

公号已发：[https://mp.weixin.qq.com/s/QrGplzX7OqLAR5fXp\\_B0YA](https://mp.weixin.qq.com/s/QrGplzX7OqLAR5fXp_B0YA)

这篇文章的问题来源于一个读者之前分享的 [OPPO 后端凉经](#)，我对比较典型的一些问题进行了分类并给出了详细的参考答案。希望能对正在参加面试的朋友们能够有点帮助！准备面试的过程中，一定要多看面经，多自测！

合集地址：[《Java 后端面经精选（附详细参考答案）》](#)。

## Java

### String 为什么是不可变的？

```
public final class String implements java.io.Serializable, Comparable<String>, CharSequence {  
    private final char value[];  
    //...  
}
```

String 真正不可变有下面几点原因：

1. 保存字符串的数组被 `final` 修饰且为私有的，并且 `String` 类没有提供/暴露修改这个字符串的方法。
2. `String` 类被 `final` 修饰导致其不能被继承，进而避免了子类破坏 `String` 不可变。

在 Java 9 之后，`String`、`StringBuilder` 与 `StringBuffer` 的实现改用 `byte` 数组存储字符串。

```
public final class String implements java.io.Serializable, Comparable<String>, CharSequence {  
    // @Stable 注解表示变量最多被修改一次，称为“稳定的”。  
    @Stable  
    private final byte[] value;  
}  
  
abstract class AbstractStringBuilder implements Appendable, CharSequence {  
    byte[] value;  
}
```

新版的 `String` 其实支持两个编码方案：Latin-1 和 UTF-16。如果字符串中包含的汉字没有超过 Latin-1 可表示范围内的字符，那就会使用 Latin-1 作为编码方案。Latin-1 编码方案下，`byte` 占一个字节(8 位)，`char` 占用 2 个字节 (16)，`byte` 相较 `char` 节省一半的内存空间。

JDK 官方就说了绝大部分字符串对象只包含 Latin-1 可表示的字符。

## Motivation

The current implementation of the `String` class stores characters in a `char` array, using two bytes (sixteen bits) for each character. Data gathered from many different applications indicates that strings are a major component of heap usage and, moreover, **that most `String` objects contain only Latin-1 characters.** Such characters require only one byte of storage, hence half of the space in the internal `char` arrays of such `String` objects is going unused.

如果字符串中包含的汉字超过 Latin-1 可表示范围内的字符，`byte` 和 `char` 所占用的空间是一样的。

这是官方的介绍：<https://openjdk.java.net/jeps/254>。

## 如何创建线程？

一般来说，创建线程有很多种方式，例如继承 `Thread` 类、实现 `Runnable` 接口、实现 `Callable` 接口、使用线程池、使用 `CompletableFuture` 类等等。

不过，这些方式其实并没有真正创建出线程。准确点来说，这些都属于是在 Java 代码中使用多线程的方法。

严格来说，Java 就只有一种方式可以创建线程，那就是通过 `new Thread().start()` 创建。不管是哪种方式，最终还是依赖于 `new Thread().start()`。

关于这个问题的详细分析可以查看这篇文章：[大家都说Java有三种创建线程的方式！并发编程中的惊天骗局！](#)。

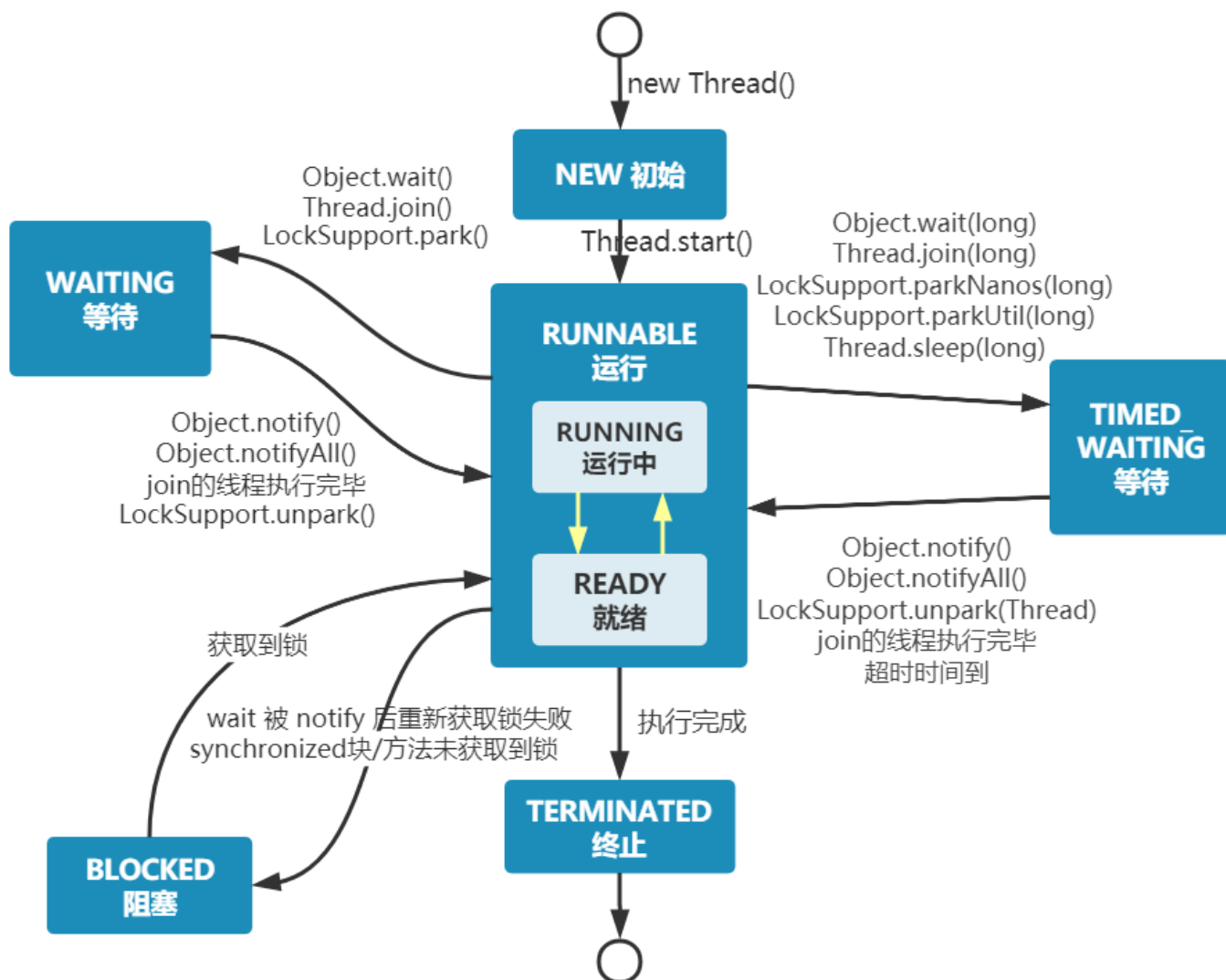
## Java 线程的状态有哪几种？

Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态：

- `NEW`: 初始状态，线程被创建出来但没有被调用 `start()`。
- `RUNNABLE`: 运行状态，线程被调用了 `start()` 等待运行的状态。
- `BLOCKED`: 阻塞状态，需要等待锁释放。
- `WAITING`: 等待状态，表示该线程需要等待其他线程做出一些特定动作（通知或中断）。
- `TIME_WAITING`: 超时等待状态，可以在指定的时间后自行返回而不是像 `WAITING` 那样一直等待。
- `TERMINATED`: 终止状态，表示该线程已经运行完毕。

线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。

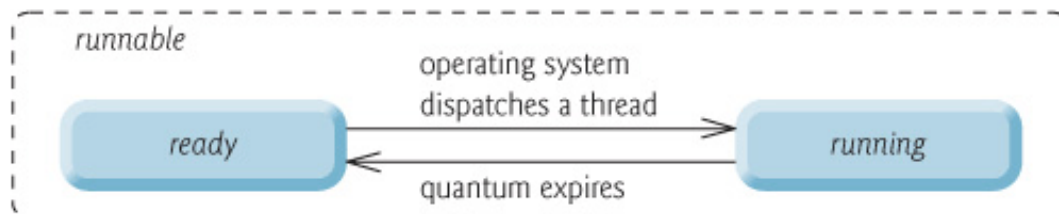
Java 线程状态变迁图(图源：[挑错 I 《Java 并发编程的艺术》中关于线程状态的三处错误](#))：



由上图可以看出：线程创建之后它将处于 **NEW（新建）** 状态，调用 `start()` 方法后开始运行，线程这时候处于 **READY（可运行）** 状态。可运行状态的线程获得了 CPU 时间片（timeslice）后就处于 **RUNNING（运行）** 状态。

在操作系统层面，线程有 **READY** 和 **RUNNING** 状态；而在 JVM 层面，只能看到 **RUNNABLE** 状态（图源：[HowToDoinJava：Java Thread Life Cycle and Thread States](#)），所以 Java 系统一般将这两个状态统称为 **RUNNABLE（运行中）** 状态。

为什么 **JVM** 没有区分这两种状态呢？（摘自：[Java 线程运行怎么有第六种状态？ - Dawell 的回答](#)）现在的时分（time-sharing）多任务（multi-task）操作系统架构通常都是用所谓的“时间分片（time quantum or time slice）”方式进行抢占式（preemptive）轮转调度（round-robin 式）。这个时间分片通常是很小的，一个线程一次最多只能在 CPU 上运行比如 10-20ms 的时间（此时处于 running 状态），也即大概只有 0.01 秒这一量级，时间片用后就要被切换下来放入调度队列的末尾等待再次调度。（也即回到 ready 状态）。线程切换的如此之快，区分这两种状态就没什么意义了。



- 当线程执行 `wait()` 方法之后，线程进入 **WAITING（等待）** 状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态。
- **TIMED\_WAITING(超时等待)** 状态相当于在等待状态的基础上增加了超时限制，比如通过 `sleep (long millis)` 方法或 `wait (long millis)` 方法可以将线程置于 **TIMED\_WAITING** 状态。当超时时间结束后，线程将会返回到 **RUNNABLE** 状态。
- 当线程进入 `synchronized` 方法/块或者调用 `wait` 后（被 `notify` ）重新进入 `synchronized` 方法/块，但是锁被其它线程占有，这个时候线程就会进入 **BLOCKED（阻塞）** 状态。
- 线程在执行完了 `run()` 方法之后将会进入到 **TERMINATED（终止）** 状态。

相关阅读：[线程的几种状态你真的了解么？](#)。

## 为什么要用线程池？项目中使用的线程池是使用内置的还是自己创建的？

线程池提供了一种限制和管理资源（包括执行一个任务）的方式。每个线程池还维护一些基本统计信息，例如已完成任务的数量。

这里借用《Java 并发编程的艺术》提到的来说一下使用线程池的好处：

- **降低资源消耗**。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度**。当任务到达时，任务可以不需要等到线程创建就能立即执行。
- **提高线程的可管理性**。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

《阿里巴巴 Java 开发手册》中强制线程池不允许使用 `Executors` 去创建，而是通过 `ThreadPoolExecutor` 构造函数的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险

`Executors` 返回线程池对象的弊端如下(后文会详细介绍到)：

- **FixedThreadPool 和 SingleThreadExecutor**：使用的是无界的 `LinkedBlockingQueue`，任务队列最大长度为 `Integer.MAX_VALUE`，可能堆积大量的请求，从而导致 OOM。
- **CachedThreadPool**：使用的是同步队列 `SynchronousQueue`，允许创建的线程数量为 `Integer.MAX_VALUE`，如果任务数量过多且执行速度较慢，可能会创建大量的线程，从而导致 OOM。
- **ScheduledThreadPool 和 SingleThreadScheduledExecutor**：使用的无界的延迟阻塞队列 `DelayedWorkQueue`，任务队列最大长度为 `Integer.MAX_VALUE`，可能堆积大量的请求，从而导致 OOM。

相关阅读：

- 8 个线程池最佳实践和坑！使用不当直接生产事故！！
- 手写一个轻量级动态线程池，很香！！

## 数据库

### 如何找到慢 SQL?

MySQL 慢查询日志是用来记录 MySQL 在执行命令中，响应时间超过预设阈值的 SQL 语句。因此，通过分析慢查询日志我们就可以找出执行速度比较慢的 SQL 语句。

出于性能层面的考虑，慢查询日志功能默认是关闭的，你可以通过以下命令开启：

```
# 开启慢查询日志功能
SET GLOBAL slow_query_log = 'ON';
# 慢查询日志存放位置
SET GLOBAL slow_query_log_file = '/var/lib/mysql/ranking-list-slow.log';
# 无论是否超时，未被索引的记录也会记录下来。
SET GLOBAL log_queries_not_using_indexes = 'ON';
# 慢查询阈值（秒），SQL 执行超过这个阈值将被记录在日志中。
SET SESSION long_query_time = 1;
# 慢查询仅记录扫描行数大于此参数的 SQL
SET SESSION min_examined_row_limit = 100;
```

设置成功之后，使用 `show variables like 'slow%';` 命令进行查看。

```
| Variable_name      | Value |
+-----+
| slow_launch_time   | 2     |
| slow_query_log      | ON    |
| slow_query_log_file| /var/lib/mysql/ranking-list-slow.log |
+-----+
3 rows in set (0.01 sec)
```

我们故意在百万数据量的表(未使用索引)中执行一条排序的语句：

```
SELECT `score`,`name` FROM `cus_order` ORDER BY `score` DESC;
```

确保自己有对应目录的访问权限：

```
chmod 755 /var/lib/mysql/
```

查看对应的慢查询日志：

```
cat /var/lib/mysql/ranking-list-slow.log
```

我们刚刚故意执行的 SQL 语句已经被慢查询日志记录了下来：

```
# Time: 2022-10-09T08:55:37.486797Z
# User@Host: root[root] @ [172.17.0.1] Id: 14
# Query_time: 0.978054 Lock_time: 0.000164 Rows_sent: 999999 Rows_examined: 1999998
SET timestamp=1665305736;
SELECT `score`,`name` FROM `cus_order` ORDER BY `score` DESC;
```

这里对日志中的一些信息进行说明：

- `Time` ：被日志记录的代码在服务器上的运行时间。
- `User@Host` ：谁执行的这段代码。
- `Query_time` ：这段代码运行时长。
- `Lock_time` ：执行这段代码时，锁定了多久。
- `Rows_sent` ：慢查询返回的记录。
- `Rows_examined` ：慢查询扫描过的行数。

实际项目中，慢查询日志通常会比较复杂，我们需要借助一些工具对其进行分析。像 MySQL 内置的 `mysqldumpslow` 工具就可以把相同的 SQL 归为一类，并统计出归类项的执行次数和每次执行的耗时等一系列对应的情况。

## 如何分析 SQL 性能

我们可以使用 `EXPLAIN` 命令来分析 SQL 的 **执行计划**。执行计划是指一条 SQL 语句在经过 MySQL 查询优化器的优化会后，具体的执行方式。

`EXPLAIN` 并不会真的去执行相关的语句，而是通过 **查询优化器** 对语句进行分析，找出最优的查询方案，并显示对应的信息。

`EXPLAIN` 适用于 `SELECT`，`DELETE`，`INSERT`，`REPLACE`，和 `UPDATE` 语句，我们一般分析 `SELECT` 查询较多。

我们这里简单来演示一下 `EXPLAIN` 的使用。

`EXPLAIN` 的输出格式如下：

```
mysql> EXPLAIN SELECT `score`,`name` FROM `cus_order` ORDER BY `score` DESC;

+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | partitions | type | possible_keys | key  | key_len | ref  | rows |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | cus_order  | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 997572 |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|    | 100.00      | Using filesort |           |      |               |      |          |      |      |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

各个字段的含义如下：

列名	含义
id	SELECT 查询的序列标识符
select_type	SELECT 关键字对应的查询类型
table	用到的表名
partitions	匹配的分区，对于未分区的表，值为 NULL
type	表的访问方法
possible_keys	可能用到的索引
key	实际用到的索引
key_len	所选索引的长度
ref	当使用索引等值查询时，与索引作比较的列或常量
rows	预计要读取的行数
filtered	按表条件过滤后，留存的记录数的百分比
Extra	附加信息

篇幅问题，我这里只是简单介绍了一下 MySQL 执行计划，详细介绍请看：[MySQL执行计划分析](#)这篇文章。

## 项目中是怎么使用索引的？联合索引了解吗？

索引是一种用于快速查询和检索数据的数据结构，其本质可以看成是一种排序好的数据结构。

索引的作用就相当于书的目录。打个比方：我们在查字典的时候，如果没有目录，那我们就只能一页一页的去找我们需要查的那个字，速度很慢。如果有目录了，我们只需要先去目录里查找字的位置，然后直接翻到那一页就行了。

虽然索引能带来查询上的效率，但是维护索引的成本也是不小的。如果一个字段不被经常查询，反而被经常修改，那么就更不应该在这种字段上建立索引了。

要选择选择合适的字段创建索引：

- **不为 NULL 的字段：**索引字段的数据应该尽量不为 NULL，因为对于数据为 NULL 的字段，数据库较难优化。如果字段频繁被查询，但又避免不了为 NULL，建议使用 0,1,true,false 这样语义较为清晰的短值或短字符作为替代。



- **被频繁查询的字段**：我们创建索引的字段应该是查询操作非常频繁的字段。
- **被作为条件查询的字段**：被作为 WHERE 条件查询的字段，应该被考虑建立索引。
- **频繁需要排序的字段**：索引已经排序，这样查询可以利用索引的排序，加快排序查询时间。
- **被经常频繁用于连接的字段**：经常用于连接的字段可能是一些外键列，对于外键列并不一定要建立外键，只是说该列涉及到表与表的关系。对于频繁被连接查询的字段，可以考虑建立索引，提高多表连接查询的效率。

使用表中的多个字段创建索引，就是 **联合索引**，也叫 **组合索引** 或 **复合索引**。

以 `score` 和 `name` 两个字段建立联合索引：

```
ALTER TABLE `cus_order` ADD INDEX id_score_name(score, name);
```

我们应该尽可能的考虑建立联合索引而不是单列索引。因为索引是需要占用磁盘空间的，可以简单理解为每个索引都对应着一颗 B+树。如果一个表的字段过多，索引过多，那么当这个表的数据达到一个体量后，索引占用的空间也是很多的，且修改索引时，耗费的时间也是较多的。如果是联合索引，多个字段在一个索引上，那么将会节约很大磁盘空间，且修改数据的操作效率也会提升。

## 缓存

### Redis 提供的数据类型有哪些？

Redis 中比较常见的数据类型有下面这些：

- **5 种基础数据类型**：String（字符串）、List（列表）、Set（集合）、Hash（散列）、Zset（有序集合）。
- **3 种特殊数据类型**：HyperLogLog（基数统计）、Bitmap（位图）、Geospatial（地理位置）。

除了上面提到的之外，还有一些其他的比如 [Bloom filter（布隆过滤器）](#)、[Bitfield（位域）](#)。

### String 的应用场景有哪些？底层实现是什么？

String 是 Redis 中最简单同时也是最常用的一个数据类型。它是一种二进制安全的数据类型，可以用来存储任何类型的数据比如字符串、整数、浮点数、图片（图片的 base64 编码或者解码或者图片的路径）、序列化后的对象。

String 的常见应用场景如下：

- 常规数据（比如 Session、Token、序列化后的对象、图片的路径）的缓存；
- 计数比如用户单位时间的请求数（简单限流可以用到）、页面单位时间的访问数；
- 分布式锁(利用 `SETNX key value` 命令可以实现一个最简易的分布式锁)；
- .....

Redis 是基于 C 语言编写的，但 Redis 的 String 类型的底层实现并不是 C 语言中的字符串（即以空字符 `\0` 结尾的字符数组），而是自己编写了 SDS（Simple Dynamic String，简单动态字符串）来作为底层实现。

SDS 最早是 Redis 作者为日常 C 语言开发而设计的 C 字符串，后来被应用到了 Redis 上，并经过了大量的修改完善以适合高性能操作。

Redis7.0 的 SDS 的部分源码如下 (<https://github.com/redis/redis/blob/7.0/src/sds.h>)：

```
/* Note: sdshdr5 is never used, we just access the flags byte directly.
 * However is here to document the layout of type 5 SDS strings. */
struct __attribute__((__packed__)) sdshdr5 {
    unsigned char flags; /* 3 lsb of type, and 5 msb of string length */
    char buf[];
};

struct __attribute__((__packed__)) sdshdr8 {
    uint8_t len; /* used */
    uint8_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};

struct __attribute__((__packed__)) sdshdr16 {
    uint16_t len; /* used */
    uint16_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};

struct __attribute__((__packed__)) sdshdr32 {
    uint32_t len; /* used */
    uint32_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};

struct __attribute__((__packed__)) sdshdr64 {
    uint64_t len; /* used */
    uint64_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};
```

通过源码可以看出，SDS 共有五种实现方式 SDS\_TYPE\_5（并未用到）、SDS\_TYPE\_8、SDS\_TYPE\_16、SDS\_TYPE\_32、SDS\_TYPE\_64，其中只有后四种实际用到。Redis 会根据初始化的长度决定使用哪种类型，从而减少内存的使用。

类型	字节	位
sdshdr5	< 1	<8
sdshdr8	1	8
sdshdr16	2	16
sdshdr32	4	32
sdshdr64	8	64

对于后四种实现都包含了下面这 4 个属性：

- `len` ：字符串的长度也就是已经使用的字节数
- `alloc` ：总共可用的字符空间大小，`alloc-len` 就是 SDS 剩余的空间大小
- `buf[]` ：实际存储字符串的数组
- `flags` ：低三位保存类型标志

SDS 相比于 C 语言中的字符串有如下提升：

1. **可以避免缓冲区溢出**：C 语言中的字符串被修改（比如拼接）时，一旦没有分配足够长度的内存空间，就会造成缓冲区溢出。SDS 被修改时，会先根据 `len` 属性检查空间大小是否满足要求，如果不满足，则先扩展至所需大小再进行修改操作。
2. **获取字符串长度的复杂度较低**：C 语言中的字符串的长度通常是经过遍历计数来实现的，时间复杂度为  $O(n)$ 。SDS 的长度获取直接读取 `len` 属性即可，时间复杂度为  $O(1)$ 。
3. **减少内存分配次数**：为了避免修改（增加/减少）字符串时，每次都需要重新分配内存（C 语言的字符串是这样的），SDS 实现了空间预分配和惰性空间释放两种优化策略。当 SDS 需要增加字符串时，Redis 会为 SDS 分配好内存，并且根据特定的算法分配多余的内存，这样可以减少连续执行字符串增长操作所需的内存重分配次数。当 SDS 需要减少字符串时，这部分内存不会立即被回收，会被记录下来，等待后续使用（支持手动释放，有对应的 API）。
4. **二进制安全**：C 语言中的字符串以空字符 `\0` 作为字符串结束的标识，这存在一些问题，像一些二进制文件（比如图片、视频、音频）就可能包括空字符，C 字符串无法正确保存。SDS 使用 `len` 属性判断字符串是否结束，不存在这个问题。

😬 多提一嘴，很多文章里 SDS 的定义是下面这样的：

```
struct sdshdr {
    unsigned int len;
    unsigned int free;
    char buf[];
};
```

这个也没错，Redis 3.2 之前就是这样定义的。后来，由于这种方式的定义存在问题，`len` 和 `free` 的定义用了 4 个字节，造成了浪费。Redis 3.2 之后，Redis 改进了 SDS 的定义，将其划分为了现在的 5 种类型。

## String 还是 Hash 存储对象数据更好呢？

- String 存储的是序列化后的对象数据，存放的是整个对象。Hash 是对对象的每个字段单独存储，可以获取部分字段的信息，也可以修改或者添加部分字段，节省网络流量。如果对象中某些字段需要经常变动或者经常需要单独查询对象中的个别字段信息，Hash 就非常适合。
- String 存储相对来说更加节省内存，缓存相同数量的对象数据，String 消耗的内存约是 Hash 的一半。并且，存储具有多层嵌套的对象时也方便很多。如果系统对性能和资源消耗非常敏感的话，String 就非常适合。

在绝大部分情况，我们建议使用 String 来存储对象数据即可！

## 多级缓存的是怎么做的？为什么还要再多加一层本地缓存呢？

这个问题的答案摘自《Java面试指北》（高质量原创 Java 面试小册）

我们这里只来简单聊聊 **本地缓存 + 分布式缓存** 的多级缓存方案，这也是最常用的多级缓存实现方式。

这个时候估计有很多小伙伴就会问了：**既然用了分布式缓存，为什么还要用本地缓存呢？**。

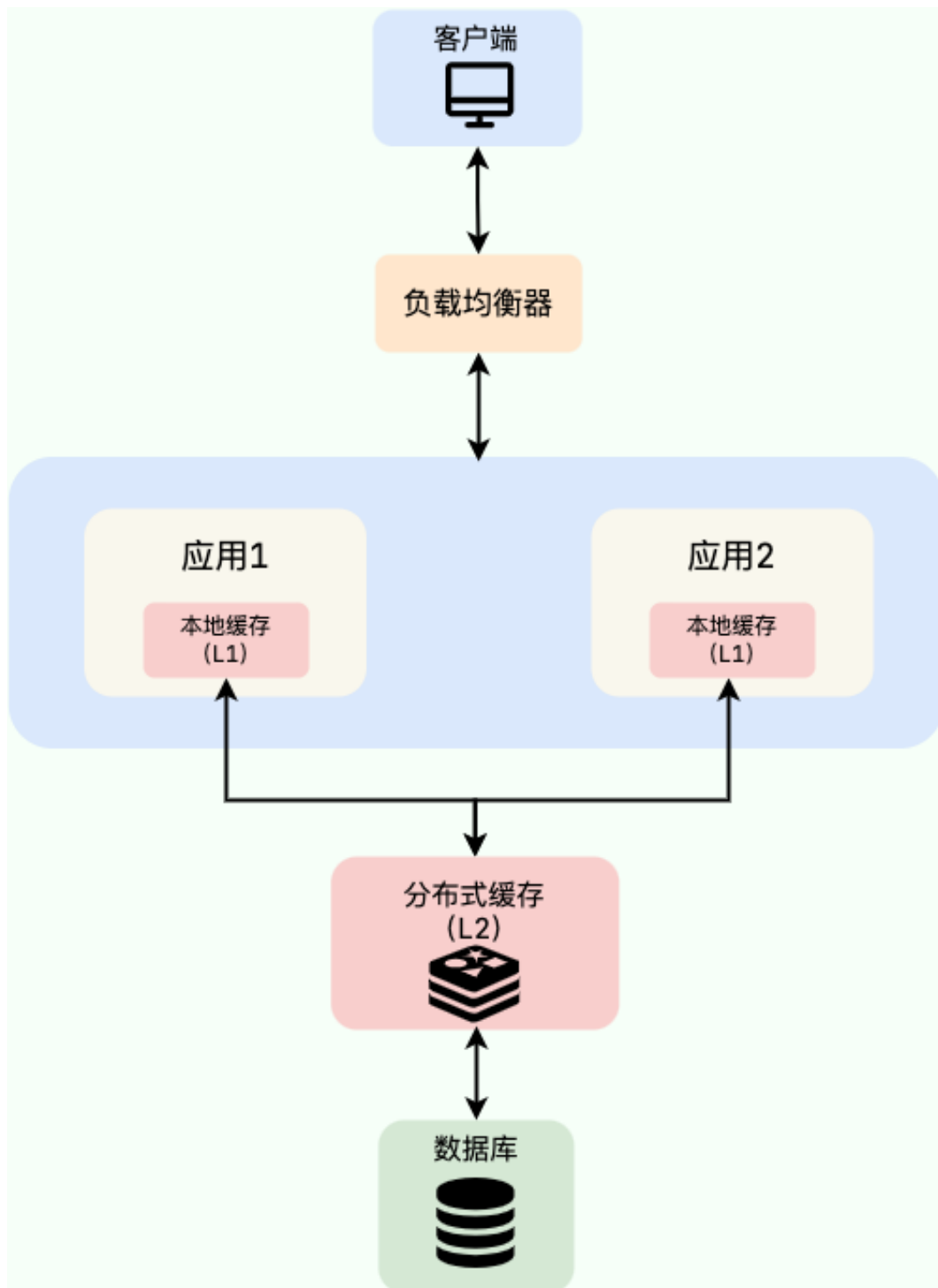
本地缓存和分布式缓存虽然都属于缓存，但本地缓存的访问速度要远大于分布式缓存，这是因为访问本地缓存不存在额外的网络开销，我们在上面也提到了。

不过，一般情况下，我们也是不建议使用多级缓存的，这会增加维护负担（比如你需要保证一级缓存和二级缓存的数据一致性）。而且，其实际带来的提升效果对于绝大部分业务场景来说其实并不是很大。

这里简单总结一下适合多级缓存的两种业务场景：

- 缓存的数据不会频繁修改，比较稳定；
- 数据访问量特别大比如秒杀场景。

多级缓存方案中，第一级缓存（L1）使用本地内存（比如 Caffeine），第二级缓存（L2）使用分布式缓存（比如 Redis）。



如果 L2 也没有此数据的话，再去数据库查询，数据查询成功后再将数据写入到 L1 和 L2 中。

[J2Cache](#) 就是一个基于本地内存和分布式缓存的两级 Java 缓存框架，感兴趣的同学可以研究一下。

## Redis 缓存穿透、缓存击穿、缓存雪崩区别和解决方案

内容较多，单独写了一篇文章详细介绍：[Redis 缓存穿透、缓存击穿、缓存雪崩区别和解决方案](#)

## 如何保证缓存和数据库数据的一致性？

细说的话可以扯很多，但是我觉得其实没太大必要（小声 BB：很多解决方案我也没太弄明白）。我个人觉得引入缓存之后，如果为了短时间的不一致性问题，选择让系统设计变得更加复杂的话，完全没必要。

下面单独对 **Cache Aside Pattern**（旁路缓存模式）来聊聊。

Cache Aside Pattern 中遇到写请求是这样的：更新 DB，然后直接删除 cache。

如果更新数据库成功，而删除缓存这一步失败的情况的话，简单说两个解决方案：

1. **缓存失效时间变短（不推荐，治标不治本）**：我们让缓存数据的过期时间变短，这样的话缓存就会从数据库中加载数据。另外，这种解决办法对于先操作缓存后操作数据库的场景不适用。
2. **增加 cache 更新重试机制（常用）**：如果 cache 服务当前不可用导致缓存删除失败的话，我们就隔一段时间进行重试，重试次数可以自己定。如果多次重试还是失败的话，我们可以把当前更新失败的 key 存入队列中，等缓存服务可用之后，再将缓存中对应的 key 删除即可。

详细介绍可以参考这篇文章：[缓存和数据库一致性问题，看这篇就够了 - 水滴与银弹](#)。

## 资料推荐

八股文资料首推 [《Java 面试指北》](#)（质量很高，专为面试打造，配合 JavaGuide 食用）。和 [JavaGuide](#)。里面不仅仅是原创八股文，还有很多对实际开发有帮助的干货。除了这两份资料之外，你还可以去网上找一些其他的优质的文章、视频来看。

# 《Java面试指北》

星球内部的《Java面试进阶指北》属于是 JavaGuide 的补充完善，两者配合使用！



☆ 收藏

目录

全部文档

目录管理

介绍	05-26 20:58
▸ 面试准备篇	10-14 15:11
▾ 技术面试题篇	03-21 19:03
▸ 系统设计	05-29 15:10
▸ Java	
▸ 数据库	
▸ 常见框架	
▸ 分布式	05-29 15:12
▸ 高并发	05-29 15:13
▸ 服务器	05-29 15:13
▸ Devops	05-29 15:15
▸ 技术面试题自测篇	05-29 15:16
▸ 面经篇	05-29 15:24
▸ 练级攻略篇	05-28 15:58
▸ 工作篇	08-14 19:12

一定不要抱着一种思想，觉得八股文或者基础问题的考查意义不大。如果你抱着这种思想复习的话，那效果可能不会太好。实际上，个人认为还是很有意义的，八股文或者基础性的知识在日常开发中也会需要经常用到。例如，线程池这块的拒绝策略、核心参数配置什么的，如果你不了解，实际项目中使用线程池可能就用的不是很明白，容易出现问题。而且，其实这种基础性的问题是最容易准备的，像各种底层原理、系统设计、场景题以及深挖你的项目这类才是最难的！

✈ JavaGuide 官方网站地址：[javaguide.cn](http://javaguide.cn)。

✈ [知识星球](#)（点击链接即可查看星球的详细介绍）包含的大部分资料都整理在了 [星球使用指南](#) 中（一定要看！！！！）。

星球专属资料概览：

🌟 知识星球的使用指南（必看）：

👉 专属资料：

星球目前一共有 6 个专栏（见图1《Java 面试指北》就在其中），均为星球专属，🔒 阅读地址：

🔗 下面是星球提供的一些专栏（目前...（会定期修改密码，避免盗版传播。只要购买过一次星球，即可永久找我获取最新密码）。

项目资料合集（持续更新中）：🔗 实战项目资料合集。

除了这些专栏之外，星球还有下面这些常用的优质 PDF 技术资源：

- 🔒 原创 PDF 面试资料（选择自己需要的即可）：🔗 原创PDF面试资料（内容很全面...。
- 🔒 Java 面试常见问题总结（2024 最新版，用于自测）：🔗 Java面试常见问题总结（20...
- 高频笔试题（非常规Leetcode类型）PDF：🔗 非常规 Leetcode 类型的高频笔试题
- 20 道 HR 面常见问题：🔗 20道HR面常见问题
- 线上常见问题案例和排查工具：🔗 进一步完善了之前整理的线上常见...

星球提供的资料可能无法满足每一位球友的期待，如果你有其他迫切需要的内容，欢迎微信联系我，给我留言，我会尽力为你提供帮助！






1、《Java面试指北》(配合 JavaGuide 使用, 会根据每一年的面试情况对内容进行更新完善, 故不提供 PDF 版本): <https://www.yuque.com/books/share/04ac99ea-7726-4a...> (密码: )

JavaGuide 地址: <https://javaguide.cn/>, 《Java 面试指北》的学习建议在这里: <https://t.zsxq.com/QNFMFAU>。如果不知道《Java面试指北》和开源版的关系, 可以看看这份建议。

2、《后端面试高频系统设计&场景题》: <https://www.yuque.com/snailclimb/tangw3> 密码: 

这部分内容本身是属于《Java面试指北》的, 后面由于内容篇幅较多, 因此被单独提了出来。

3、《Java 必读源码系列》(目前已经整理了 Dubbo 2.6.x、Netty 4.x、SpringBoot2.1 的源码): <https://www.yuque.com/books/share/7f846c65-f32e-41...> (密码: )

欢迎在评论区说出你们想要看的框架/中间件的源码!


4. 《从零开始写一个RPC框架》: <https://www.yuque.com/books/share/b7a2512c-6f7a-4a...> (密码: )

RPC 框架地址: <https://gitee.com/SnailClimb/guide-rpc-framework>。

5、《分布式、高并发、Devops 面试扫盲》: 已经并入《Java面试指北》中。

6、《Kafka常见面试题/知识点总结》: <https://www.yuque.com/books/share/dd07d89b-9437-4f...> (密码: )

7、《程序员副业赚钱之路》 <https://www.yuque.com/books/share/1bd77211-f7e0-41...> (密码: )

8、《Guide的读书笔记与文章精选集》(经典书籍精读笔记分享) <https://www.yuque.com/books/share/f63faff5-53f9-41...> (密码: )



### 星球部分面试资料介绍：

- [《Java 面试指北》](#)（面试专版，Java面试必备）
- [《后端面试高频系统设计&场景题》](#)（20+高频系统设计&场景面试题）
- [《Java 必读源码系列》](#)（目前已经整理了 Dubbo 2.6.x 、 Netty 4.x、SpringBoot2.1 的源码）
- [《后端高频笔试题（非常规Leetcode类型）》](#)（新增多线程相关的手撕题）
- [《Java 面试常见问题总结（2024 最新版）》](#)（350+ 道超高频面试题总结）
- [20 道 HR 面的常见问题](#)（HR 面必备）

### 星球项目资源：

- [网盘项目](#)（网盘项目通用的介绍模板、优化思路以及面试知识点考察分析）
- [手写 RPC 框架](#)（如何从零开始基于 Netty+Kyro+Zookeeper 实现一个简易的 RPC 框架）