《Java 性能优化实战 21 讲》

李国 前京东、陌陌高级架构师

— 拉勾教育出品 —

07｜案例分析：无处不在的缓存
高并发系统的法宝

案例分析：无处不在的缓存，高并发系统的法宝

在最核心的 CPU 中，存在多级缓存

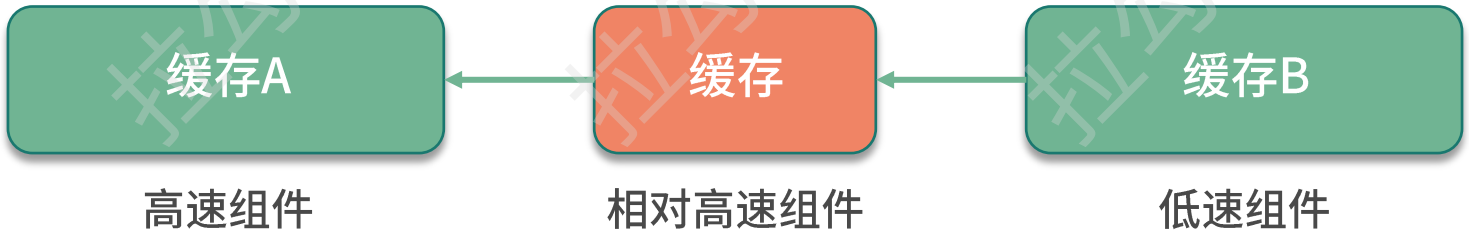
类似 Redis 的缓存框架可以消除内存和存储之间的差异

缓存的优化效果：

- 让原本载入非常缓慢的页面，瞬间秒开
- 让本是压力山大的数据库，瞬间清闲下来

案例分析：无处不在的缓存，高并发系统的法宝

缓存本质——协调两个速度差异非常大的组件

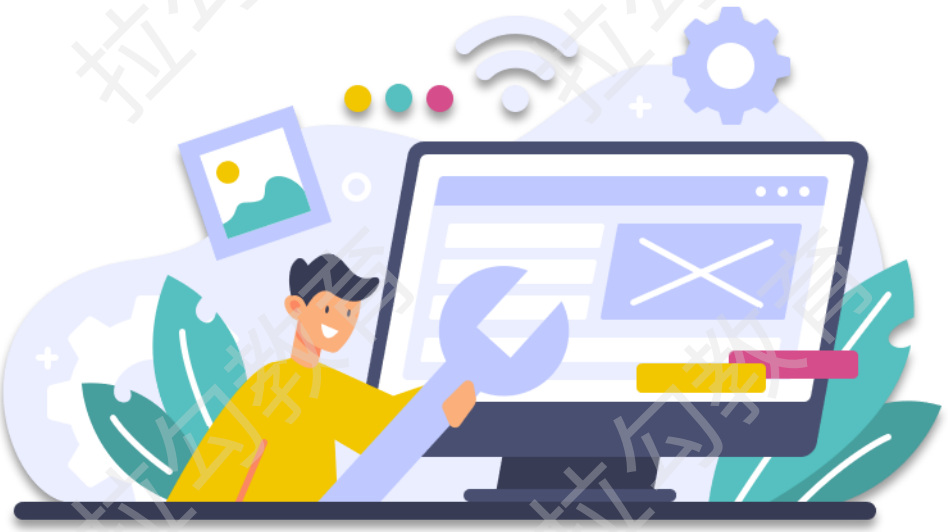


案例分析：无处不在的缓存，高并发系统的法宝

根据缓存所处的物理位置，一般分为**进程内缓存**和**进程外缓存**

在 Java 中，**进程内缓存**，就是**堆内缓存**

Spring 的默认实现里，包含 Ehcache、JCache、Caffeine、Guava Cache 等



Guava 的 LoadingCache

Guava 中的 **LoadingCache**（下面简称 LC），是**堆内缓存工具**

缓存一般是比较昂贵的组件，容量是有限制的：

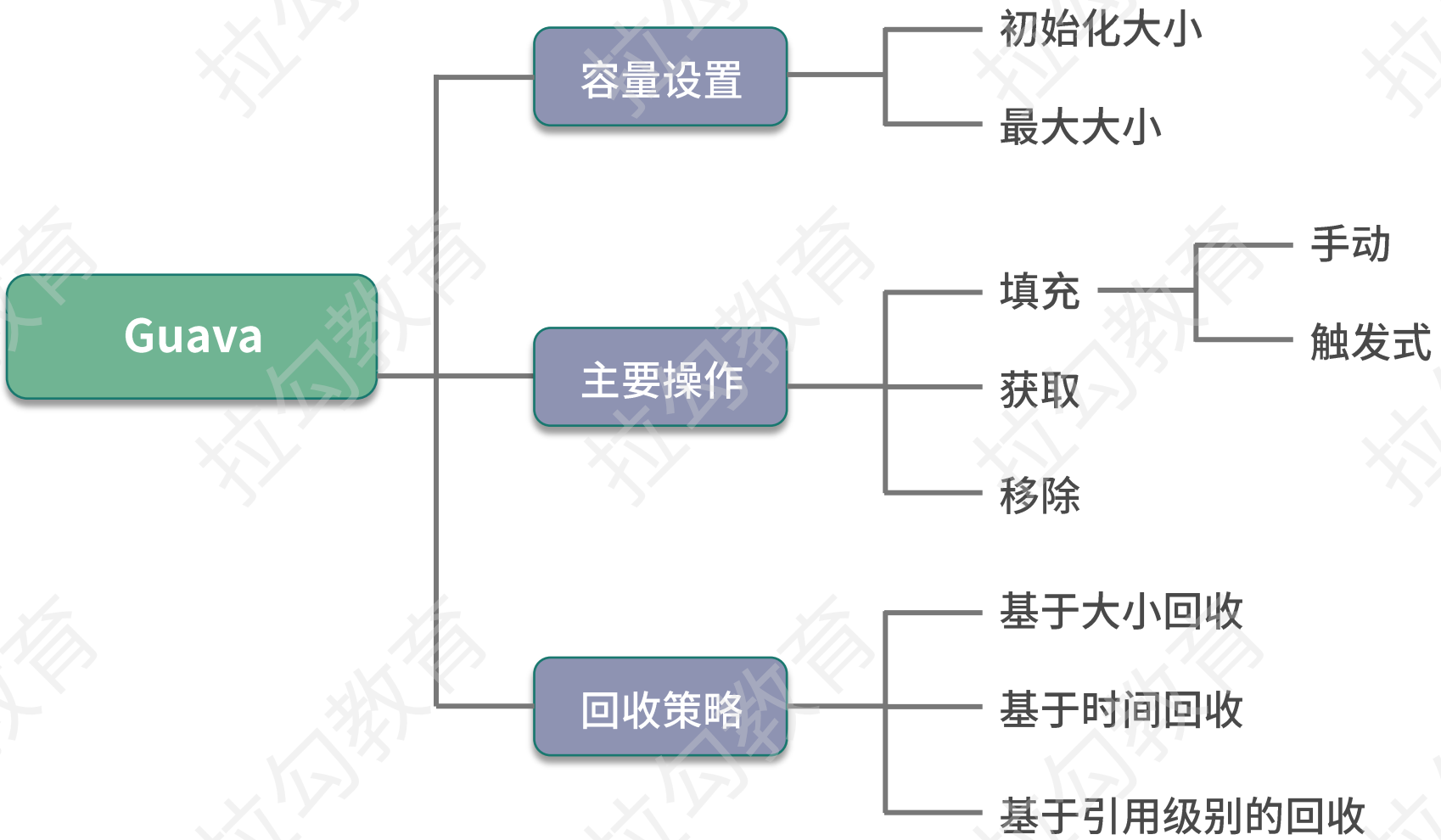
- **缓存空间过小**，会造成高命中率的元素被频繁移出
- **缓存空间过大**，浪费宝贵的缓存资源，对垃圾回收产生一定的压力

Guava 的 LoadingCache

通过 Maven，可引入 guava 的 jar 包

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>29.0-jre</version>
</dependency>
```

Guava 的 LoadingCache



缓存初始化

通过下面的参数设置一下 LC 的大小

- **maximumSize** 设置缓存池的最大容量，达到此容量将会清理其他元素
- **initialCapacity** 默认值是 16，表示初始化大小
- **concurrencyLevel** 默认值是 4，和初始化大小配合使用

  表示会将缓存的内存划分成 4 个 segment，用来支持高并发的存取

缓存操作

缓存数据是怎么放进去的呢？

- **使用 put 方法手动处理**

比如从数据库里查询出一个 User 对象，然后手动调用代码进去

- **主动触发**

通过提供一个 CacheLoader 的实现，可以在用到这个对象的时候，进行延迟加载

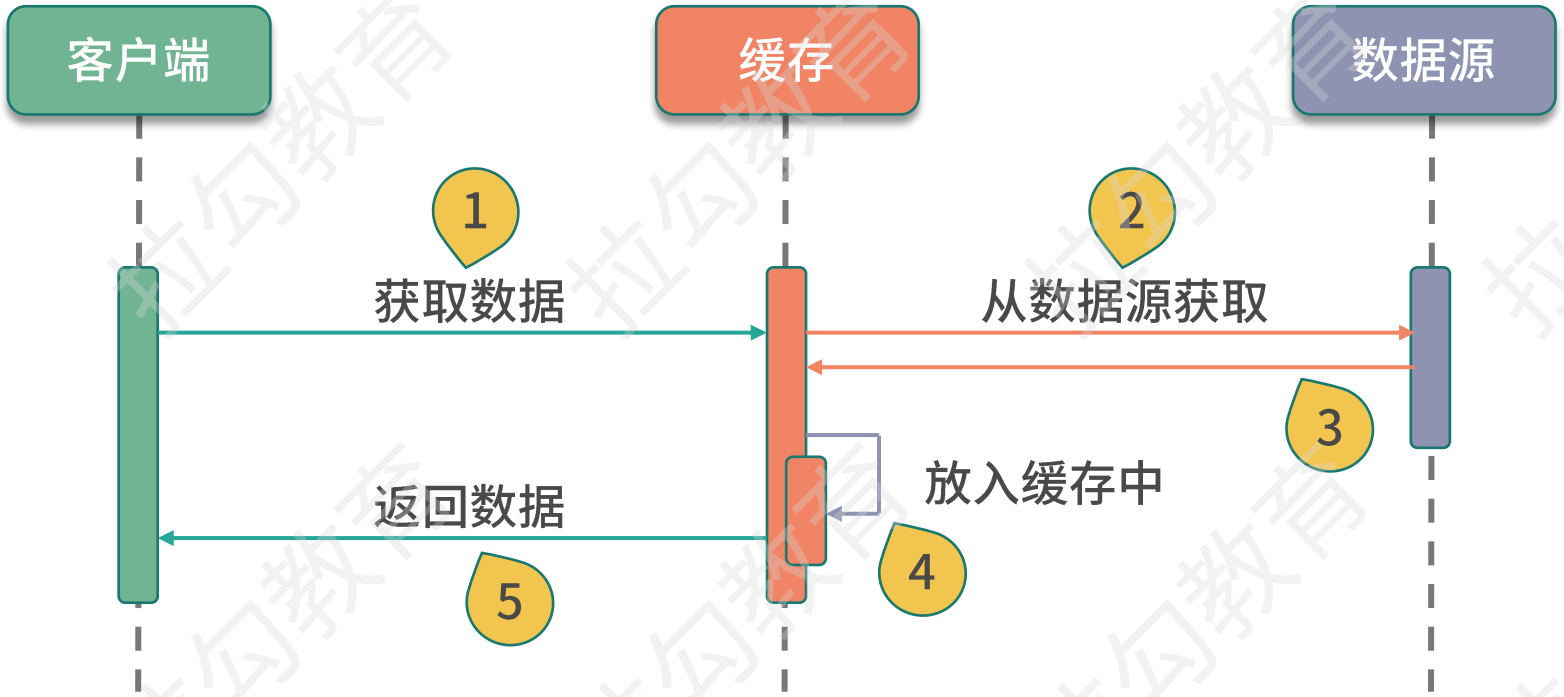
缓存操作


```
public static void main(String[] args) {
    LoadingCache<String, String> lc = CacheBuilder
            .newBuilder()
            .build(new CacheLoader<String, String>() {
                @Override
                public String load(String key) throws Exception {
                    return slowMethod(key);
                }
            });
}

static String slowMethod(String key) throws Exception {
    Thread.sleep(1000);
    return key + ".result";
}
```

缓存操作

使用 get 方法获取缓存的值。比如，执行 lc.get("a") 时



缓存操作

手动删除某一个元素——invalidate 方法

数据的删除操作，只需要设置一个监听器

```
.removalListener(notification ->
System.out.println(notification))
```

回收策略

（1）基于容量

如果缓存满了，就会按照 LRU 算法来移除其他元素

（2）基于时间

- 通过 expireAfterWrite 方法设置数据写入以后在某个时间失效
- 通过 expireAfterAccess 方法设置最早访问的元素，并优先将其删除

（3）基于 JVM 的垃圾回收

对象的引用有强、软、弱、虚等四个级别，通过 weakKeys 等函数可设置相应的引用级别

当 JVM 垃圾回收的时候，会主动清理这些数据

回收策略

高频面试题：如果你同时设置了 weakKeys 和 weakValues函数，LC 会有什么反应?

答案：如果同时设置了这两个函数，它代表的意思是

当没有任何强引用，与 key 或者 value 有关系时，就删掉整个缓存项

缓存造成内存故障

LC 通过 recordStats 函数，对缓存加载和命中率等情况进行监控

注意：LC 是基于数据条数而不是基于缓存物理大小的

如果缓存的对象特别大，会造成不可预料的内存占用

缓存造成内存故障

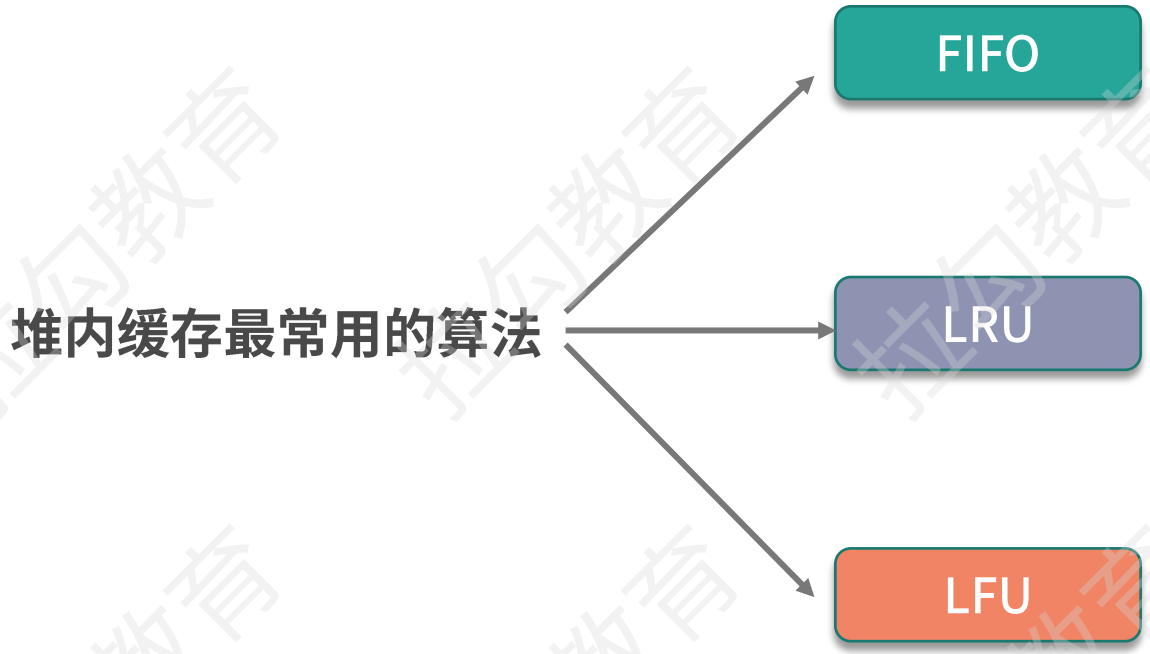
大多数堆内缓存，会将对象的引用设置成**弱引用或软引用**

当缓存使用非常频繁，数据量又比较大的情况下

如果发生了垃圾回收（GC），缓存空间会被释放掉，但又被迅速占满，从而会再次触发垃圾回收

方法：把缓存设置的小一些，减轻 JVM 的负担

缓存算法——算法介绍



缓存算法——算法介绍

FIFO

先进先出的模式。如果缓存容量满了，将会移除最先加入的元素

LRU

LRU 最近最少使用的意思，当缓存容量达到上限，会优先移除那些最久未被使用的数据

缓存算法——算法介绍

LFU

LFU 是最近最不常用的意思，增加了访问次数的维度

- 缓存满时，将优先移除访问次数最少的元素
- 有多个访问次数相同的元素时，优先移除最久未被使用的元素

实现一个 LRU 算法

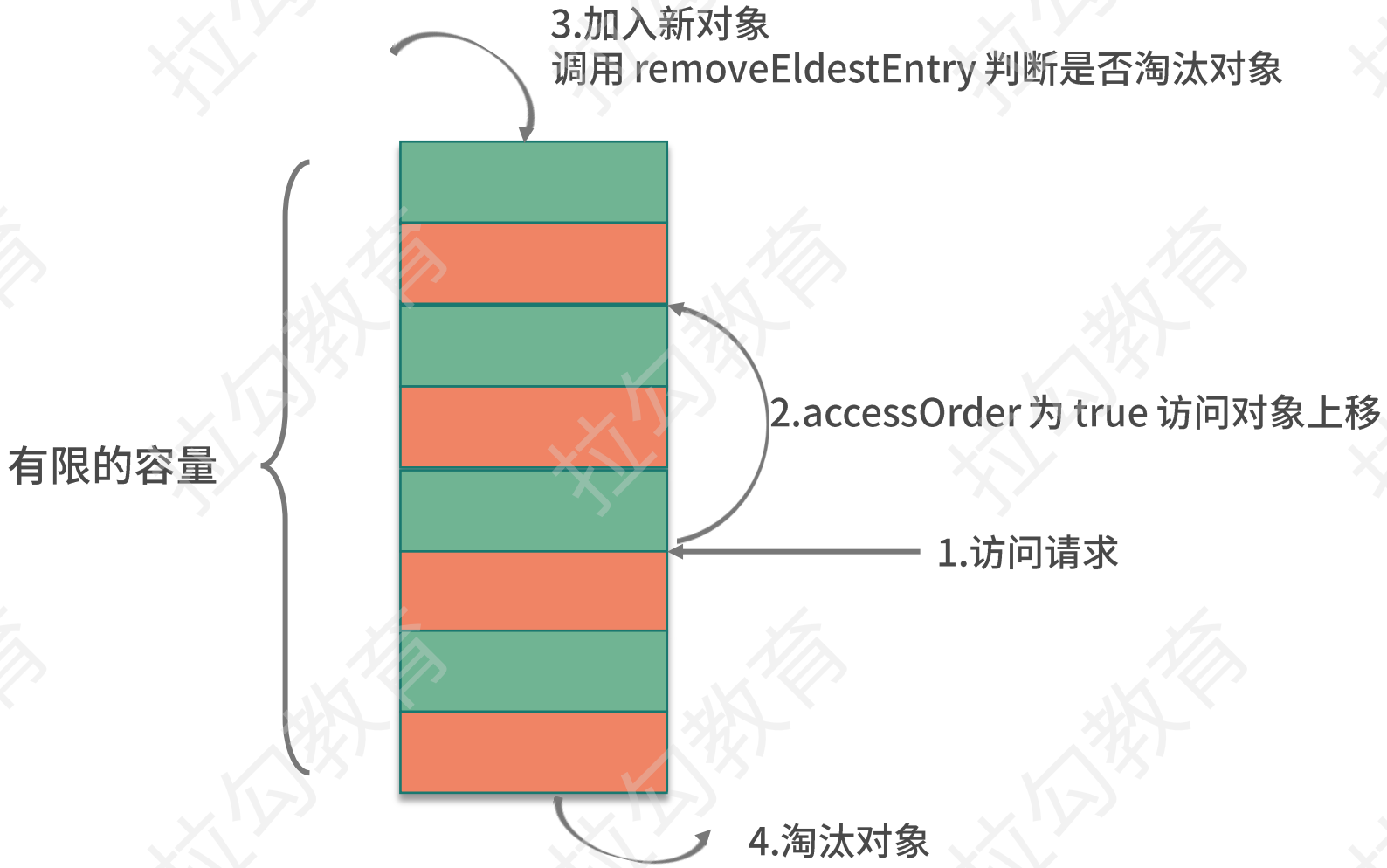
最常用的 LRU 算法—— **LinkedHashMap**

```
public LinkedHashMap(int initialCapacity,
        float loadFactor,
        boolean accessOrder)
```

当 accessOrder 的值为 **true** 时，将按照对象的**访问顺序**排序

当 accessOrder 的值为 **false** 时，将按照对象的**插入顺序**排序

实现一个 LRU 算法



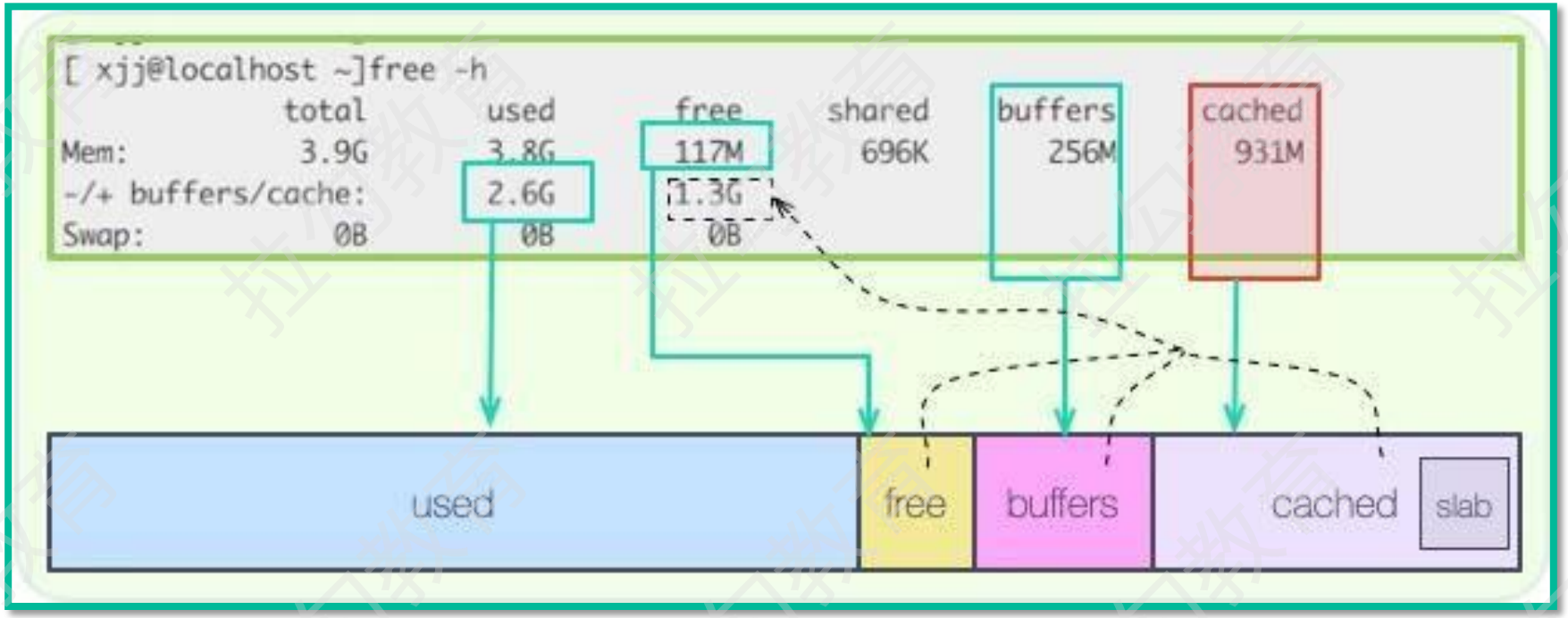
实现一个 LRU 算法

拉勾教育
互联网人实战大学

```
public class LRU extends LinkedHashMap {
    int capacity;
    public LRU(int capacity) {
        super(16, 0.75f, true);
        this.capacity = capacity;
    }
    @Override
    protected boolean removeEldestEntry(Map.Entry eldest) {
        return size() > capacity;
    }
}
```

进一步加速

在 Linux 系统中，通过 free 命令，能够看到系统内存的使用状态



进一步加速

操作系统使用智能的**预读算法**（readahead），将数据从硬盘中加载到缓存中

预读算法有三个关键点：

- **预测性**，能够根据应用的使用数据，提前预测应用后续的操作目标
- **提前**，能够将这些数据提前加载到缓存中，保证命中率
- **批量**，将小块的、频繁的读取操作，合并成顺序的批量读取，提高性能

进一步加速

如果数据集合比较小，访问频率又非常高

可以使用**完全载入的方式**，来替换懒加载的方式

在系统启动的时候，将数据加载到缓存中

缓存优化的一般思路

缓存针对的主要是**读操作**。下面的场景可以使用缓存组件进行性能优化：

- 存在数据热点，缓存的数据能够被频繁使用
- 读操作明显比写操作要多
- 下游功能存在着比较悬殊的性能差异，下游服务能力有限
- 加入缓存以后，不会影响程序的正确性，或者引入不可预料的复杂性

缓存优化的一般思路

缓存组件和**缓冲**，在两个组件速度严重不匹配的时候，引入的一个中间层

服务的目标不同：

- **缓冲**——数据一般只使用一次，等待缓冲区满了，就执行 flush 操作
- **缓存**——数据被载入之后，可以多次使用，数据将会共享多次

缓存最重要的指标就是命中率

影响命中率的因素

（1）缓存容量

缓存的容量总是有限制的。缓存不是越大越好，它不能明显挤占业务的内存

（2）数据集类型

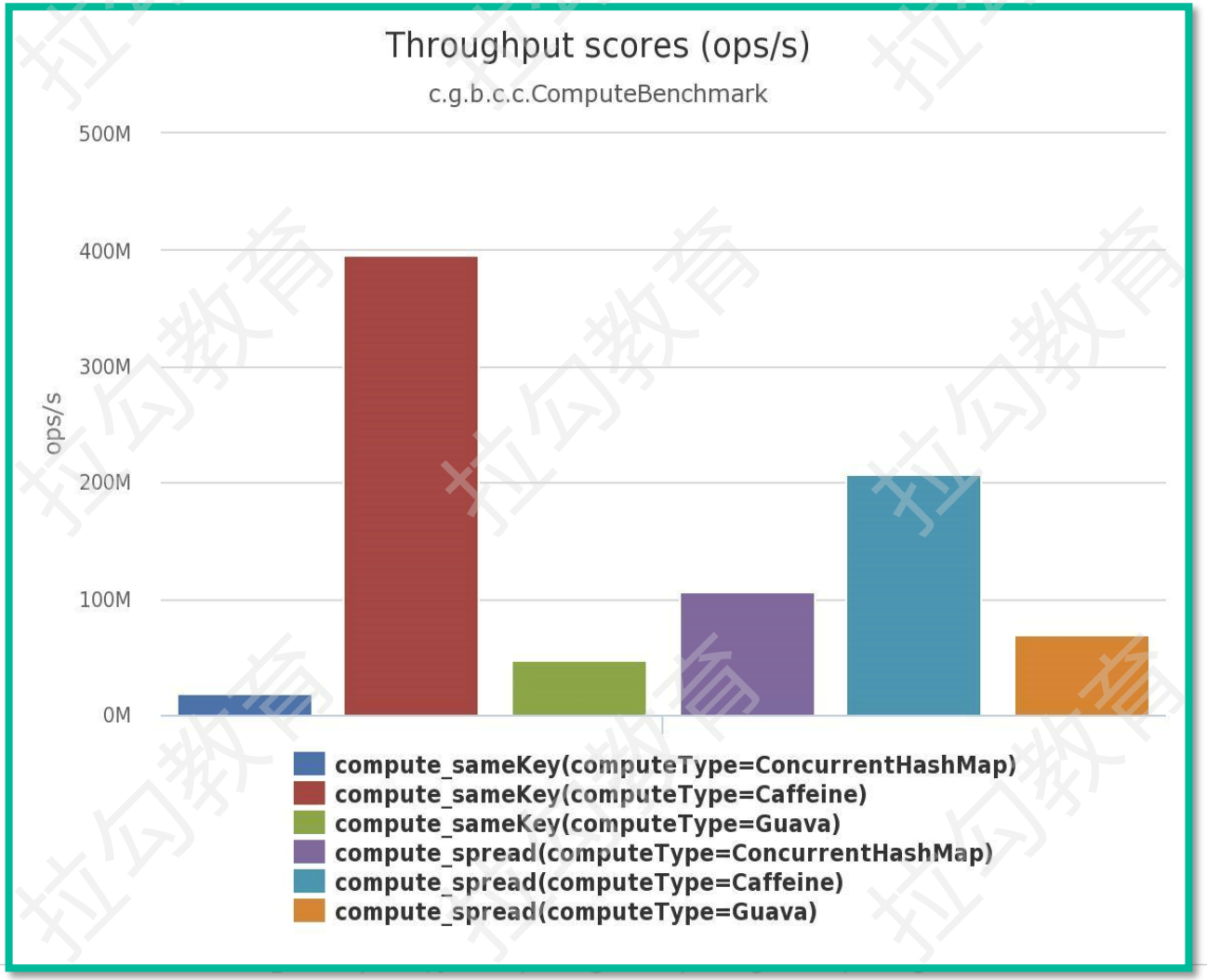
如果缓存的数据是非热点数据，或者操作几次就不再使用的冷数据，命中率会很低

（3）缓存失效策略

缓存算法影响命中率和性能，目前效率最高的算法是 Caffeine 使用的 W-TinyLFU 算法

新版本的 spring-cache，已经默认支持 Caffeine

影响命中率的因素



影响命中率的因素

推荐使用 **Guava Cache** 或者 **Caffeine** 作为堆内缓存解决方案

通过它们提供的一系列监控指标，来调整缓存的大小和内容

- 缓存命中率达到 **50% 以上**，作用开始变得显著
- 缓存命中率低于 **10%**，需要考虑缓存组件的必要性

影响命中率的因素

如何保证缓存与源数据的同步？

小结

- 以 Guava 的 LoadingCache 为例，讲解了堆内缓存设计的一些思路
- 介绍了一个因为缓存不合理利用所造成的内存故障
- 讲解了三个常用的缓存算法 LRU、LFU、FIFO

  并以 LinkedHashMap 为基础，实现了一个最简单的 LRU 缓存
- 使用预读或者提前载入等方式，进一步加速应用的方法

  readahead 技术在操作系统、数据库中使用非常多，性能提升比较显著
- 利用缓存框架的一些监控数据，来调整缓存的命中率，要达到 50% 的命中率才算有较好的效果

缓存应用的例子

- **HTTP 304 状态码——Not Modified**，浏览器客户端会发送一个条件性的请求

服务端可以通过 If-Modified-Since 头信息判断缓冲的文件是否是最新的

如果是，客户端直接使用缓存

- **CDN**，是一种**变相的缓存**

用户会从离它最近最快的节点，读取文件内容

如果这个节点没有缓存这个文件，CDN 节点会从源站拉取一份

缓存应用的例子

当遇到性能相差悬殊的两个组件，想要提升它们的速度

可以考虑使用**缓存**的方式，通过缓解差异，将系统进行加速

Next：08 |《案例分析：Redis 如何助力秒杀业务》

拉勾教育

—互联网人实战大学—

关注拉勾「教育公众号」
获取更多课程信息

L / A / G / O / U