`指针，可以在双端链表中方便地向后遍历节点。
3. **value数据域**：`value`数据域用于存储链表节点所包含的数据元素。这个数据元素可以是任意类型的数据，因此在Redis中的双端链表中，通常使用`void *`类型来表示。这种设计使得双端链表可以存储任意类型的数据元素。

通过这些属性，双端链表节点构成了链表的基本组成部分，它们通过`prev`和`next`指针连接在一起，形成了双向链接的链表结构。

对于链表中描述链表整体属性的元数据，它的结构如下：

```
typedef struct list {
    listNode *head;  // 头节点指针
    listNode *tail;  // 尾节点指针
    unsigned long len;  // 链表长度
    // 其他字段...
} list;
```

从结构中可以看出元数据中还有两个特殊的节点：头节点（head node）和尾节点（tail node），它们分别位于链表的头部和尾部。而他们的作用如下：

1. **头节点（head node）**：

头节点是双端链表中的第一个节点，也是链表的入口。它通常用于存储链表的起始位置信息，以便快速定位链表的起始位置。在双端链表中，头节点的特点是没有前一个节点，即头节点的`prev`指针为`NULL`。头节点通常用于存储链表的头部元数据或者哨兵节点。
2. **尾节点（tail node）**：

尾节点是双端链表中的最后一个节点，也是链表的结束位置。它通常用于存储链表的结束位置信息，以便快速定位链表的结束位置。在双端链表中，尾节点的特点是没有后一个节点，即尾节点的`next`指针为`NULL`。尾节点通常用于存储链表的尾部元数据或者哨兵节点。

在Redis中，通常会使用头节点和尾节点来表示双端链表的起始位置和结束位置，以方便对链表进行操作。Redis中的双端链表常见操作如下：

* 头节点（head）：表示双端链表的头部节点，通过头节点可以快速定位链表的起始位置，通常用于添加和删除链表的头部元素。
* 尾节点（tail）：表示双端链表的尾部节点，通过尾节点可以快速定位链表的结束位置，通常用于添加和删除链表的尾部元素。

通过头节点和尾节点，可以方便地对双端链表进行头部插入、尾部插入、头部删除、尾部删除等操作，从而实现了对双端链表的高效操作。

除了上述头尾节点以外，链表的元数据中还有`len`参数，这个参数用于记录链表的当前长度。每当链表中添加或删除节点时，Redis会相应地更新`len`字段的值，以反映链表的当前长度。这个参数与SDS里类似，获取链表长度时不用再遍历整个链表，直接拿到`len`值就可以了，这个时间复杂度是 O(1)。

压缩列表

Redis中的压缩列表（ziplist）是一种特殊的数据结构，用于存储列表和哈希数据类型中的元素。压缩列表通过将多个小的数据单元压缩在一起，以节省内存空间，并提高访问效率。

![image.png](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/b58f250377f441e085c740ca8c5aab49~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1774&h=1066&s=328682&e=png&b=ffffff ”image.png”)

image.png

对于压缩列表，它的主要作用如下：

1. **紧凑的存储形式**： 压缩列表以一种紧凑的方式存储数据，将多个元素紧密地排列在一起，节省了存储空间。在压缩列表中，相邻的元素可以共享同一个内存空间，这种紧凑的存储形式可以大大减少内存的消耗。
2. **灵活的编码方式**： 压缩列表中的每个元素都可以采用不同的编码方式进行存储，包括整数编码、字符串编码和字节数组编码等。根据元素的类型和大小，压缩列表会选择合适的编码方式来存储数据，以进一步节省内存空间。
3. **快速的随机访问**： 压缩列表支持快速的随机访问操作，可以通过下标索引来访问压缩列表中的任意元素。由于压缩列表采用紧凑的存储形式，因此可以通过简单的偏移计算来实现快速的元素访问，具有较高的访问效率。
4. **动态调整大小**： 压缩列表支持动态调整大小，可以根据实际需要自动扩展或收缩内存空间。当压缩列表中的元素数量增加时，可以动态地分配额外的内存空间，以容纳更多的元素；当元素数量减少时，可以释放多余的内存空间，以节省内存资源。
5. **适用于小型数据集**： 压缩列表适用于存储小型数据集，例如长度较短的列表或者哈希表。由于压缩列表采用紧凑的存储形式，并且支持快速的随机访问，因此特别适合于存储数量较少但访问频繁的数据。

字典

在Redis中，字典（dictionary）是一种用于存储键值对数据的数据结构，也称为哈希表（hash table）。字典是Redis中最常用的数据结构之一，具有快速查找、动态调整大小、哈希冲突处理、迭代器支持等特点，适用于各种数据存储和操作需求，实现键值对存储和快速查找。

字典以键值对的形式存储数据，每个键都与一个值相关联。在Redis中，键和值都可以是任意类型的数据，如字符串、整数、列表或哈希表。

字典利用哈希表实现，具备快速查找的特性。通过将键映射到哈希表的索引位置，字典能以常数时间复杂度（O(1)）内查找、插入和删除键值对，即使在大型数据集中也能保持高效。

此外，字典支持动态调整大小，随着键值对数量的变化，能自动扩展或收缩内存空间，以适应数据量的变化。

在存储数据时，如果产生了哈希冲突，字典可以采用开放寻址法或链表法等策略，根据哈希表的大小和负载因子选择合适的冲突解决方法，确保查找性能高效。

跳跃表

跳跃表（Skip List）是一种基于链表的数据结构，它利用多级索引来加速查找操作，类似于平衡树，但实现起来更加简单，具有较好的平均查找性能。在Redis中，跳跃表用于有序集合（Sorted Set）数据类型的实现，提供了高效的有序数据存储和检索功能。

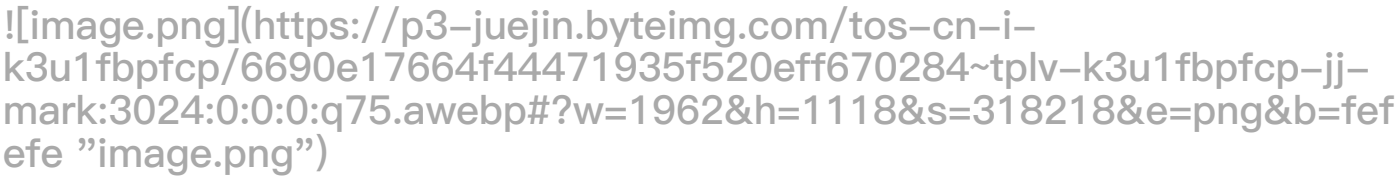


image.png

跳跃表通过维护多级索引，每个级别的索引都是原始链表的子集，用于快速定位元素。每个节点在不同级别的索引中都有一个指针，通过这些指针，可以在不同级别上进行快速查找，从而提高了查找效率。

![image.png](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/539e5757e1b046e498b3f8ba8523fc2d~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1902&h=692&s=81286&e=png&b=ffffff"image.png")

image.png

跳跃表的平均查找性能为O(log n)，与平衡树相当，但实现起来更加简单。跳跃表通过多级索引来实现快速查找，使得查找时间随着数据量的增加而呈对数增长。但是跳跃表的空间复杂度相对较高，因为它需要额外的空间来维护多级索引。不过跳跃表的空间占用通常是合理的，且具有可控性，可以根据实际需求调整级别和索引节点的数量，以平衡空间和性能的需求。

除此之外，跳跃表支持动态调整大小，可以根据实际需要自动扩展或收缩内存空间。当有序集合中的元素数量增加时，跳跃表会动态地增加级别和索引节点，以提高查找效率；当元素数量减少时，可以收缩跳跃表的大小，以节省内存资源。并且跳跃表的插入和删除操作具有较高的效率，通过维护多级索引，可以在O(log n)的时间复杂度内完成插入和删除操作。

单线程模型

Redis中的单线程模型是指Redis在其核心数据处理部分采用单一的主线程来执行网络IO操作、接收客户端命令请求、执行命令操作以及返回结果。Redis服务端的网络IO和键值对读写操作都由一个线程统一负责，而诸如持久化、集群数据同步等任务则是由其他线程来执行。在单线程模型下，Redis服务器是单线程运行的，即每个客户端的请求都是依次顺序执行的。

而使用单线程所带来的好处：

1. **避免上下文切换**：

多线程环境下，线程间的上下文切换会带来额外的CPU开销。Redis通过单线程模型消除了多线程环境下的上下文切换成本，使得CPU资源更多地用于执行实际的命令处理。
2. **简化数据操作的并发控制**：

单线程模型确保了同一时间内只有一个操作在处理数据，因此不需要使用锁机制来保护数据的完整性，避免了多线程编程中常见的锁竞争和死锁问题，从而提高了系统的执行效率。
3. **内存操作性能优越**：

Redis是一个基于内存操作的数据库，大部分操作都在内存中完成，本身就有很高的执行速度。单线程模型下，内存操作无需考虑并发控制，因此能够实现更高的内存读写效率。

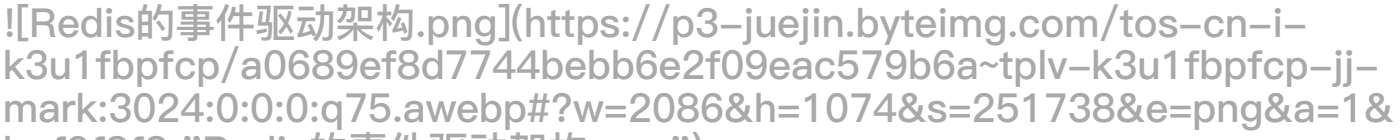
在日常开发中，我们通常会使用并发编程来提高服务的吞吐量。这时，我们可能会产生一个疑问：Redis的单线程模型是否能够充分利用CPU资源呢?

实际上，由于Redis是基于内存的操作，使用Redis时，CPU很少会成为瓶颈。相反，Redis主要受限于服务器内存和网络带宽。例如，在典型的Linux系统上，通过使用pipelining技术，Redis能够实现较高的吞吐量，每秒可以处理大量的请求。因此，如果应用程序主要使用O(N)或O(log(N))的命令，它几乎不会对CPU资源造成过多的负载。综上所述，考虑到单线程模型的实现简单且CPU很少成为瓶颈，因此采用单线程方案是合理的选择。

单线程模型限制了Redis的并发能力。由于只有一个线程在处理请求，无法充分利用多核处理器的性能优势，所以可能到达服务端的请求不可能被立即处理。那么Redis是如何保证单线程的资源利用率和处理效率呢?

IO多路复用技术:

Redis通过使用IO多路复用技术（如epoll、kqueue或select等），在一个线程内同时监听多个socket连接，当有网络事件发生时（如读写就绪），再逐一处理。这样可以处理大量并发连接，并在单线程中高效地调度网络事件，使得单线程也能应对高并发场景。所以Redis服务端，整体来看，就是一个以事件驱动的程序，它的操作都是基于事件的方式进行的。Redis的事件驱动架构如图:



Redis的事件驱动架构.png

Redis的事件驱动架构是一种基于非阻塞I/O多路复用技术设计的高效处理并发请求的机制。在Redis中，事件驱动架构通过监听和处理各种网络I/O事件以及定时事件，使得Redis服务端能够在一个线程内高效地服务于多个客户端连接，并执行相关的命令操作。

事件驱动架构主要由以下几个组成部分构成:

1. **套接字（Socket）**：

套接字是客户端与Redis服务端之间进行通信的基础接口，用于双向数据传输。
2. **I/O多路复用**：

Redis服务端通过使用如epoll、kqueue等I/O多路复用技术，可以同时监听多个套接字上的读写事件。当某个客户端的套接字上有数据可读或可写时，内核会通知Redis服务端，而无需Redis反复检查每一个套接字状态。

Redis默认使用的IO多路复用技术确实是epoll。其主要优点如下：

* **并发连接限制**

相比于select和poll，epoll没有预设的并发连接数限制，能够处理的并发连接数只受限于系统资源，适合处理大规模并发连接。
* **内存拷贝优化**

epoll采用事件注册机制，仅和通知就绪的文件描述符，无需像select和poll那样在每次调用时都拷贝整个文件描述符集合，从而减少了内存拷贝的开销。
* **活跃连接感知**

epoll提供了水平触发（level-triggered）和边缘触发（edge-triggered）两种模式，可以更准确地感知活跃连接，仅当有事件发生时才唤醒处理，避免了无效的轮询操作，提升了事件处理的效率。
* **高效事件处理**

epoll利用红黑树存储待监控的文件描述符，并使用内核层面的回调机制，当有文件描述符就绪时，会直接通知应用程序，从而减少了CPU空转和上下文切换的成本。

1. **文件事件分派器（File Event Demultiplexer）**：

文件事件分派器是Redis事件驱动的核心组件，它负责将内核传递过来的就绪事件分发给对应的处理器。在Redis中，每个套接字都关联了一个或多个事件处理器，如客户端连接请求处理器、命令请求处理器和命令响应处理器等。
2. **事件处理器（Event Handlers）**：

事件处理器是Redis中处理特定事件的实际执行者。当文件事件分派器接收到一个就绪事件时，它会调用对应的事件处理器来执行相应操作，如读取客户端的命令请求，执行命令并对结果进行编码，然后将响应数据写回客户端。

而对于Redis中设计的事件主要分为两个大类：

* **文件事件（File Events）**：主要对应网络I/O操作，包括客户端连接请求（AE_READABLE事件）、客户端命令请求（AE_READABLE事件）和服务端命令回复（AE_WRITABLE事件）。
* **时间事件（Time Events）**：对应定时任务，如键值对过期检查、持久化操作等。所有时间事件都被存放在一个无序链表中，每当时间事件执行器运行时，会遍历链表并处理已到达预定时间的事件。

通过事件驱动架构，Redis能够在一个线程内并发处理大量客户端请求，而无需为每个客户端创建独立的线程。此外，由于Redis的高效内存管理、数据结构优化和单线程模型，避免了多线程环境下的锁竞争和上下文切换开销，从而实现了极高的吞吐量和响应速度。

> 在Redis 6.x版本中，虽然引入了多线程处理网络IO的部分，但核心命令执行依然保持单线程事件驱动的模型，以维持Redis原有的性能优势。

IO多路复用模型

IO多路复用的核心在于内核的是应用程序的文件描述符而非直接监控连接本身。客户端运行时产生的不同事件类型的套接字操作，会被内核捕获。在服务器端，I/O多路复用机制负责收集这些事件并将它们加入事件队列，随后通过文件事件分发器分发至对应事件处理器进行处理。

以Redis为例，在其单线程模型下，内核不间断地监测所有客户端socket的连接请求和数据传输状况。只要检测到任何socket上有待处理的动作，便会立即将控制权转交给Redis线程。这样一来，尽管仅依靠单线程，Redis仍能有效地处理多个并发的IO流。

select/epoll等IO多路复用技术提供了一种基于事件触发的回调模式，每当有不同事件发生时，Redis能够迅速调用相应的事件处理器，始终保持在处理事件的状态，从而提升了其响应速度。

由于Redis线程并不会因为等待某个特定socket的IO操作完毕而停滞，它可以流畅地在多个客户端间切换，即时响应每个客户端的不同请求，从而实现在单线程环境下对大量并发连接的有效处理和高并发性能。

简单高效的通信协议

Redis Cluster在集群内部通信中借鉴了Gossip协议的理念，采用了一种基于Gossip风格的消息传播机制。这种机制能够有效地将集群状态和节点信息在集群中的各个节点间进行快速传播和同步。类比于流行病的传播模型，Gossip协议允许节点随机选择邻居节点进行通信，从而在全网状结构中快速传播更新。

Redis Cluster、Consul和Apache Cassandra等分布式系统都采用了Gossip协议或者类似的机制来维护集群的健康状态和一致性。通过Gossip协议，节点们可以高效地共享和更新集群的元数据，如节点加入、离开、故障转移等信息。

然而，纯粹的Gossip协议在实践中可能存在信息冗余的问题，即已接收到某一信息的节点在后续的传播中可能会收到相同的信息。为了避免这种冗余和提高通信效率，这些系统通常会对Gossip协议进行优化，例如在节点间记录已知信息的状态，避免重复传播已知的更新。即便如此，Gossip协议仍然是在大规模分布式系统中实现高可用性和强一致性的有效手段，其高效性体现在只需局部通信即可逐渐达成全局一致性，同时具备良好的扩展性和容错性。

总结

最后，我们来总结一下，如何在面试中回答Redis为什么快的原因：

1. **纯内存操作**：

Redis利用内存进行数据存储，其操作基于内存读写，由于内存访问速度远超硬盘，使得Redis在处理数据时具有极高的读写速度。特别是对于简单的存取操作，由于线程在内存中执行的时间非常短，主要的时间消耗在于网络I/O，因此Redis在处理大量快速读写请求时表现出卓越的性能。
2. **单线程模型**：

Redis采用单线程模型处理客户端请求，这一设计确保了操作的原子性，避免了多线程环境下的上下文切换和锁竞争问题。这使得Redis在处理命令请求时能够保持高度的确定性和一致性，同时也简化了编程模型，降低了并发控制的复杂性。
3. **IO多路复用技术**：

Redis通过采用IO多路复用模型，如epoll，能够在一个线程中高效地处理多个客户端连接。单线程轮询监听多个套接字描述符，并将数据库的读、写、连接建立和关闭等操作转化为事件，通过自定义的事件分离器和事件处理器来高效地处理这些事件，从而避免了在等待IO操作时的阻塞。
4. **高效数据结构**：

* Redis的整体设计围绕高效数据结构展开，其中包括但不限于全局哈希表（字典），该结构提供O(1)的平均时间复杂度，并通过rehash操作动态调整哈希桶数量，减少哈希冲突，采用渐进式rehash避免一次性操作过大导致的阻塞。
* 除此之外，Redis还广泛应用了多种优化过的数据结构，如压缩表（ziplist）用于存储短数据以节省内存，跳跃表（skiplist）用于有序集合提供快速的范围查询，以及其他如列表、集合等数据结构，均针对不同场景进行深度优化，确保了在读取和操作数据时的高性能。

本文已收录于我的个人博客：[码农Academy的博客，专注分享Java技术干货，包括Java基础、Spring Boot、Spring Cloud、Mysql、Redis、Elasticsearch、中间件、架构设计、面试题、程序员攻略等](http://cxyroad.com/ "https://www.coderacademy.online/")

原文链接: https://juejin.cn/post/7350652060872310793