《Java性能优化与面试21讲》

李国

— 拉勾教育出品 —

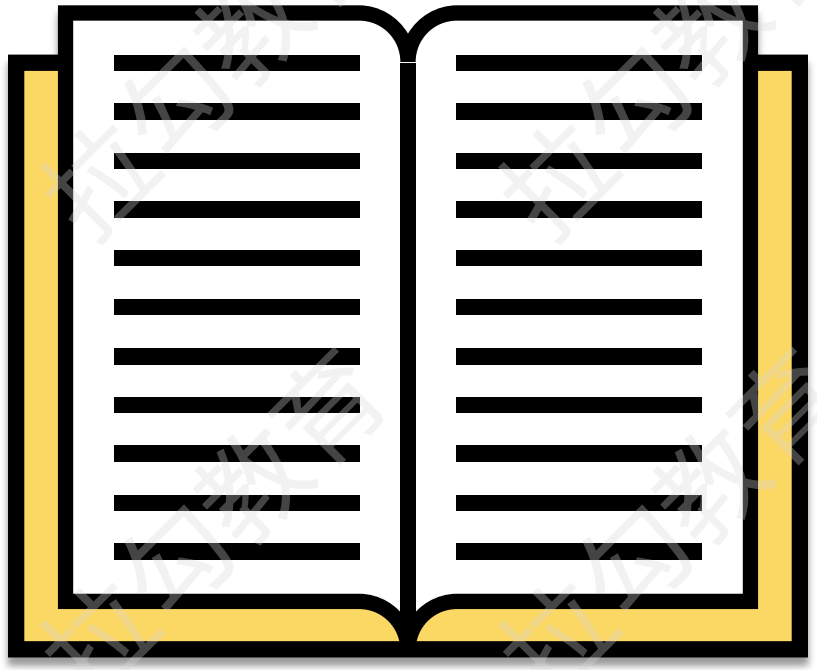
06 | 案例分析：缓冲区如何让代码加速

深入理解缓冲的本质

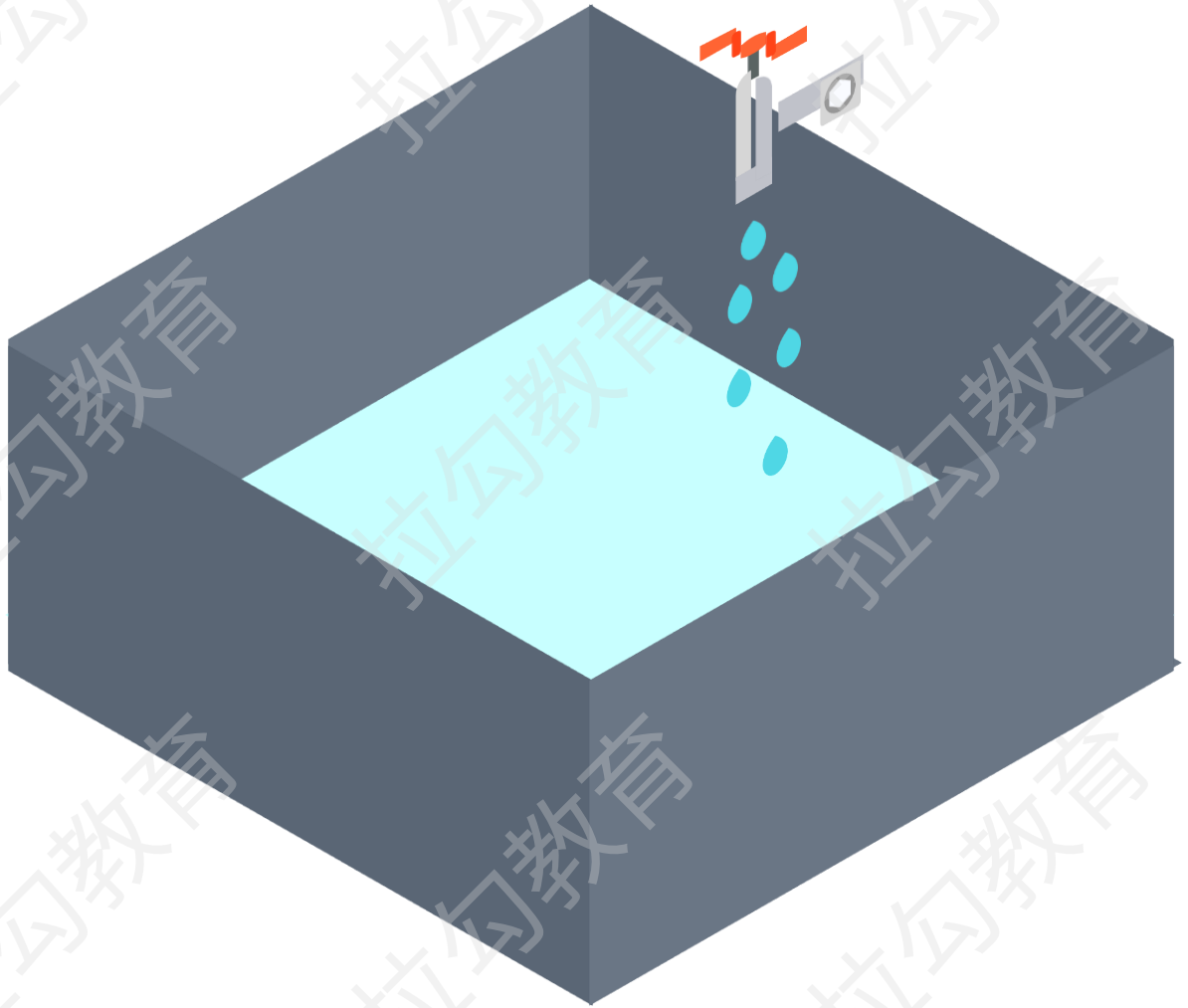
缓冲（Buffer）

通过对数据进行暂存，然后批量进行传输或者操作

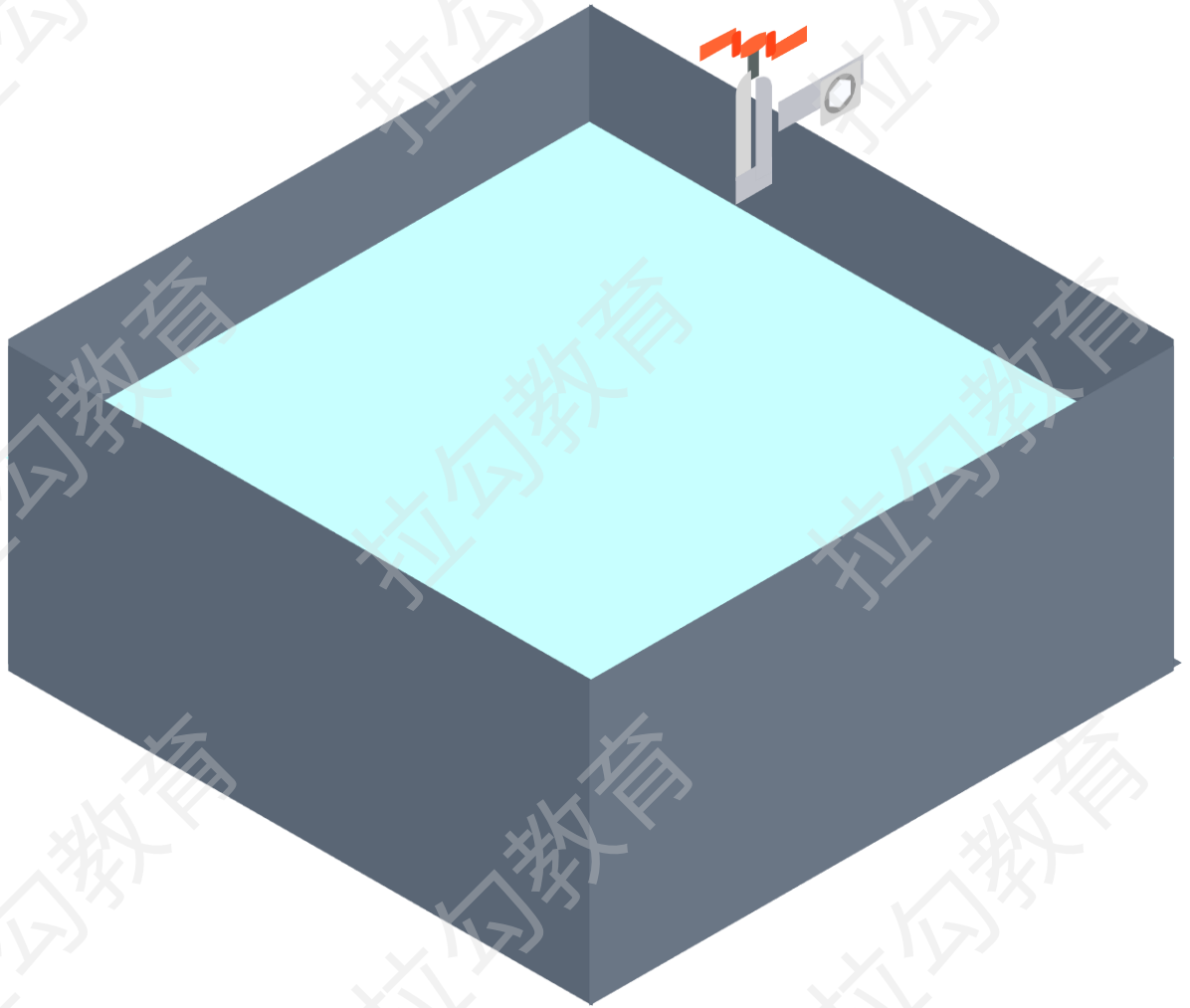
多采用顺序方式，来缓解不同设备之间次数频繁但速度缓慢的随机读写



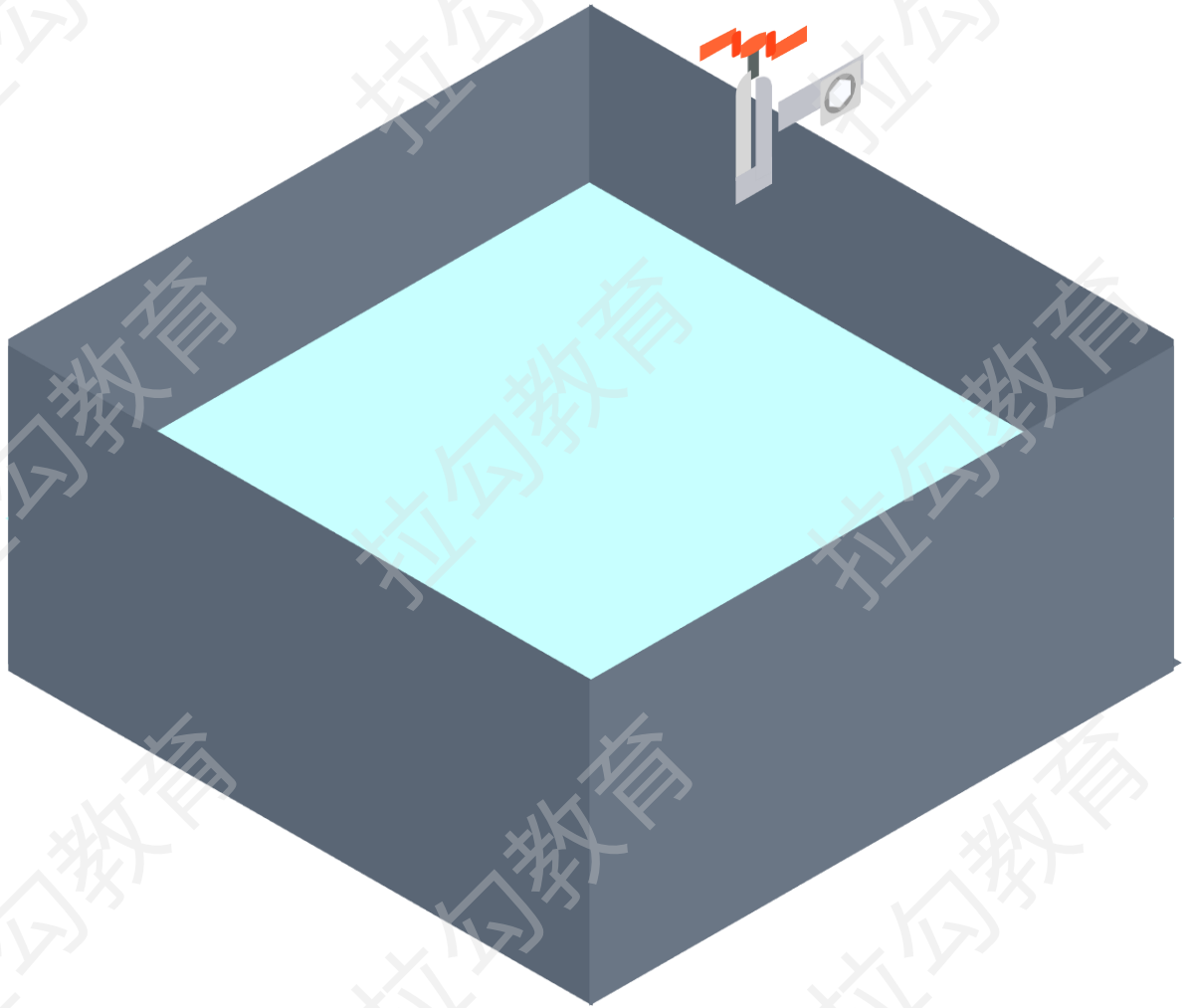
深入理解缓冲的本质



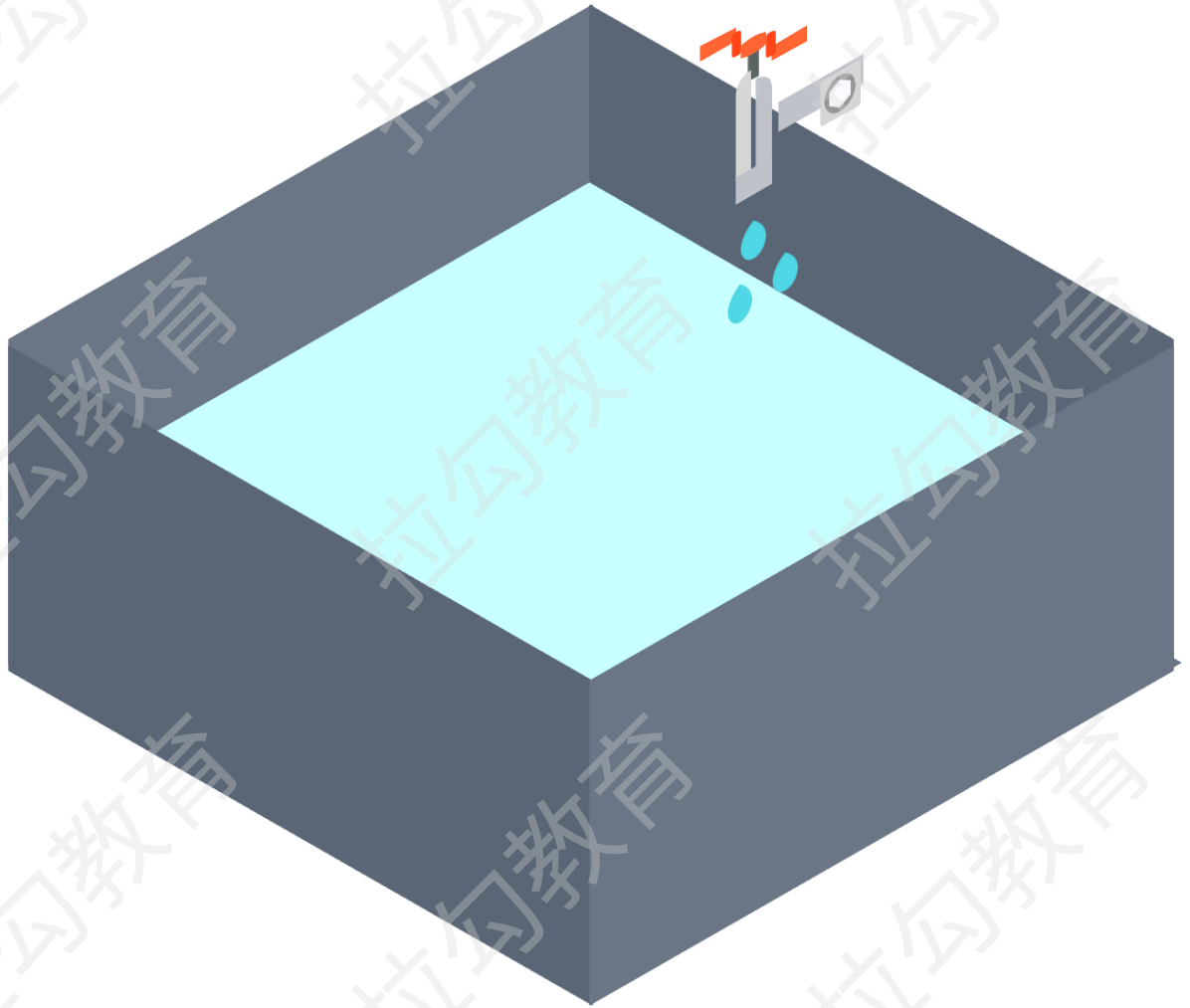
深入理解缓冲的本质



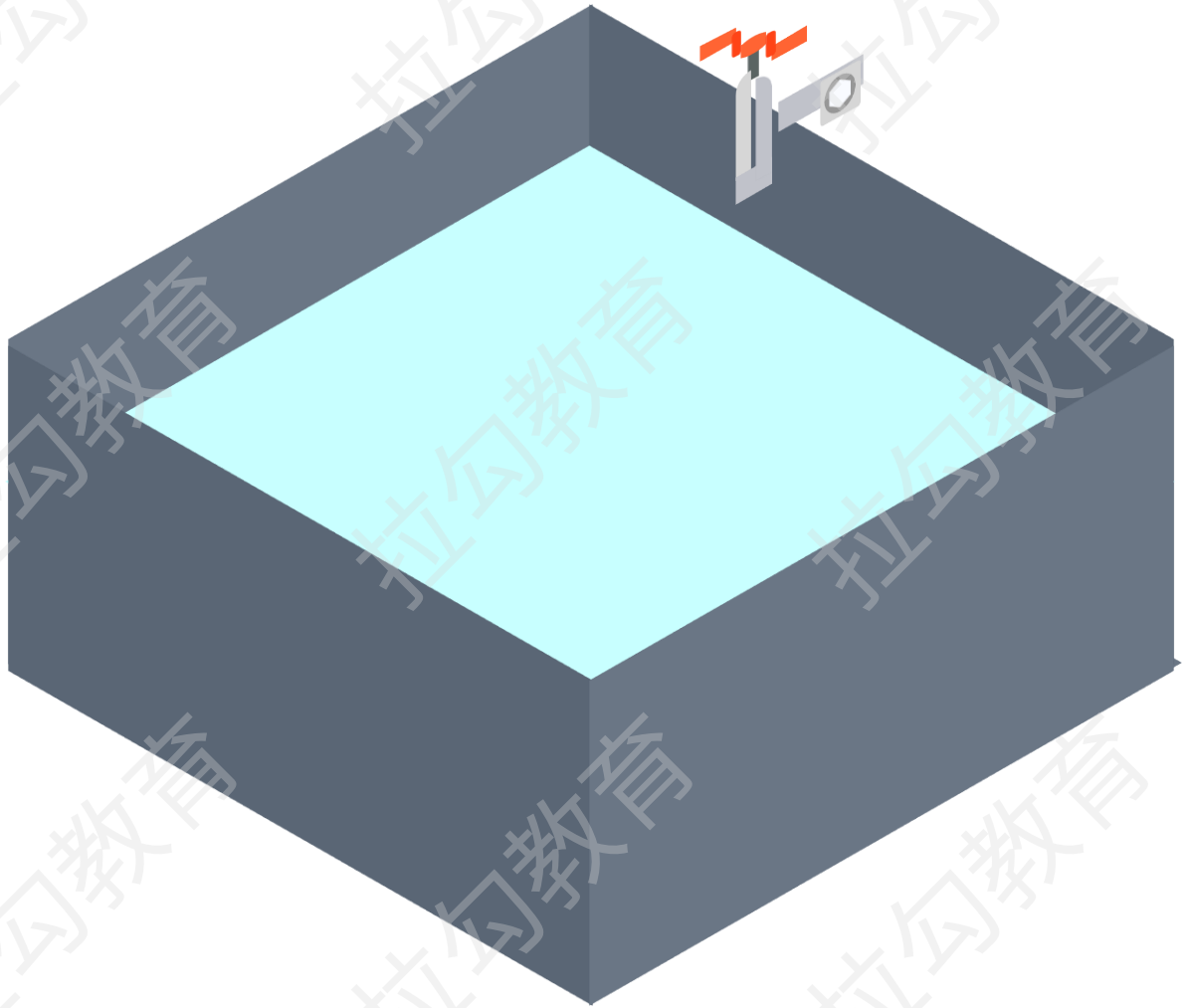
深入理解缓冲的本质



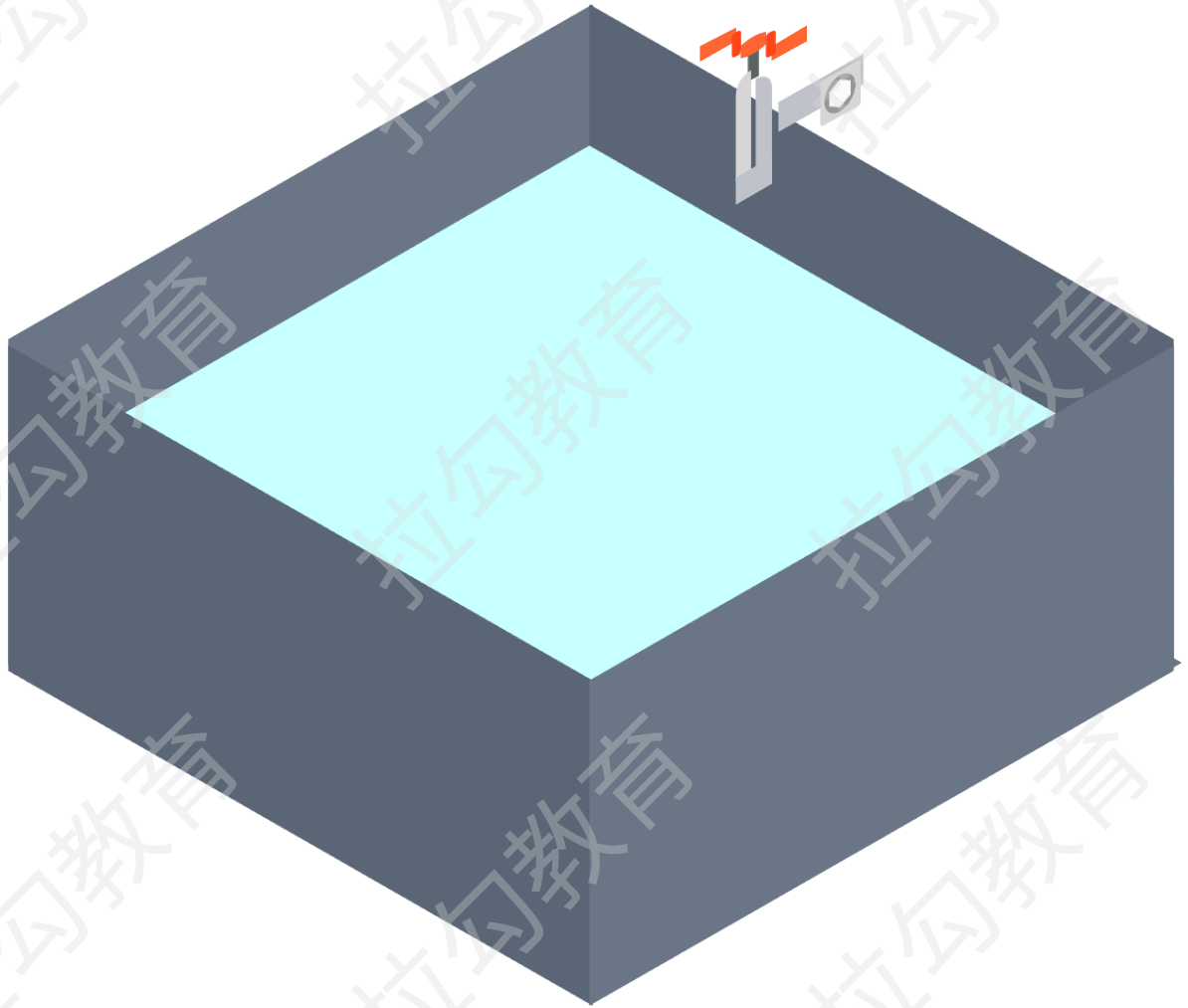
深入理解缓冲的本质



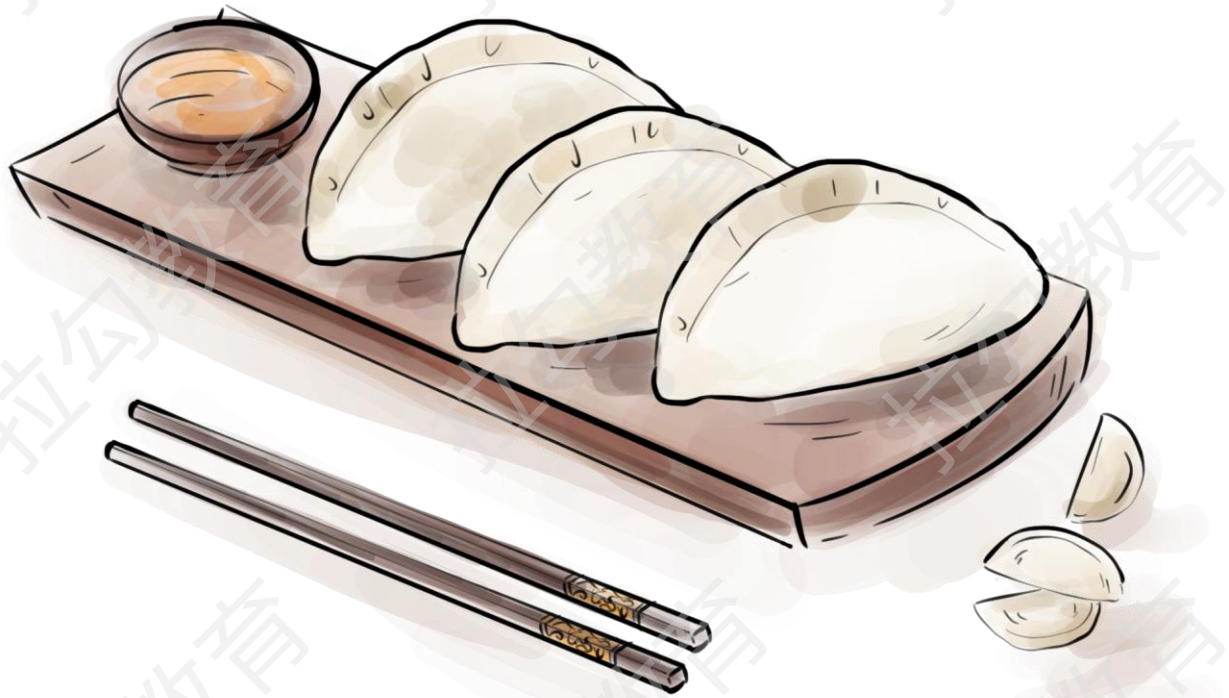
深入理解缓冲的本质



深入理解缓冲的本质



深入理解缓冲的本质



深入理解缓冲的本质

宏观上

JVM 的堆就是一个大的缓冲区，代码不停地在堆空间中生产对象

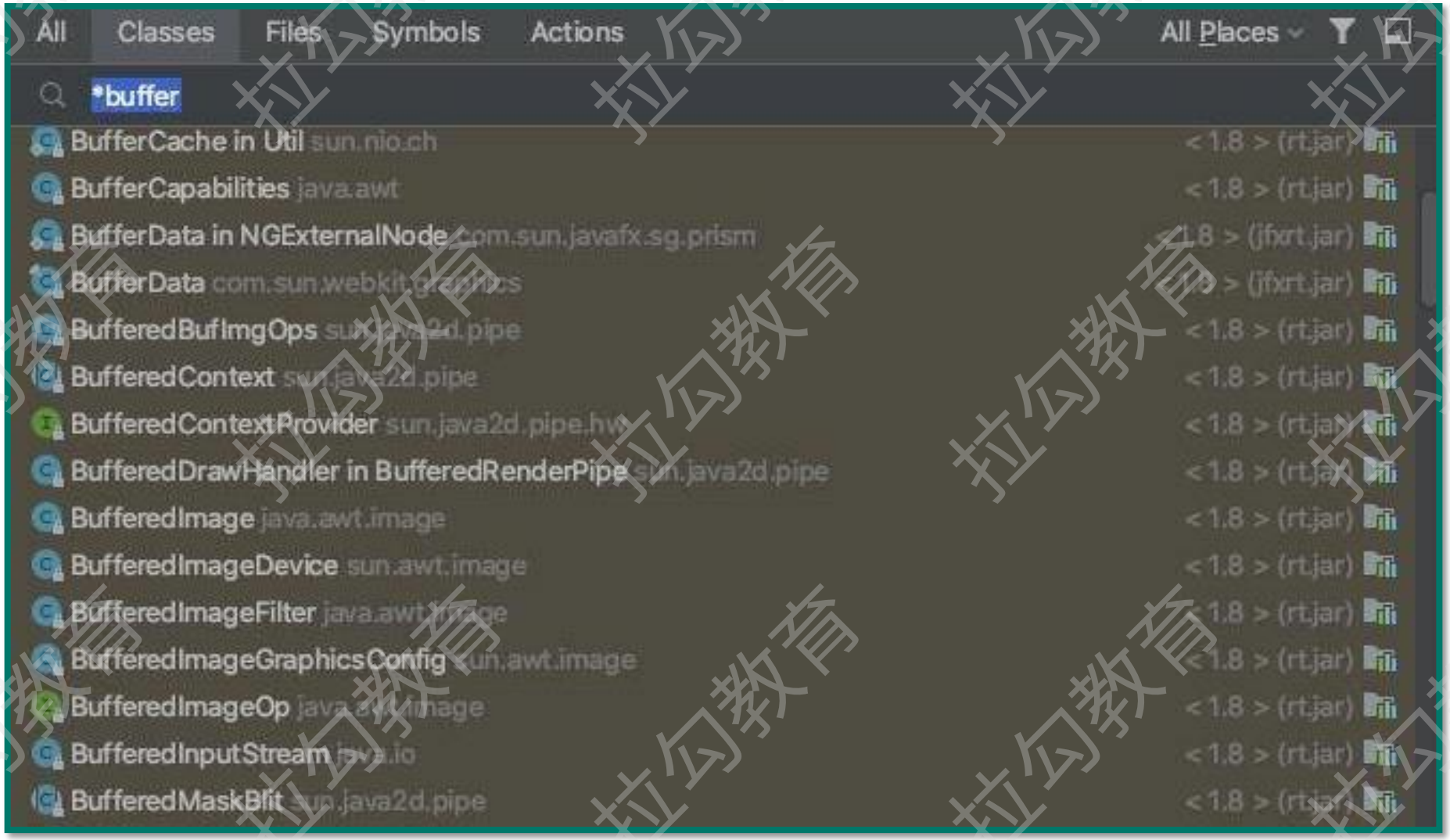
而垃圾回收器进程则在背后默默地进行垃圾回收

深入理解缓冲的本质

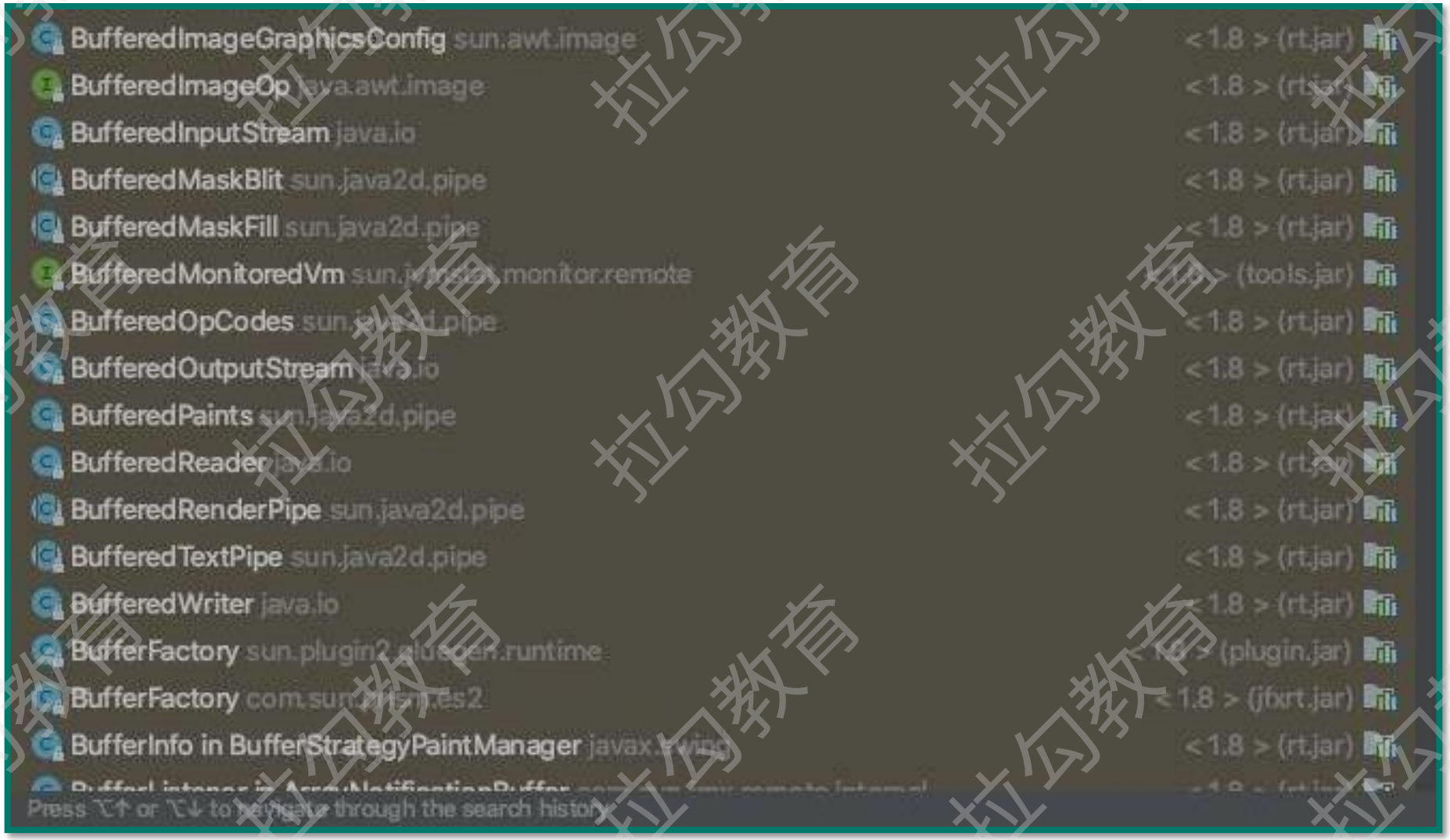
缓冲区的好处：

- 缓冲双方能各自保持自己的操作节奏，操作处理顺序也不会打乱，可以 one by one 顺序进行
- 以批量的方式处理，减少网络交互和繁重的 I/O 操作，从而减少性能损耗
- 优化用户体验，比如常见的音频/视频缓冲加载，通过提前缓冲数据，达到流畅的播放效果

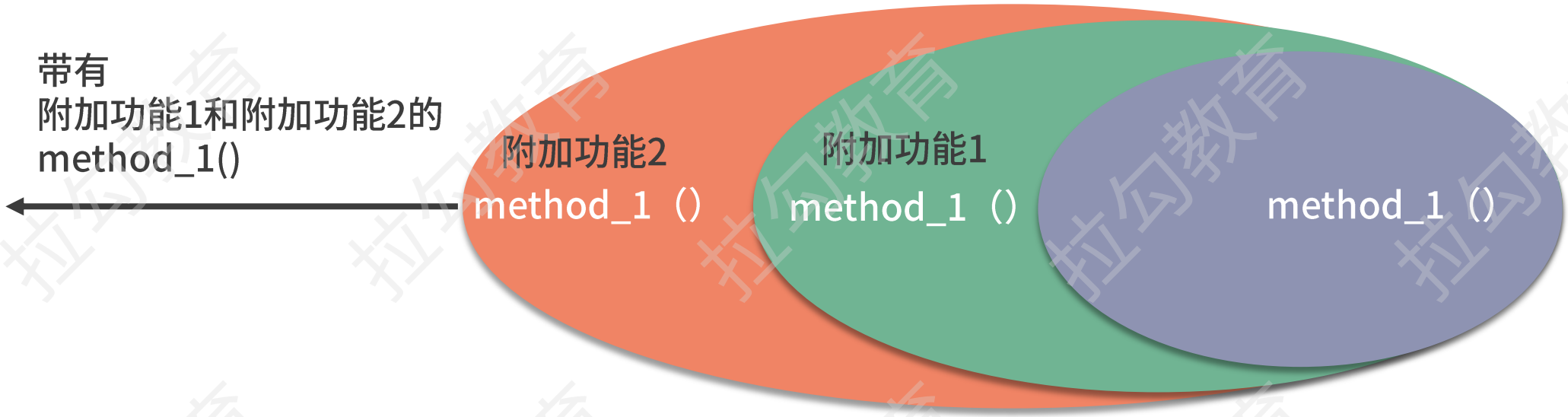
深入理解缓冲的本质


All
Classes
Files
Symbols
Actions
All Places
*buffer
BufferCache in Util sun.nio.ch
BufferCapabilities java.awt
BufferData in NGExternalNode com.sun.javafx.sg.prism
BufferData com.sun.webkit.graphics
BufferedBufImgOps sun.java2d.pipe
BufferedContext sun.java2d.pipe
BufferedContextProvider sun.java2d.pipe.hw
BufferedDrawHandler in BufferedRenderPipe sun.java2d.pipe
BufferedImage java.awt.image
BufferedImageDevice sun.awt.image
BufferedImageFilter java.awt.image
BufferedImageGraphicsConfig sun.awt.image
BufferedImageOp java.awt.image
BufferedInputStream java.io
BufferedMaskBlit sun.java2d.pipe
< 1.8 > (rt.jar)
< 1.8 > (jfxrt.jar)

深入理解缓冲的本质


BufferedImageGraphicsConfig sun.awt.image <1.8 > (rt.jar)
BufferedImageOp java.awt.image <1.8 > (rt.jar)
BufferedInputStream java.io <1.8 > (rt.jar)
BufferedMaskBlit sun.java2d.pipe <1.8 > (rt.jar)
BufferedMaskFill sun.java2d.pipe <1.8 > (rt.jar)
BufferedMonitoredVm sun.jvmstat.monitor.remote (tools.jar)
BufferedOpCodes sun.java2d.pipe <1.8 > (rt.jar)
BufferedOutputStream java.io <1.8 > (rt.jar)
BufferedPaints sun.java2d.pipe <1.8 > (rt.jar)
BufferedReader java.io <1.8 > (rt.jar)
BufferedRenderPipe sun.java2d.pipe <1.8 > (rt.jar)
BufferedTextPipe sun.java2d.pipe <1.8 > (rt.jar)
BufferedWriter java.io <1.8 > (rt.jar)
BufferFactory sun.plugin2.gluegen.runtime (plugin.jar)
BufferFactory com.sun.prism.es2 <1.8 > (jfxrt.jar)
BufferInfo in BufferStrategyPaintManager javax.swing <1.8 > (rt.jar)
Press ⌥↑ or ⌥↓ to navigate through the search history

文件读写流



文件读写流


```
int result = 0;
try (Reader reader = new FileReader(FILE_PATH)) {
    int value;
    while ((value = reader.read()) != -1) {
        result += value;
    }
}
return result;
```

文件读写流


```
int result = 0;
try (Reader reader = new BufferedReader(new FileReader(FILE_PATH))) {
    int value;
    while ((value = reader.read()) != -1) {
        result += value;
    }
}
return result;
```

文件读写流


```
//代码来自JDK
public synchronized int read() throws IOException {
    if (pos >= count) {
        fill();
        if (pos >= count)
            return -1;
    }
    return getBufIfOpen()[pos++] & 0xff;
}
```

文件读写流


```
//代码来自JDK
private void fill() throws IOException {
    byte[] buffer = getBufIfOpen();
    if (markpos < 0)
        pos = 0;            /* no mark: throw away the buffer */
    else if (pos >= buffer.length)  /* no room left in buffer */
        if (markpos > 0) {  /* can throw away early part of the buffer */
            int sz = pos - markpos;
            System.arraycopy(buffer, markpos, buffer, 0, sz);
            pos = sz;
            markpos = 0;
        } else if (buffer.length >= marklimit) {
            markpos = -1;   /* buffer got too big, invalidate mark */
            pos = 0;        /* drop buffer contents */
        } else if (buffer.length >= MAX_BUFFER_SIZE) {
            throw new OutOfMemoryError("Required array size too large");
        } else {            /* grow buffer */
            int nsz = (pos <= MAX_BUFFER_SIZE - pos) ?
                    pos * 2 : MAX_BUFFER_SIZE;
            if (nsz > marklimit)
```

文件读写流


```
            if (nsz > marklimit)
                nsz = marklimit;
            byte nbuf[] = new byte[nsz];
            System.arraycopy(buffer, 0, nbuf, 0, pos);
            if (!bufUpdater.compareAndSet(this, buffer, nbuf)) {
                // Can't replace buf if there was an async close.
                // Note: This would need to be changed if fill()
                // is ever made accessible to multiple threads.
                // But for now, the only way CAS can fail is via close.
                // assert buf == null;
                throw new IOException("Stream closed");
            }
            buffer = nbuf;
        }
        count = pos;
        int n = getInIfOpen().read(buffer, pos, buffer.length - pos);
        if (n > 0)
            count = n + pos;
    }
```

文件读写流


```
int n = getInIfOpen().read(buffer, pos, buffer.length - pos);
```

文件读写流


```
int n = getInIfOpen().read(buffer, pos, buffer.length - pos);
```

字符流操作的对象，一般是文件或者 Socket

要从这些缓慢的设备中，通过频繁的交互获取数据，效率非常慢

而缓冲区的数据是保存在内存中的，能够显著地提升读写速度

文件读写流


```
Benchmark                                  Mode  Cnt     Score    Error  Units
BenchmarkReader.bufferedReaderTest         avgt    5    77.946 ±  12.640  ms/op
BenchmarkReader.fileReadTest               avgt    5   131.570 ± 111.055  ms/op
```

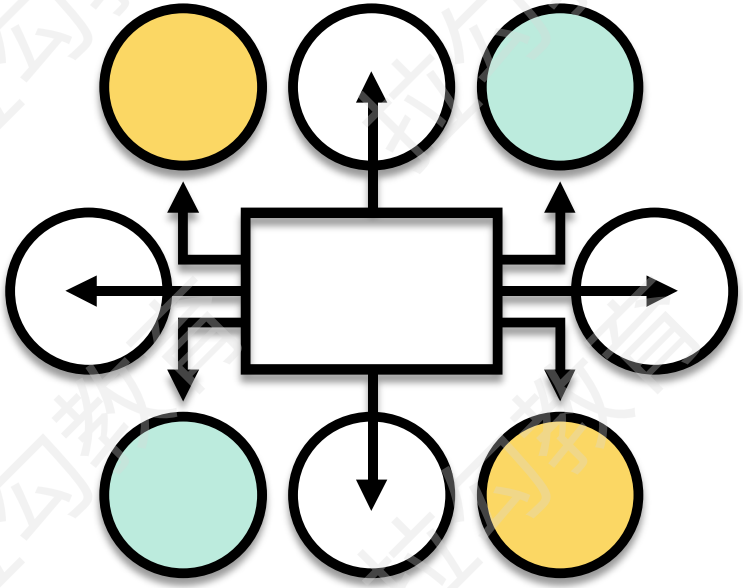
日志缓冲

在高并发应用中

即使对日志进行了采样，日志数量依旧惊人，所以选择高速的日志组件至关重要

SLF4J 是 Java 里标准的日志记录库，它是一个允许你使用任何 Java 日志记录库的抽象适配层

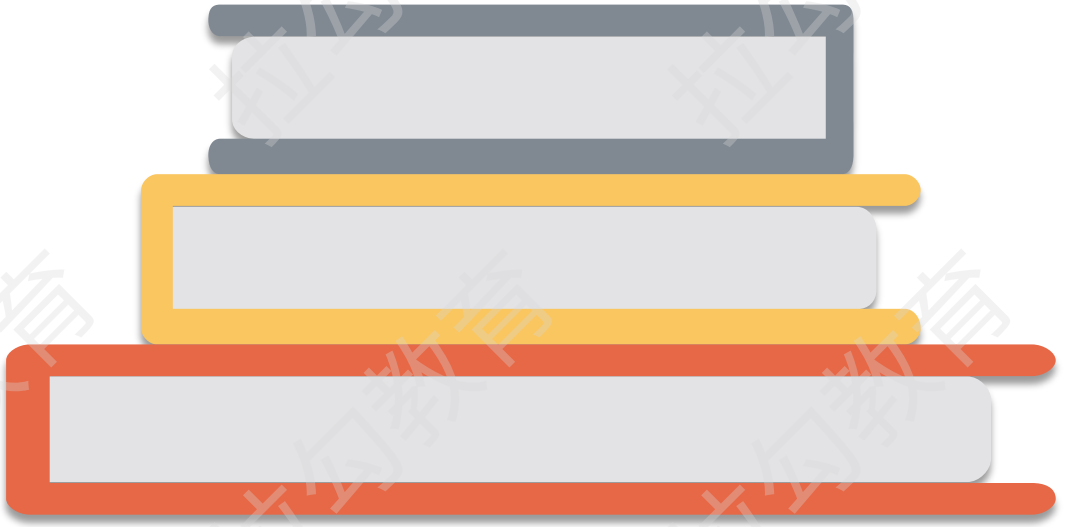
最常用的实现是 Logback，支持修改后自动 reload，JUL 还要流行



日志缓冲

使用异步日志有两个考虑：

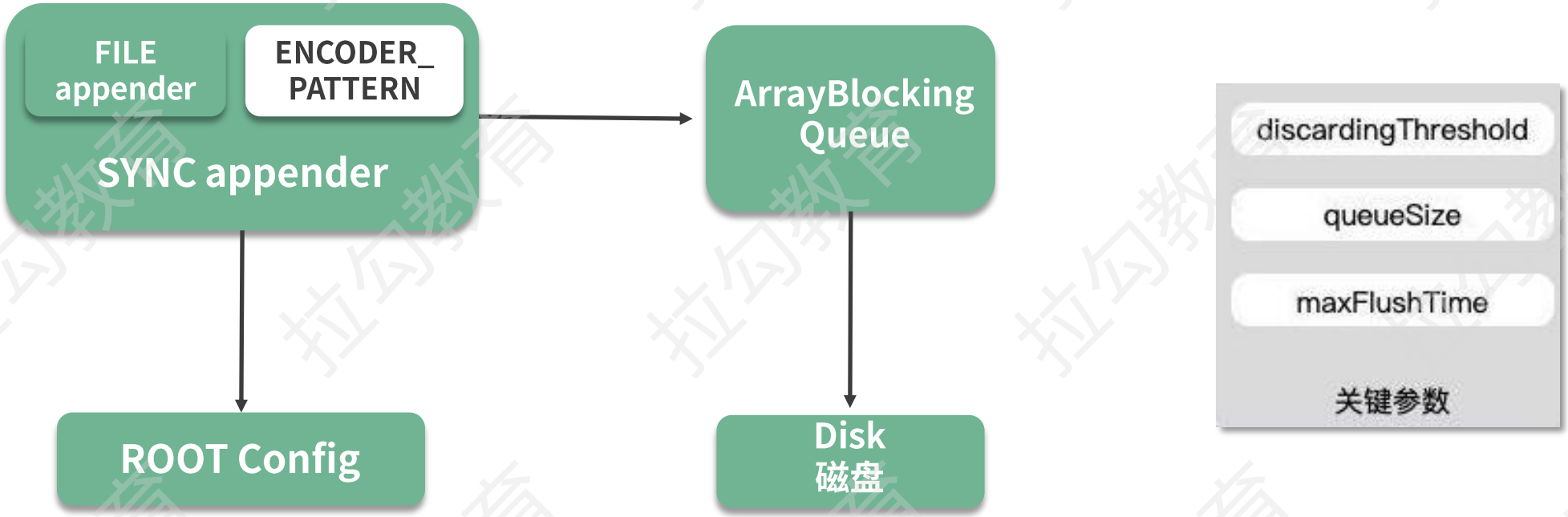
- 同步日志的写入，会阻塞业务，导致服务接口的耗时增加
- 日志写入磁盘的代价是昂贵的，如果每产生一条日志就写入一次，CPU会花很多时间在磁盘 I/O 上



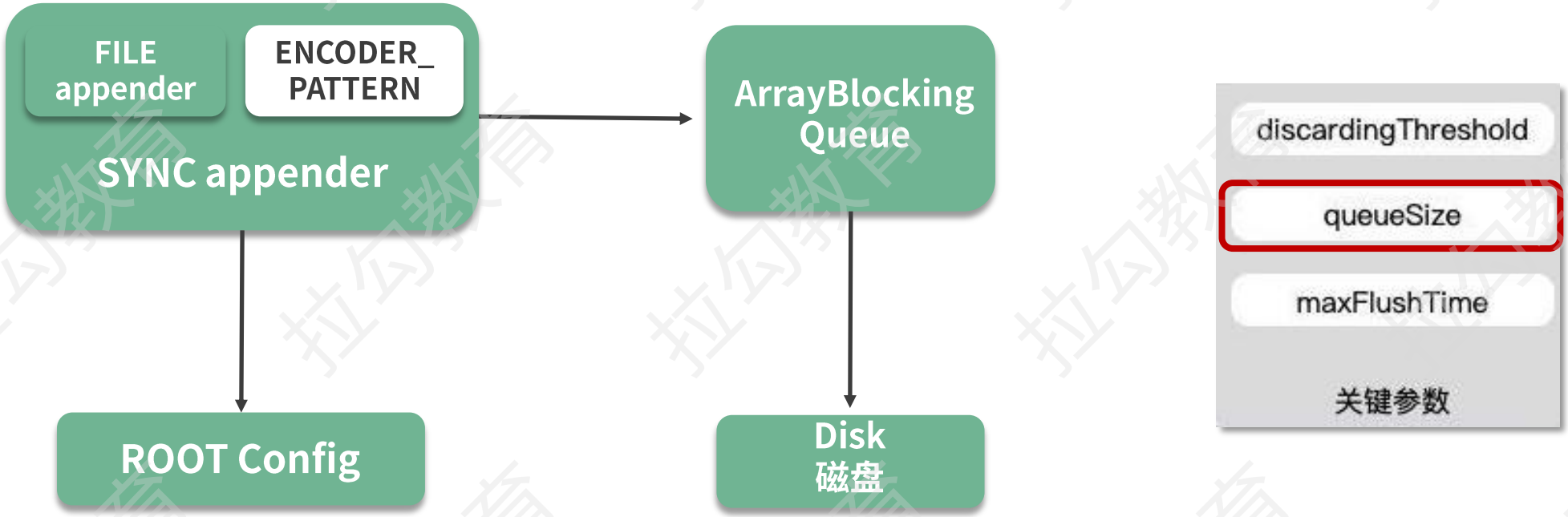
日志缓冲


```
<appender name ="ASYNC" class=
"ch.qos.logback.classic.AsyncAppender">
    <discardingThreshold >0</discardingThreshold>
    <queueSize>512</queueSize>
    <!--这里指定了一个已有的Appender-->
    <appender-ref ref ="FILE"/>
</appender>
```

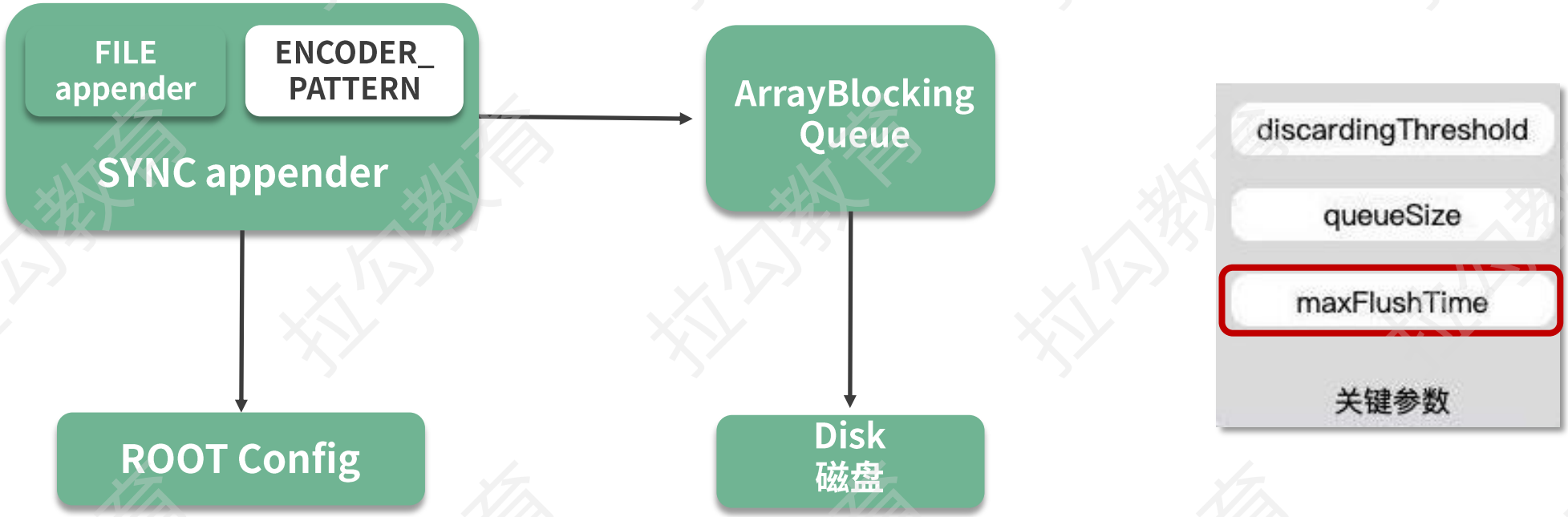
日志缓冲



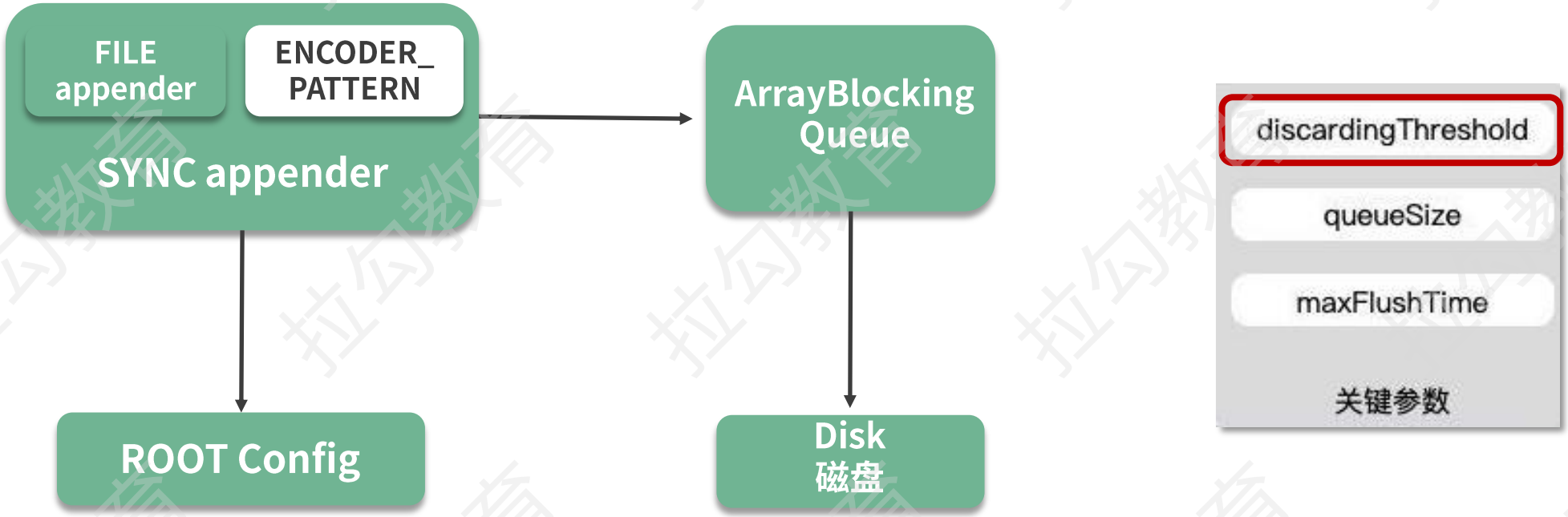
日志缓冲



日志缓冲



日志缓冲

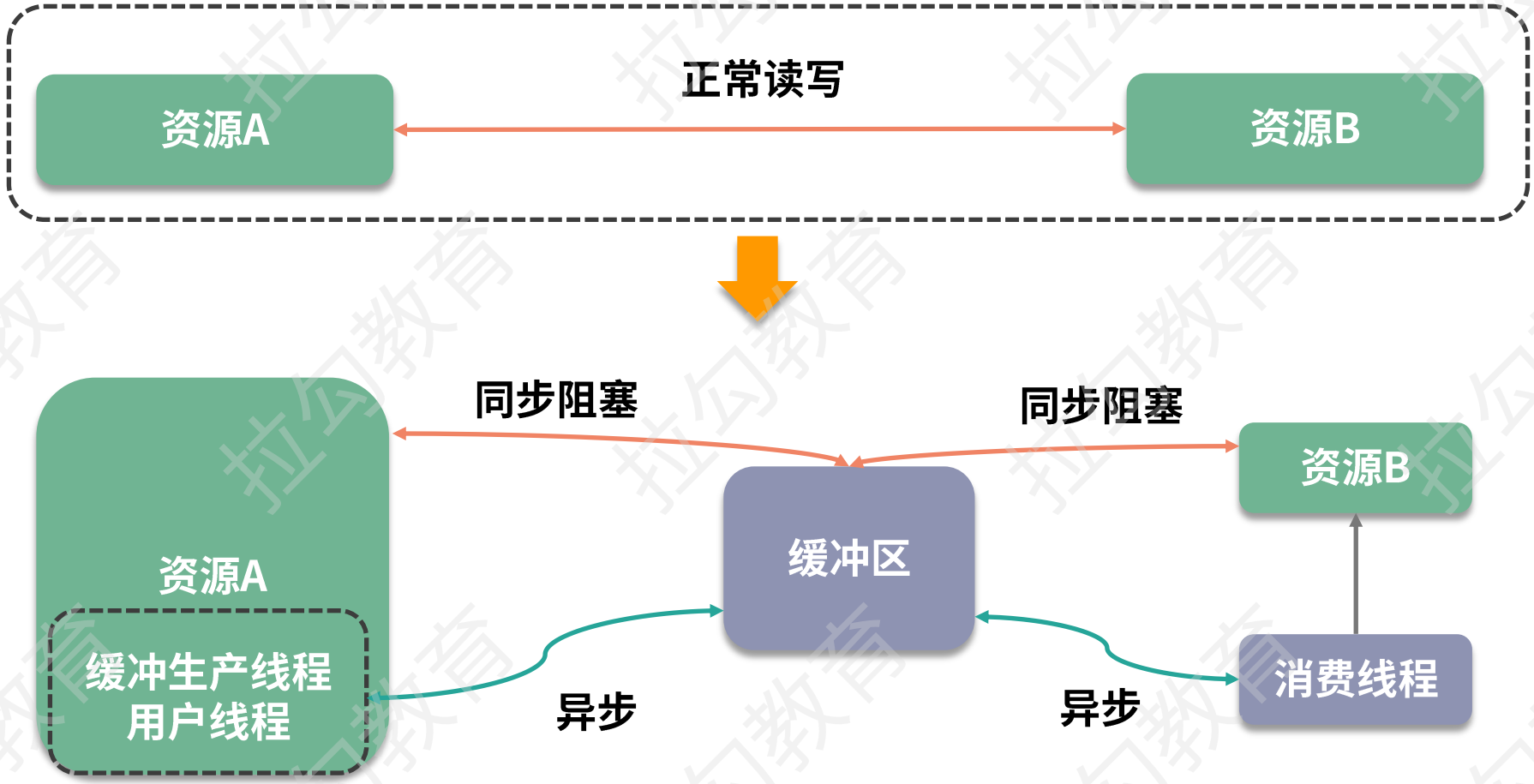


缓冲区优化思路

缓冲区是可以提高性能的

但通常会引入一个异步的问题，使得编程模型变复杂

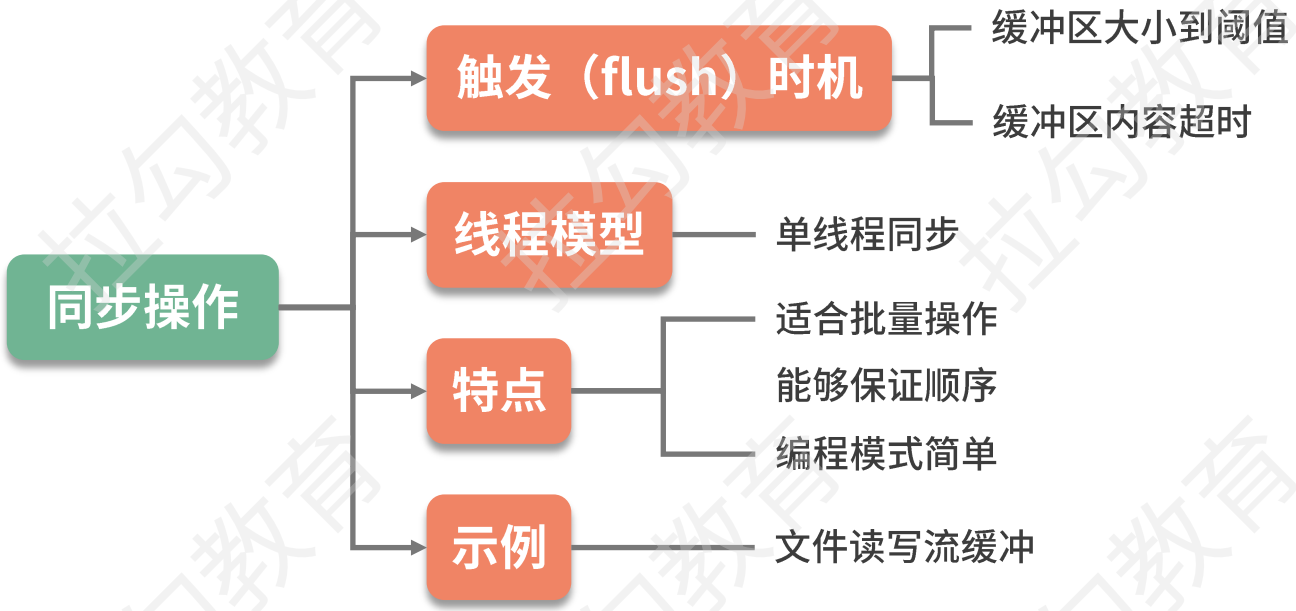
缓冲区优化思路



缓冲区优化思路

1. 同步操作

编程模型相对简单，在一个线程中就可完成，你只需要控制缓冲区的大小，并把握处理的时机



缓冲区优化思路

2. 异步操作

缓冲区的生产者一般是同步调用，但也可以采用异步方式进行填充

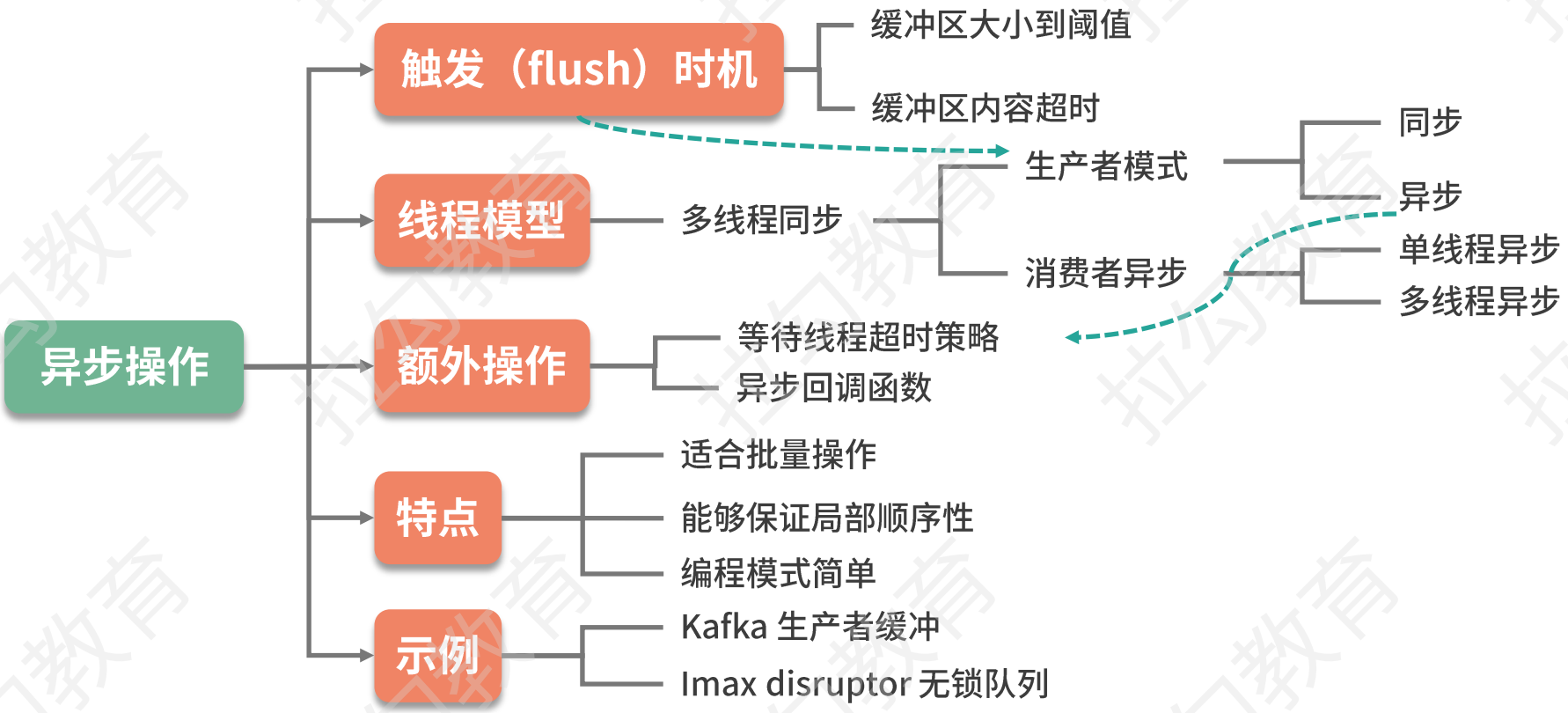
一旦采用异步操作，就涉及到缓冲区满了以后，生产者的一些响应策略

此时应将这些策略抽象出来，根据业务的属性选择

比如**直接抛弃**、**抛出异常**，或者**直接在用户的线程进行等待**

缓冲区优化思路

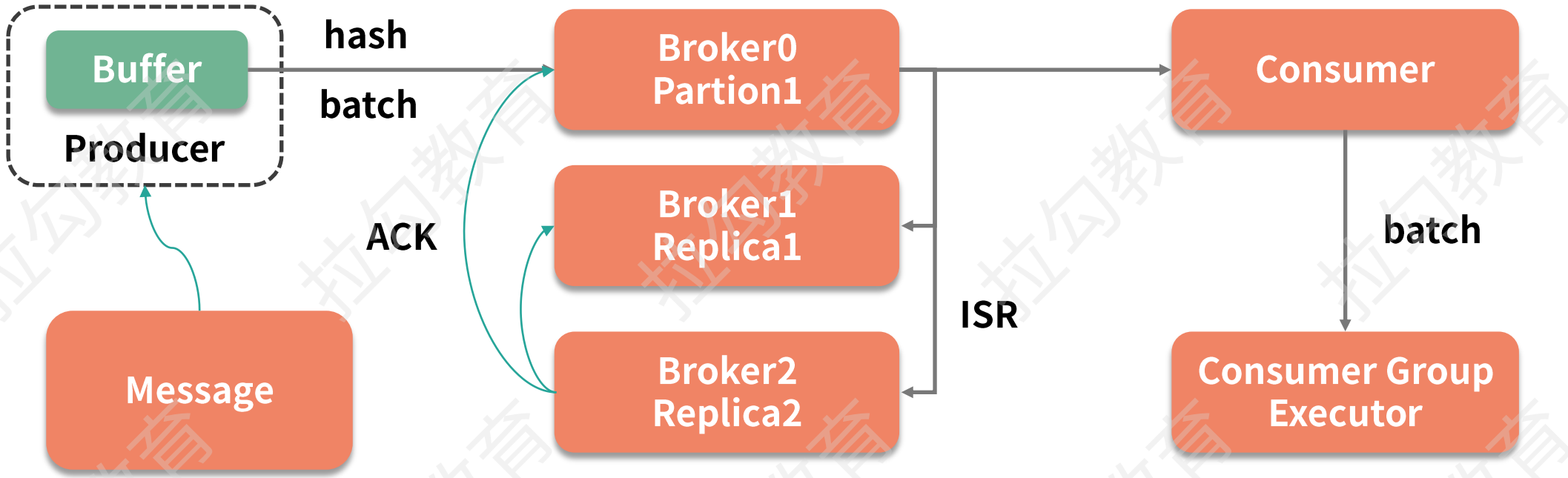
2. 异步操作



缓冲区优化思路

拉勾教育
互联网人实战大学

3. Kafka缓冲区示例



缓冲区优化思路

3. Kafka缓冲区示例

解决的办法有两种：

- 把缓冲区设置得非常小，此时消息会退化成单条发送，这会严重影响性能
- 消息发送前记录一条日志，消息发送成功后，通过回调再记录一条日志

  通过扫描生成的日志，就可以判断哪些消息丢失

缓冲区优化思路

3. Kafka缓冲区示例

Kafka 生产者会影响业务的高可用么？

缓冲区优化思路

3. Kafka缓冲区示例

Kafka 生产者会影响业务的高可用么？

缓冲区大小毕竟是有限制的

如果消息产生得过快

或者生产者与 broker 节点之间有网络问题

缓冲区就会一直处于 full 的状态

缓冲区优化思路

4. 其他做法

- StringBuilder 和 StringBuffer，通过将要处理的字符串缓冲起来，最后完成拼接，提高字符串拼接的性能
- 操作系统在写入磁盘，或者网络 I/O 时，会开启特定的缓冲区，来提升信息流转的效率

  通常可使用 flush 函数强制刷新数据

  比如通过调整 Socket 的参数 SO_SNDBUF 和 SO_RCVBUF 提高网络传输性能
- MySQL 的 InnoDB 引擎，通过配置合理的 innodb_buffer_pool_size，减少换页，增加数据库的性能
- 在一些比较底层的工具中，也会变相地用到缓冲

  比如常见的 ID 生成器，使用方通过缓冲一部分 ID 段，就可以避免频繁、耗时的交互

缓冲区优化思路

5. 注意事项

比较严重就是缓冲区内容丢失

即使使用 addShutdownHook 做了优雅关闭，有些情形依旧难以防范避免

比如机器突然间断电，应用程序进程突然死亡等

所以**内容写入缓冲区之前，需要先预写日志，故障后重启时，就会根据这些日志进行数据恢复**

小结

缓冲区优化是对正常的业务流程进行截断

然后加入缓冲组件的一个操作，分为**同步**和**异步**方式，其中**异步方式的实现难度相对更高**

大多数组件都可以通过设置一些参数，来控制缓冲区大小，从而取得较大的性能提升

注意某些极端场景（断电、异常退出、kill -9等）可能会造成数据丢失

Next：第07讲《工具实践：基准测试 JMH，精确测量方法性能》

拉勾教育

—互联网人实战大学—

下载「拉勾教育App」
获取更多内容

L / A / G / O / U