

公号已发：<https://mp.weixin.qq.com/s/KsbJMOXPiZ5VkQEdFy5hvA>

这篇文章的面试问题来源于一个读者之前分享的[美团暑期实习面经](#)。在这篇文章中，他分享了自己备战面试的经历和在美团实习的感受，内容挺不错的。

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！

ps:由于内容过多，本文只整理Java相关的面试问题。

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 基础篇

### 接口和抽象类的区别

共同点：

- 都不能被实例化。
- 都可以包含抽象方法。
- 都可以有默认实现的方法（Java 8 可以用 `default` 关键字在接口中定义默认方法）。

区别：

- 接口主要用于对类的行为进行约束，你实现了某个接口就具有了对应的行为。抽象类主要用于代码复用，强调的是所属关系。
- 一个类只能继承一个类，但是可以实现多个接口。
- 接口中的成员变量只能是 `public static final` 类型的，不能被修改且必须有初始值，而抽象类的成员变量默认 `default`，可在子类中被重新定义，也可被重新赋值。

### String、StringBuffer、StringBuilder 的区别

可变性

`String` 是不可变的（后面会详细分析原因）。

`StringBuilder` 与 `StringBuffer` 都继承自 `AbstractStringBuilder` 类，在 `AbstractStringBuilder` 中也是使用字符数组保存字符串，不过没有使用 `final` 和 `private` 关键字修饰，最关键的是这个 `AbstractStringBuilder` 类还提供了很多修改字符串的方法比如 `append` 方法。

```

abstract class AbstractStringBuilder implements Appendable, CharSequence {
    char[] value;

    public AbstractStringBuilder append(String str) {
        if (str == null)
            return appendNull();

        int len = str.length();
        ensureCapacityInternal(count + len);
        str.getChars(0, len, value, count);

        count += len;
        return this;
    }
    //...
}

```

## 线程安全性

`String` 中的对象是不可变的，也就可以理解为常量，线程安全。`AbstractStringBuilder` 是 `StringBuilder` 与 `StringBuffer` 的公共父类，定义了一些字符串的基本操作，如 `expandCapacity`、`append`、`insert`、`indexOf` 等公共方法。`StringBuffer` 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。`StringBuilder` 并没有对方法进行加同步锁，所以是非线程安全的。

## 性能

每次对 `String` 类型进行改变的时候，都会生成一个新的 `String` 对象，然后将指针指向新的 `String` 对象。`StringBuffer` 每次都会对 `StringBuffer` 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用 `StringBuilder` 相比使用 `StringBuffer` 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

## 反射的应用场景

像咱们平时大部分时候都是在写业务代码，很少会接触到直接使用反射机制的场景。但是！这并不代表反射没有用。相反，正是因为反射，你才能这么轻松地使用各种框架。像 Spring/Spring Boot、MyBatis 等等框架中都大量使用了反射机制。

这些框架中也大量使用了动态代理，而动态代理的实现也依赖反射。

比如下面是通过 JDK 实现动态代理的示例代码，其中就使用了反射类 `Method` 来调用指定的方法。

```

public class DebugInvocationHandler implements InvocationHandler {
    /**
     * 代理类中的真实对象
     */
    private final Object target;
}

```

```

    public DebugInvocationHandler(Object target) {
        this.target = target;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws
    InvocationTargetException, IllegalAccessException {
        System.out.println("before method " + method.getName());
        Object result = method.invoke(target, args);
        System.out.println("after method " + method.getName());
        return result;
    }
}

```

另外，像 Java 中的一大利器 **注解** 的实现也用到了反射。

为什么你使用 Spring 的时候，一个 `@Component` 注解就声明了一个类为 Spring Bean 呢？为什么你通过一个 `@Value` 注解就读取到配置文件中的值呢？究竟是怎么起作用的呢？

这些都是因为你可以基于反射分析类，然后获取到类/属性/方法/方法的参数上的注解。你获取到注解之后，就可以做进一步的处理。

## HashMap 源码实现，链表转红黑树条件，红黑树是否会退化为链表

### JDK1.8 之前

JDK1.8 之前 `HashMap` 底层是 **数组和链表** 结合在一起使用也就是 **链表散列**。`HashMap` 通过 key 的 `hashCode` 经过扰动函数处理过后得到 hash 值，然后通过 `(n - 1) & hash` 判断当前元素存放的位置（这里的 n 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。

所谓扰动函数指的就是 `HashMap` 的 `hash` 方法。使用 `hash` 方法也就是扰动函数是为了防止一些实现比较差的 `hashCode()` 方法 换句话说使用扰动函数之后可以减少碰撞。

### JDK 1.8 `HashMap` 的 `hash` 方法源码:

JDK 1.8 的 `hash` 方法 相比于 JDK 1.7 `hash` 方法更加简化，但是原理不变。

```

static final int hash(Object key) {
    int h;
    // key.hashCode(): 返回散列值也就是hashcode
    // ^: 按位异或
    // >>>: 无符号右移, 忽略符号位, 空位都以0补齐
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}

```

对比一下 JDK1.7 的 HashMap 的 hash 方法源码.

```

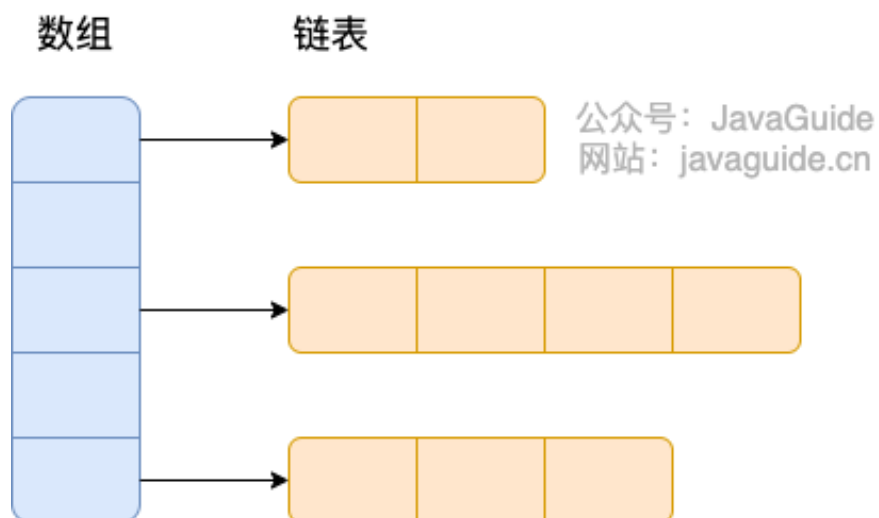
static int hash(int h) {
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).

    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}

```

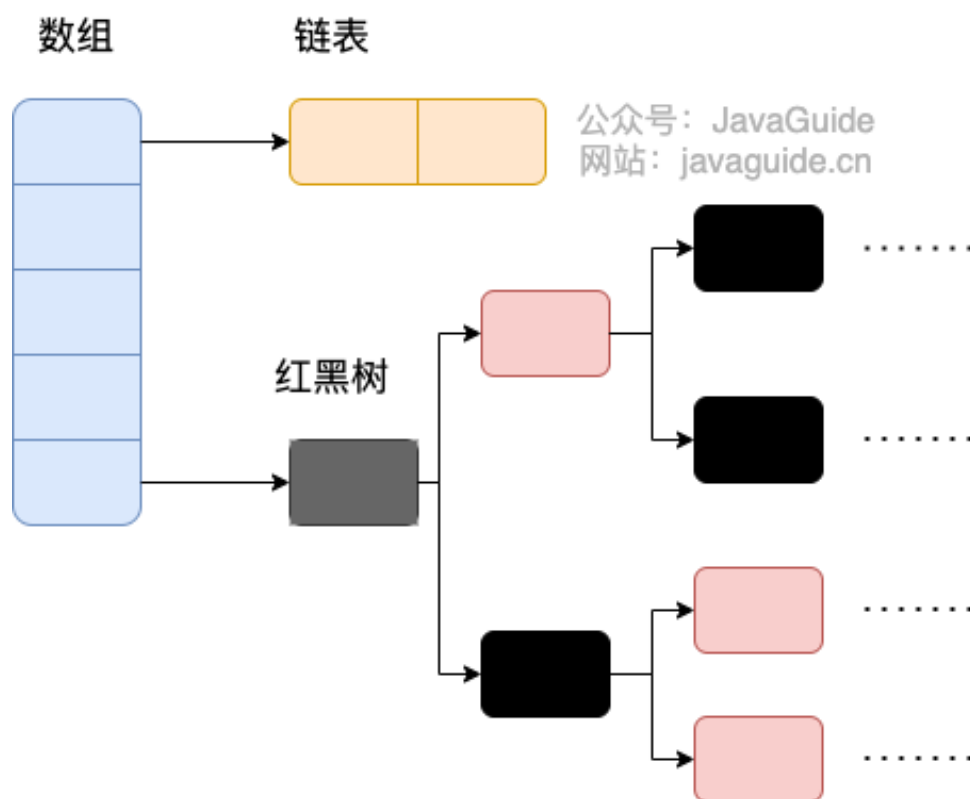
相比于 JDK1.8 的 hash 方法, JDK 1.7 的 hash 方法的性能会稍差一点点, 因为毕竟扰动了 4 次。

所谓“**拉链法**”就是: 将链表和数组相结合。也就是说创建一个链表数组, 数组中每一格就是一个链表。若遇到哈希冲突, 则将冲突的值加到链表中即可。



## JDK1.8 之后

相比于之前的版本，JDK1.8 之后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间。



TreeMap、TreeSet 以及 JDK1.8 之后的 HashMap 底层都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

我们来结合源码分析一下 HashMap 链表到红黑树的转换。

### 1、putVal 方法中执行链表转红黑树的判断逻辑。

链表的长度大于 8 的时候，就执行 treeifyBin（转换红黑树）的逻辑。

```
// 遍历链表
for (int binCount = 0; ; ++binCount) {
    // 遍历到链表最后一个节点
    if ((e = p.next) == null) {
        p.next = newNode(hash, key, value, null);
        // 如果链表元素个数大于等于TREEIFY_THRESHOLD (8)
        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
            // 红黑树转换（并不会直接转换成红黑树）
            treeifyBin(tab, hash);
    }
}
```

```

        break;
    }
    if (e.hash == hash &&
        ((k = e.key) == key || (key != null && key.equals(k))))
        break;
    p = e;
}

```

## 2、treeifyBin 方法中判断是否真的转换为红黑树。

```

final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    // 判断当前数组的长度是否小于 64
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        // 如果当前数组的长度小于 64，那么会选择先进行数组扩容
        resize();
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        // 否则才将列表转换为红黑树

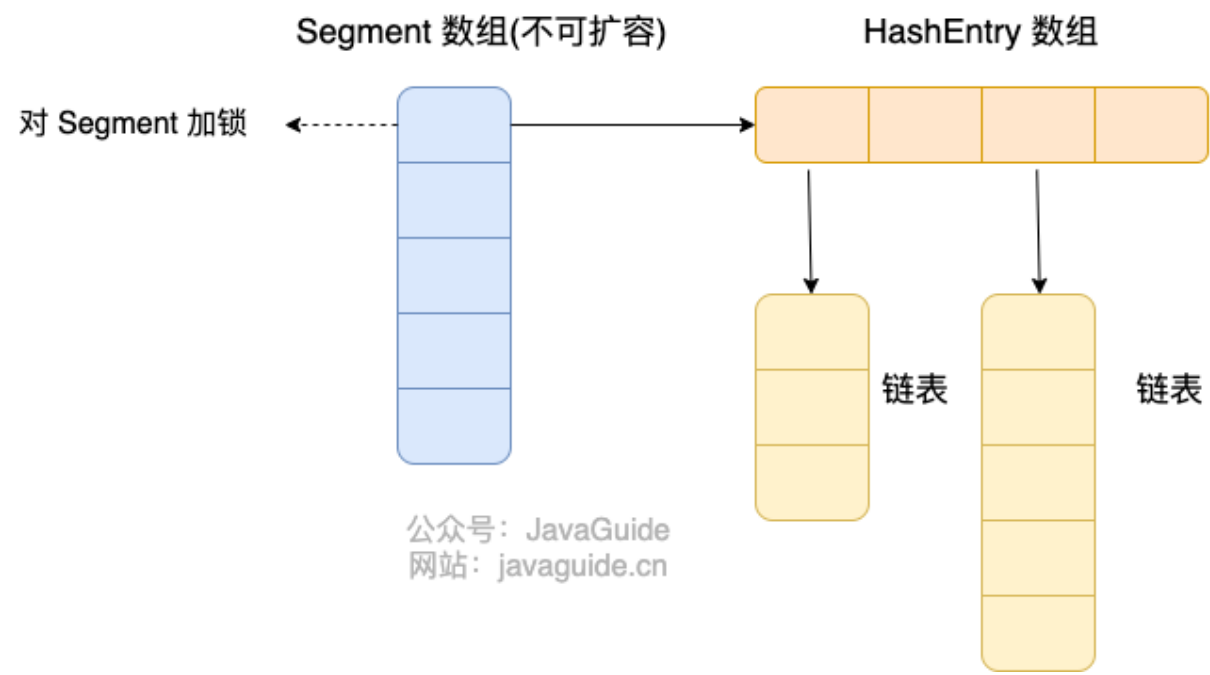
        TreeNode<K,V> hd = null, tl = null;
        do {
            TreeNode<K,V> p = replacementTreeNode(e, null);
            if (tl == null)
                hd = p;
            else {
                p.prev = tl;
                tl.next = p;
            }
            tl = p;
        } while ((e = e.next) != null);
        if ((tab[index] = hd) != null)
            hd.treeify(tab);
    }
}

```

将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树。

# ConcurrentHashMap 的底层实现

## JDK1.8 之前



首先将数据分为一段一段（这个“段”就是 Segment）的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据时，其他段的数据也能被其他线程访问。

ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。

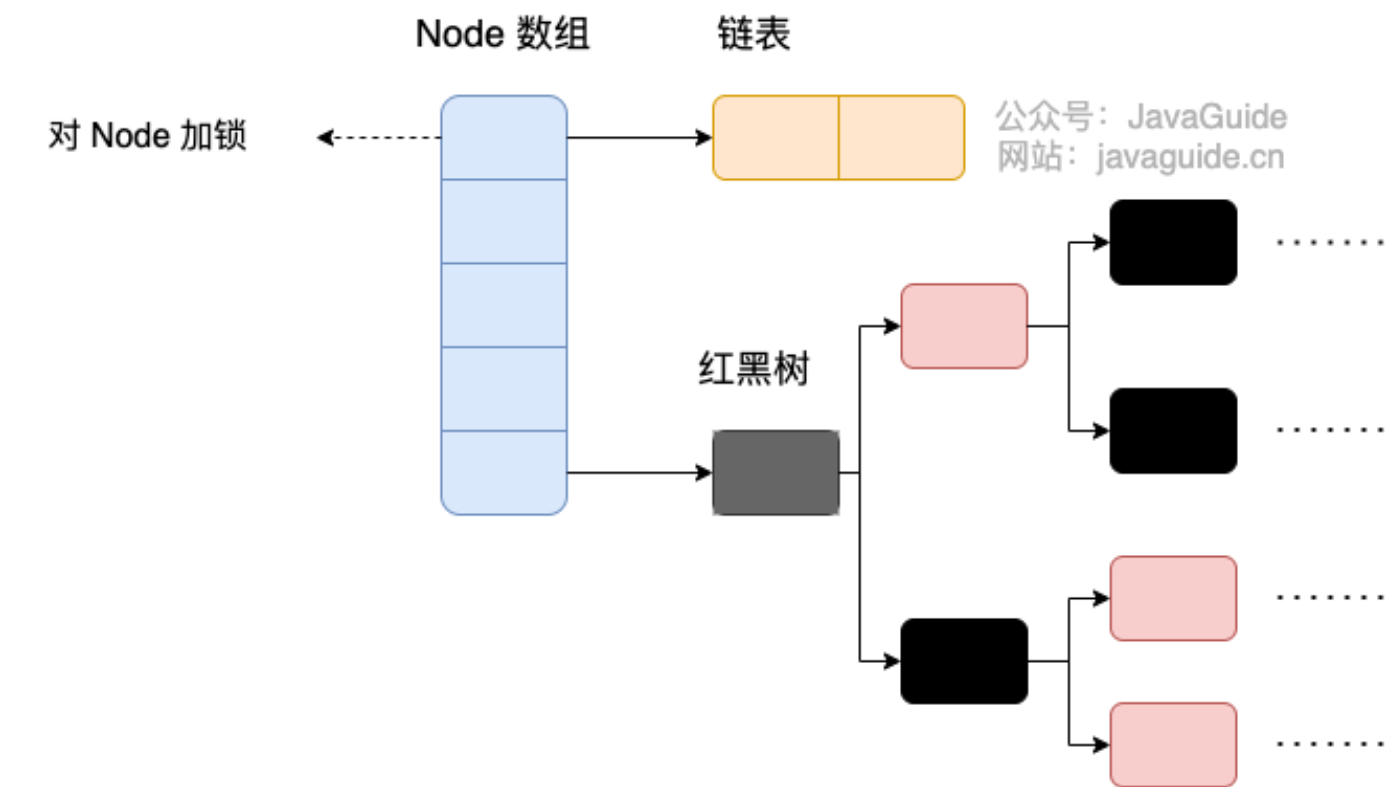
Segment 继承了 ReentrantLock，所以 Segment 是一种可重入锁，扮演锁的角色。HashEntry 用于存储键值对数据。

```
static class Segment<K,V> extends ReentrantLock implements Serializable {  
    ...  
}
```

一个 ConcurrentHashMap 里包含一个 Segment 数组，Segment 的个数一旦初始化就不能改变。Segment 数组的大小默认是 16，也就是说默认可以同时支持 16 个线程并发写。

Segment 的结构和 HashMap 类似，是一种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，每个 Segment 守护着一个 HashEntry 数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 的锁。也就是说，对同一 Segment 的并发写入会被阻塞，不同 Segment 的写入是可以并发执行的。

## JDK1.8 之后



Java 8 几乎完全重写了 `ConcurrentHashMap`，代码量从原来 Java 7 中的 1000 多行，变成了现在的 6000 多行。

`ConcurrentHashMap` 取消了 `Segment` 分段锁，采用 `Node + CAS + synchronized` 来保证并发安全。数据结构跟 `HashMap 1.8` 的结构类似，数组+链表/红黑二叉树。Java 8 在链表长度超过一定阈值（8）时将链表（寻址时间复杂度为  $O(N)$ ）转换为红黑树（寻址时间复杂度为  $O(\log(N))$ ）。

Java 8 中，锁粒度更细，`synchronized` 只锁定当前链表或红黑二叉树的首节点，这样只要 hash 不冲突，就不会产生并发，就不会影响其他 Node 的读写，效率大幅提升。

## Java 多线程

### 预防和避免线程死锁

如何预防死锁？破坏死锁的产生的必要条件即可：

1. 破坏请求与保持条件：一次性申请所有的资源。
2. 破坏不剥夺条件：占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。
3. 破坏循环等待条件：靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。

如何避免死锁？



避免死锁就是在资源分配时，借助于算法（比如银行家算法）对资源分配进行计算评估，使其进入安全状态。

**安全状态** 指的是系统能够按照某种线程推进顺序（P1、P2、P3.....Pn）来为每个线程分配所需资源，直到满足每个线程对资源的最大需求，使每个线程都可顺利完成。称  $\langle P1、P2、P3.....Pn \rangle$  序列为安全序列。

## 简述 CAS

CAS 的全称是 **Compare And Swap（比较与交换）**，用于实现乐观锁，被广泛应用于各大框架中。CAS 的思想很简单，就是用一个预期值和要更新的变量值进行比较，两值相等才会进行更新。

CAS 是一个原子操作，底层依赖于一条 CPU 的原子指令。

**原子操作** 即最小不可拆分的操作，也就是说操作一旦开始，就不能被打断，直到操作完成。

CAS 涉及到三个操作数：

- **V**：要更新的变量值(Var)
- **E**：预期值(Expected)
- **N**：拟写入的新值(New)

当且仅当 V 的值等于 E 时，CAS 通过原子方式用新值 N 来更新 V 的值。如果不等，说明已经有其它线程更新了 V，则当前线程放弃更新。

**举一个简单的例子：**线程 A 要修改变量 i 的值为 6，i 原值为 1（V = 1，E=1，N=6，假设不存在 ABA 问题）。

1. i 与 1 进行比较，如果相等，则说明没被其他线程修改，可以被设置为 6。
2. i 与 1 进行比较，如果不相等，则说明被其他线程修改，当前线程放弃更新，CAS 操作失败。

当多个线程同时使用 CAS 操作一个变量时，只有一个会胜出，并成功更新，其余均会失败，但失败的线程并不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。

Java 语言并没有直接实现 CAS，CAS 相关的实现是通过 C++ 内联汇编的形式实现的（JNI 调用）。因此，CAS 的具体实现和操作系统以及 CPU 都有关系。

sun.misc 包下的 Unsafe 类提供了 compareAndSwapObject、compareAndSwapInt、compareAndSwapLong 方法来实现的对 Object、int、long 类型的 CAS 操作

```

/**
 * CAS
 * @param o      包含要修改field的对象
 * @param offset 对象中某field的偏移量
 * @param expected 期望值
 * @param update 更新值
 * @return      true | false
 */
public final native boolean compareAndSwapObject(Object o, long offset, Object expected, Object
update);

public final native boolean compareAndSwapInt(Object o, long offset, int expected,int update);

public final native boolean compareAndSwapLong(Object o, long offset, long expected, long
update);

```

## 乐观锁和悲观锁

### 什么是悲观锁？

悲观锁总是假设最坏的情况，认为共享资源每次被访问的时候就会出现问題(比如共享数据被修改)，所以每次在获取资源操作的时候都会上锁，这样其他线程想拿到这个资源就会阻塞直到锁被上一个持有者释放。也就是说，共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程。

像 Java 中 `synchronized` 和 `ReentrantLock` 等独占锁就是悲观锁思想的实现。

```

public void performSynchronisedTask() {
    synchronized (this) {
        // 需要同步的操作
    }
}

private Lock lock = new ReentrantLock();
lock.lock();
try {
    // 需要同步的操作
} finally {
    lock.unlock();
}

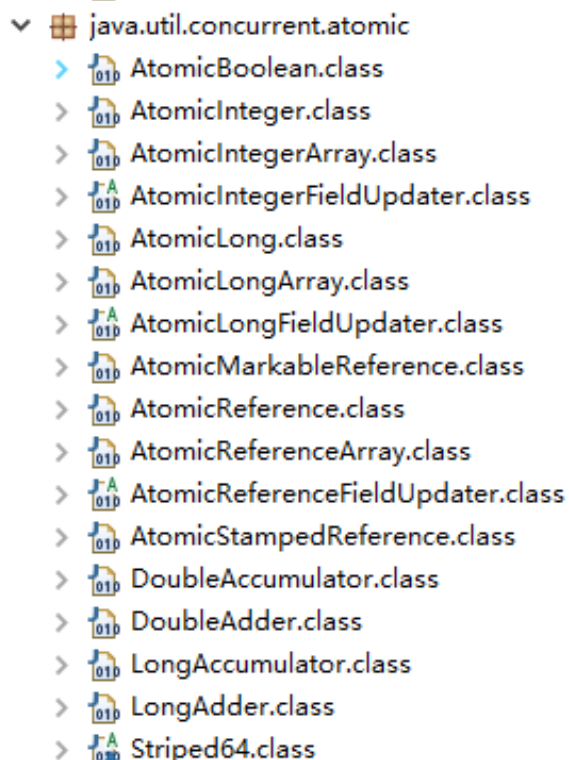
```

高并发的场景下，激烈的锁竞争会造成线程阻塞，大量阻塞线程会导致系统的上下文切换，增加系统的性能开销。并且，悲观锁还可能会存在死锁问题，影响代码的正常运行。

## 什么是乐观锁？

乐观锁总是假设最好的情况，认为共享资源每次被访问的时候不会出现问题，线程可以不停地执行，无需加锁也无需等待，只是在提交修改的时候去验证对应的资源（也就是数据）是否被其它线程修改了（具体方法可以使用版本号机制或 CAS 算法）。

在 Java 中 `java.util.concurrent.atomic` 包下面的原子变量类（比如 `AtomicInteger`、`LongAdder`）就是使用了乐观锁的一种实现方式 **CAS** 实现的。



```
// LongAdder 在高并发场景下会比 AtomicInteger 和 AtomicLong 的性能更好
// 代价就是会消耗更多的内存空间（空间换时间）

LongAdder sum = new LongAdder();

sum.increment();
```

高并发的场景下，乐观锁相比悲观锁来说，不存在锁竞争造成线程阻塞，也不会有死锁的问题，在性能上往往会更胜一筹。但是，如果冲突频繁发生（写占比非常多的情况），会频繁失败和重试，这样同样会非常影响性能，导致 CPU 飙升。

不过，大量失败重试的问题也是可以解决的，像我们前面提到的 `LongAdder` 以空间换时间的方式就解决了这个问题。

理论上来说：

- 悲观锁通常多用于写比较多的情况（多写场景，竞争激烈），这样可以避免频繁失败和重试影响性能，悲观锁的开销是固定的。不过，如果乐观锁解决了频繁失败和重试这个问题的话（比如 `LongAdder` ），也是可以考虑使用乐观锁的，要视实际情况而定。
- 乐观锁通常多用于写比较少少的情况（多读场景，竞争较少），这样可以避免频繁加锁影响性能。不过，乐观锁主要针对的对象是单个共享变量（参考 `java.util.concurrent.atomic` 包下面的原子变量类）。

## Java 线程池的几个参数以及其具体意义

```
/**
 * 用给定的初始参数创建一个新的ThreadPoolExecutor。
 */
public ThreadPoolExecutor(int corePoolSize, //线程池的核心线程数量
                          int maximumPoolSize, //线程池的最大线程数
                          long keepAliveTime, //当线程数大于核心线程数时，多余的空闲线程存活的最长时间
                          TimeUnit unit, //时间单位
                          BlockingQueue<Runnable> workQueue, //任务队列，用来储存等待执行任务的队列
                          ThreadFactory threadFactory, //线程工厂，用来创建线程，一般默认即可
                          RejectedExecutionHandler handler //拒绝策略，当提交的任务过多而不能及时处
    理时，我们可以定制策略来处理任务
) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

`ThreadPoolExecutor` 3 个最重要的参数：

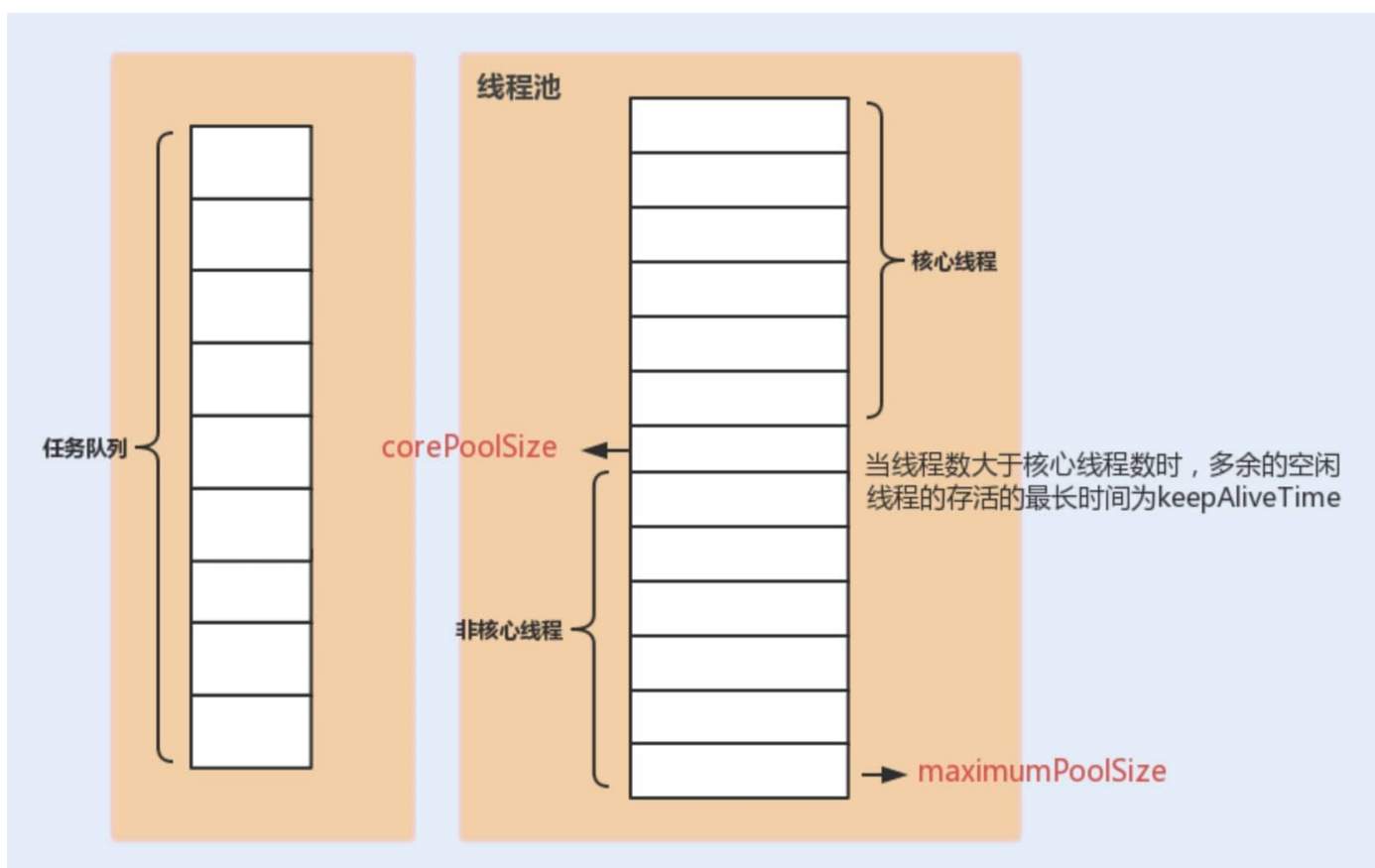
- `corePoolSize` ：任务队列未达到队列容量时，最大可以同时运行的线程数量。
- `maximumPoolSize` ：任务队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数。

- `workQueue` : 新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，新任务就会被存放在队列中。

`ThreadPoolExecutor` 其他常见参数：

- `keepAliveTime` : 线程池中的线程数量大于 `corePoolSize` 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 `keepAliveTime` 才会被回收销毁。
- `unit` : `keepAliveTime` 参数的时间单位。
- `threadFactory` : `executor` 创建新线程的时候会用到。
- `handler` : 饱和策略（后面会单独详细介绍一下）。

下面这张图可以加深你对线程池中各个参数的相互关系的理解（图片来源：《Java 性能调优实战》）：



当任务队列未滿时，最多可以同时运行的线程数量就是核心线程数。任务队列中存放的任务满了之后，最多可以同时运行的线程数量就是最大线程数。

## 线程池的拒绝策略

如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任务时，`ThreadPoolExecutor` 定义一些策略：

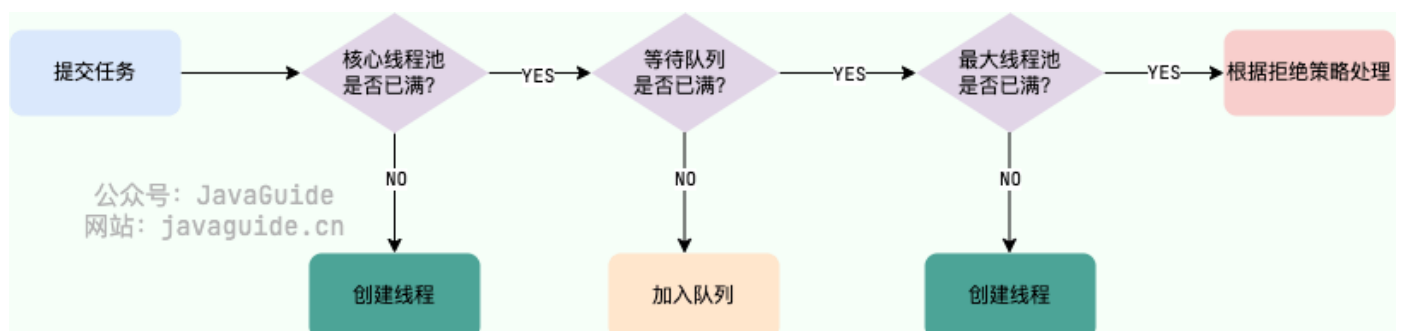
- `ThreadPoolExecutor.AbortPolicy` : 抛出 `RejectedExecutionException` 来拒绝新任务的处理。

- `ThreadPoolExecutor.CallerRunsPolicy` : 调用执行自己的线程运行任务，也就是直接在调用 `execute` 方法的线程中运行( `run` )被拒绝的任务，如果执行程序已关闭，则会丢弃该任务。因此这种策略会降低对于新任务提交速度，影响程序的整体性能。如果你的应用程序可以承受此延迟并且你要求任何一个任务请求都要被执行的话，你可以选择这个策略。
- `ThreadPoolExecutor.DiscardPolicy` : 不处理新任务，直接丢弃掉。
- `ThreadPoolExecutor.DiscardOldestPolicy` : 此策略将丢弃最早的未处理的任务请求。

举个例子：Spring 通过 `ThreadPoolTaskExecutor` 或者我们直接通过 `ThreadPoolExecutor` 的构造函数创建线程池的时候，当我们不指定 `RejectedExecutionHandler` 拒绝策略来配置线程池的时候，默认使用的是 `AbortPolicy` 。在这种拒绝策略下，如果队列满了，`ThreadPoolExecutor` 将抛出 `RejectedExecutionException` 异常来拒绝新来的任务，这代表你将丢失对这个任务的处理。如果不想丢弃任务的话，可以使用 `CallerRunsPolicy` 。`CallerRunsPolicy` 和其他的几个策略不同，它既不会抛弃任务，也不会抛出异常，而是将任务回退给调用者，使用调用者的线程来执行任务。

```
public static class CallerRunsPolicy implements RejectedExecutionHandler {  
  
    public CallerRunsPolicy() { }  
  
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
        if (!e.isShutdown()) {  
            // 直接主线程执行，而不是线程池中的线程执行  
            r.run();  
        }  
    }  
}
```

## 描述线程池的执行过程



1. 如果当前运行的线程数小于核心线程数，那么就会新建一个线程来执行任务。
2. 如果当前运行的线程数等于或大于核心线程数，但是小于最大线程数，那么就把该任务放入到任务队列里等待执行。
3. 如果向任务队列投放任务失败（任务队列已经满了），但是当前运行的线程数是小于最大线程数的，就新建一个线程来执行任务。

4. 如果当前运行的线程数已经等同于最大线程数了，新建线程将会使当前运行的线程超出最大线程数，那么当前任务会被拒绝，饱和策略会调用 `RejectedExecutionHandler.rejectedExecution()` 方法。

## JVM 篇

### 简述双亲委派模型

什么是双亲委派模型？

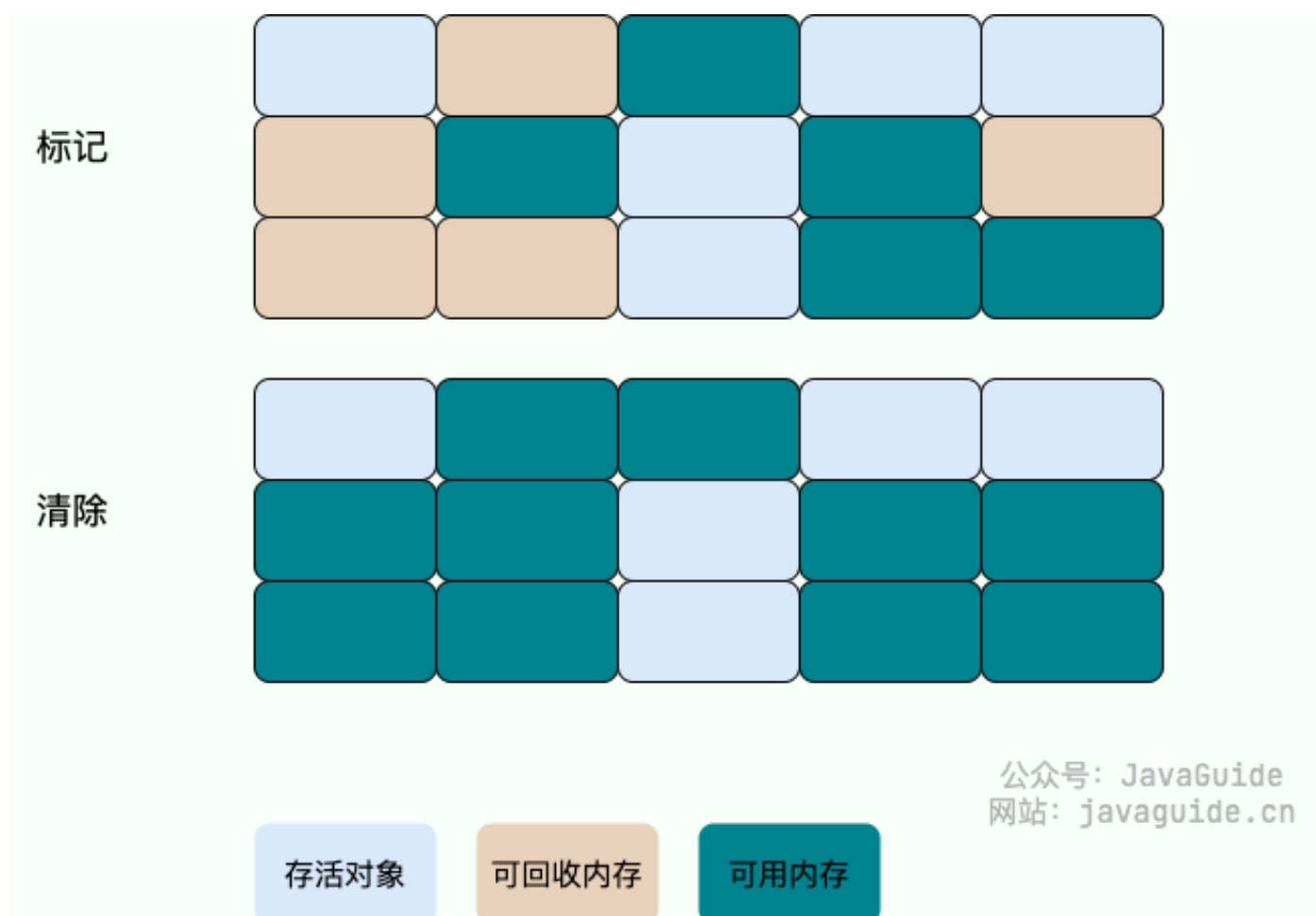
### 垃圾回收算法有哪些

#### 标记-清除算法

标记-清除（Mark-and-Sweep）算法分为“标记（Mark）”和“清除（Sweep）”阶段：首先标记出所有不需要回收的对象，在标记完成后统一回收掉所有没有被标记的对象。

它是最基础的收集算法，后续的算法都是对其不足进行改进得到。这种垃圾收集算法会带来两个明显的问题：

1. **效率问题**：标记和清除两个过程效率都不高。
2. **空间问题**：标记清除后会产生大量不连续的内存碎片。



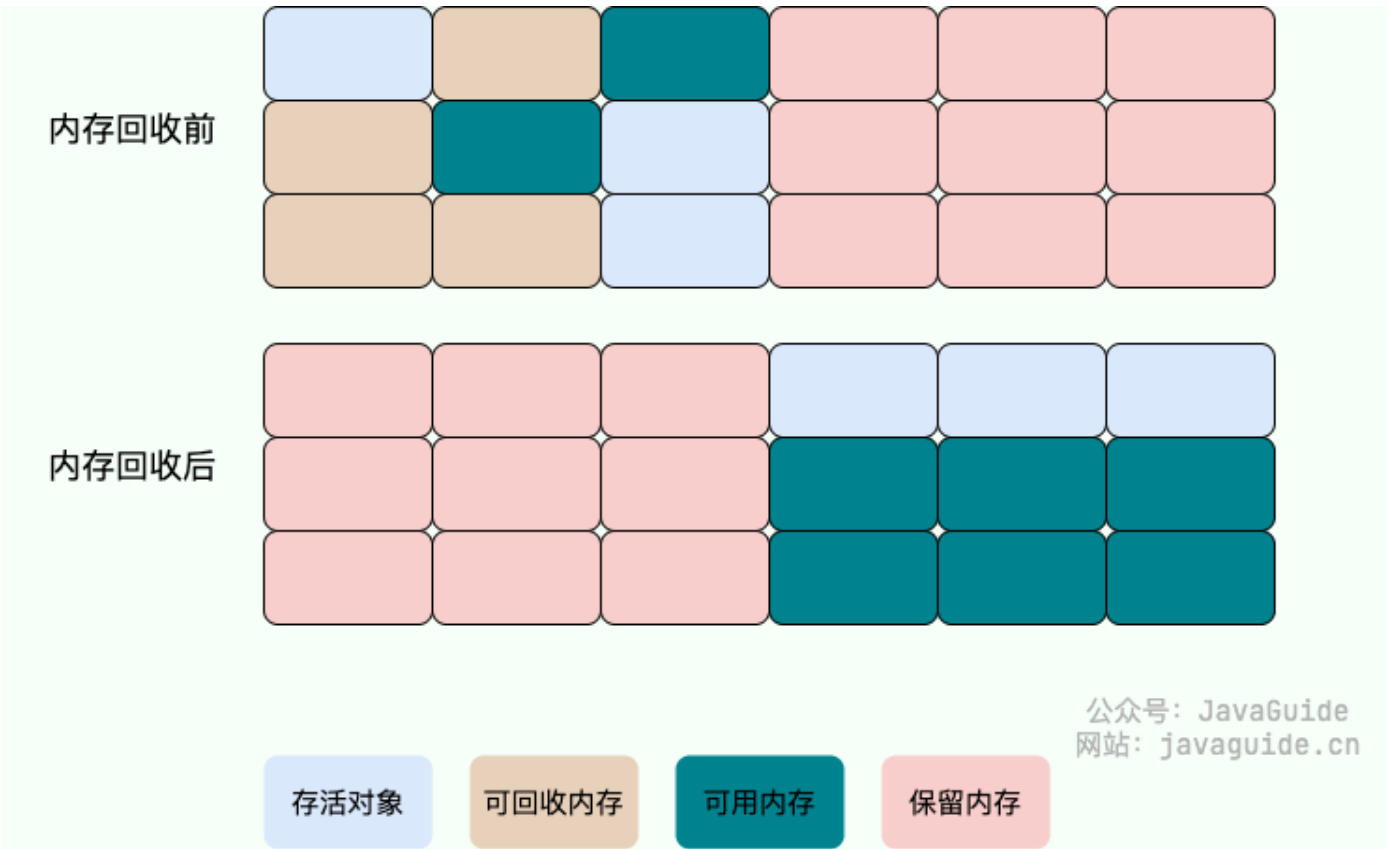
关于具体是标记可回收对象还是不可回收对象，众说纷纭，两种说法其实都没问题，我个人更倾向于是前者。

如果按照前者的理解，整个标记-清除过程大致是这样的：

- 1. 当一个对象被创建时，给一个标记位，假设为 0 (false)；
- 2. 在标记阶段，我们将所有可达对象（或用户可以引用的对象）的标记位设置为 1 (true)；
- 3. 扫描阶段清除的就是标记位为 0 (false)的对象。

## 复制算法

为了解决标记-清除算法的效率和内存碎片问题，复制（Copying）收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。



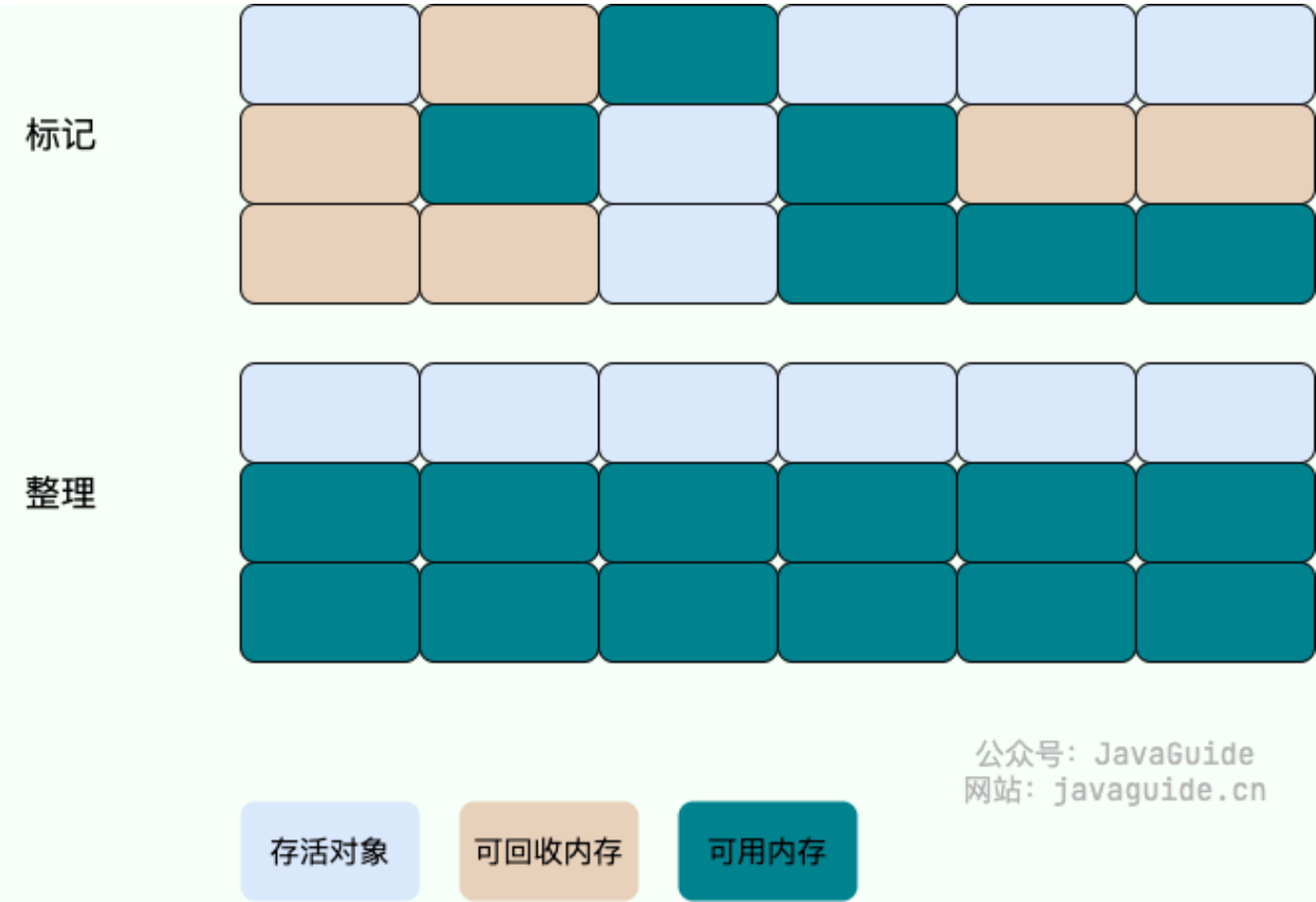
虽然改进了标记-清除算法，但依然存在下面这些问题：

- 可用内存变小：可用内存缩小为原来的一半。
- 不适合老年代：如果存活对象数量比较大，复制性能会变得很差。



## 标记-整理算法

标记-整理（Mark-and-Compact）算法是根据老年代的特点提出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。



由于多了整理这一步，因此效率也不高，适合老年代这种垃圾回收频率不是很高的场景。

## 分代收集算法

当前虚拟机的垃圾收集都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将 Java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

比如在新生代中，每次收集都会有大量对象死去，所以可以选择“标记-复制”算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。

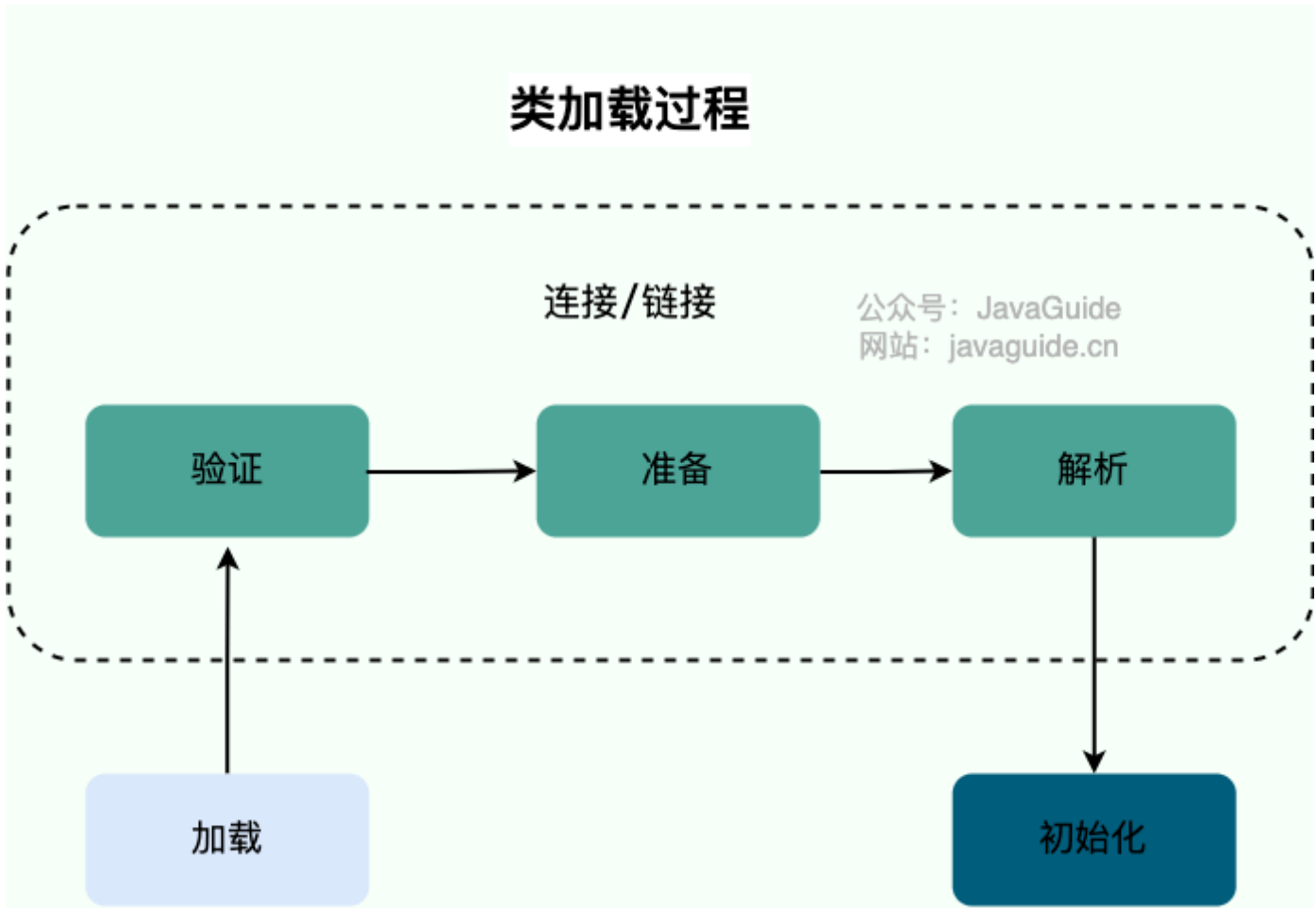
**延伸面试问题：** HotSpot 为什么要分为新生代和老年代？

根据上面的对分代收集算法的介绍回答。

## 简述类加载过程

Class 文件需要加载到虚拟机中之后才能运行和使用，那么虚拟机是如何加载这些 Class 文件呢？

系统加载 Class 类型的文件主要三步：加载->连接->初始化。连接过程又可分为三步：验证->准备->解析。



## 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）