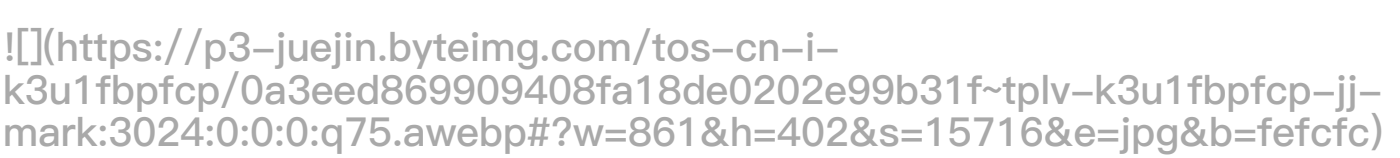
一招MAX降低10倍，现在它是我的了| 京东零售技术团队

一.背景

性能优化是一场永无止境的旅程。

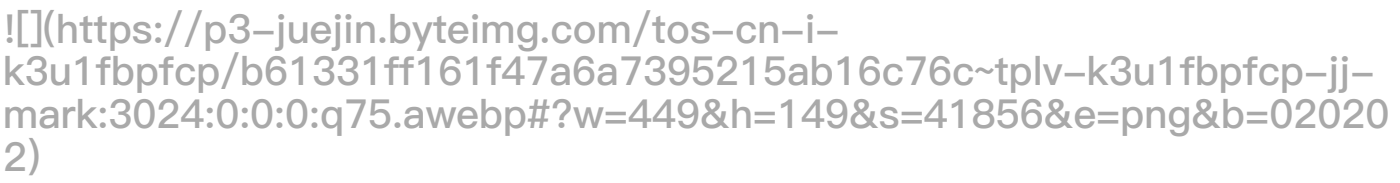
到家门店系统，作为到家核心基础服务之一，门店C端接口有着调用量高，性能要求高的特点。

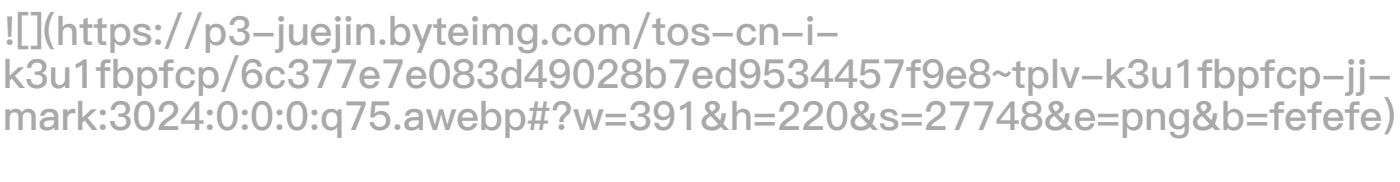
C端服务经过演进，核心接口先查询本地缓存，如果本地缓存没有命中，再查询Redis。本地缓存命中率99%，服务性能比较平稳。



随着门店数据越来越多，本地缓存容量逐渐增大到3G左右。虽然对垃圾回收器和JVM参数都进行调整，**由于本地缓存数据量越来越大，本地缓存数据对于应用GC的影响越来越明显**，Y**GC平均耗时****100ms**，**特别是大促期间调用方接口毛刺感知也越来越明显**。

由于本地缓存在每台机器上容量是固定的，即便是将机器扩容，对与GC毛刺也没有明显效果。





二.初识此物心已惊-OHC初识
==========

本地缓存位于应用程序的内存中，读取和写入速度非常快，可以快速响应请求,无需额外的网络通信,但是**一般本地缓存存在JVM内，数据量过多会影响GC，造成GC频率、耗时增加**；如果用Redis的话有网络通信的开销。

框架	简介	特点	堆外缓存	性能（一般情况）
Guava Cache	Guava Cache是Google的本地缓存库，提供了基本的缓存功能。它简单易用、轻量级，并支持基本的缓存操作。	·支持最大容量限制 ·支持两种过期删除策略（插入时间和访问时间） ·支持简单的统计功能 ·基于

LRU算法实现 | 不支持 | 性能中等 |
Caffeine	Caffeine是一个高性能的本地缓存库，提供了丰富的功能和配置选项。它支持高并发性能、低延迟和一些高级功能，如缓存过期、异步刷新和缓存统计等。	·提供了丰富的功能和配置选项；高并发性能和低延迟；支持缓存过期、异步刷新和缓存统计等功能； ·基于java8实现的新一代缓存工具，缓存性能接近理论最优。 ·可以看作是Guava Cache的增强版，功能上两者类似，不同的是Caffeine采用了一种结合LRU、LFU优点的算法：W-TinyLFU，在性能上有明显的优越性	不支持	性能出色
Ehcache	Encache是一个纯Java的进程内缓存框架，具有快速、精干等特点，是Hibernate中默认的CacheProvider。同Caffeine和Guava Cache相比，Encache的功能更加丰富，扩展性更强	·支持多种缓存淘汰算法，包括LRU、LFU和FIFO ·缓存支持堆内存储、堆外存储、磁盘存储（支持持久化）三种 ·支持多种集群方案，解决数据共享问题	支持	性能一般
OHC	OHC（Off-Heap Cache）是一个高性能的堆外缓存库，专为高并发和低延迟而设计。它使用堆外内存和自定义的数据结构来提供出色的性能	**·针对高并发和低延迟进行了优化；使用自定义数据结构和无锁并发控制；较低的GC开销； ·在高并发和低延迟的缓存访问场景下表现出色**	支持	性能最佳

通过对本地缓存的调研，**堆外缓存**可以很好兼顾上面的问题。堆外缓存把数据放在JVM堆外的，缓存数据对GC影响较小，同时它是在机器内存中的，相对与Redis也没有网络开销，最终选择OHC。

三.习得技能心自安-OHC使用
==========

talk is cheap, show me the code! OCH是骡子是马我们遛一遛。

1.引入POM

OHC 存储的是二进制数组，需要实现OHC序列化接口，**将缓存数据与二进制数组之间序列化和反序列化**。

这里使用的是**Protostuff**，当然也可以使用**kryo**、Hession等，通过压测验证选择适合的序列化框架。

```
<!--OHC相关-->
<dependency>
<groupId>org.caffinitas.ohc</groupId>
<artifactId>ohc-core</artifactId>
<version>0.7.4</version>
</dependency>

<!--OHC 存储的是二进制数组，所以需要实现OHC序列化接口，将缓存数据与二进制数组之间序列化和反序列化-->
<!--这里使用的是protostuff，当然也可以使用kryo、Hession等，通过压测验证选择适合的-->
<dependency>
<groupId>io.protostuff</groupId>
<artifactId>protostuff-core</artifactId>
<version>1.6.0</version>
</dependency>
<dependency>
<groupId>io.protostuff</groupId>
<artifactId>protostuff-runtime</artifactId>
<version>1.6.0</version>
</dependency>
```

2.创建OHC缓存

OHC缓存创建

```
OHCache<String, XxxxInfo> basicStoreInfoCache =
OHCacheBuilder.<String, XxxxInfo>newBuilder()
                    .keySerializer(new OhcStringSerializer()) //key的序列化器
                    .valueSerializer(new OhcProtostuffXxxxInfoSerializer())
//value的序列化器
                    .segmentCount(512) // 分段数量 默认=2*CPU核数
                    .hashTableSize(100000)// 哈希表大小 默认=8192
                    .capacity(1024 * 1024 * 1024) //缓存容量 单位B 默认64MB
```

```
                .eviction(Eviction.LRU) // 淘汰策略 可选
LRU\W_TINY_LFU\NONE
                .timeouts(false) //不使用过期时间，根据业务自己选择
                .build();

```

自定义序列化器，这里key-String 序列化器，这里直接**复用OCH源码中测试用例的String序列化器**；

value-自定义对象序列化器，这里用Protostuff实现,也可以自己选择使用kryo、Hession等实现；

```
//key-String 序列化器，这里直接复用OCH源码中测试用例的String序列化器
public class OhcStringSerializer implements CacheSerializer<String> {

    @Override
    public int serializedSize(String value) {
        return writeUTFLen(value);
    }

    @Override
    public void serialize(String value, ByteBuffer buf) {
        // 得到字符串对象UTF-8编码的字节数组
        byte[] bytes = value.getBytes(Charsets.UTF_8);
        buf.put((byte) ((bytes.length >>> 8) & 0xFF));
        buf.put((byte) ((bytes.length >>> 0) & 0xFF));
        buf.put(bytes);
    }


    @Override
    public String deserialize(ByteBuffer buf) {
        int length = (((buf.get() & 0xff) << 8) + ((buf.get() & 0xff) << 0));
        byte[] bytes = new byte[length];
        buf.get(bytes);
        return new String(bytes, Charsets.UTF_8);
    }
    static int writeUTFLen(String str) {
        int strlen = str.length();
        int utflen = 0;
        int c;

        for (int i = 0; i < strlen; i++) {
```

```
            c = str.charAt(i);
            if ((c >= 0x0001) && (c <= 0x007F)){
                utflen++;}
            else if (c > 0x07FF){
                utflen += 3;}
            else{
                utflen += 2;
            }
        }

        if (utflen > 65535) {
            throw new RuntimeException("encoded string too long: " +
utflen + " bytes");
        }
        return utflen + 2;
    }
}


//value-自定义对象序列化器，这里用Protostuff实现,可以自己选择使用
kryo、Hession等实现
public class OhcProtostuffXxxxInfoSerializer implements
CacheSerializer<XxxxInfo> {

    /**
     * 将缓存数据序列化到 ByteBuffer 中，ByteBuffer是OHC管理的堆外内存
区域的映射。
     */
    @Override
    public void serialize(XxxxInfo t, ByteBuffer byteBuffer) {
        byteBuffer.put(ProtostuffUtils.serialize(t));
    }
    /**
     * 对堆外缓存的数据进行反序列化
     */
    @Override
    public XxxxInfo deserialize(ByteBuffer byteBuffer) {
        byte[] bytes = new byte[byteBuffer.remaining()];
        byteBuffer.get(bytes);
        return ProtostuffUtils.deserialize(bytes, XxxxInfo.class);
    }

    /**
     * 计算字序列化后占用的空间
     */
    @Override
    public int serializedSize(XxxxInfo t) {
        return ProtostuffUtils.serialize(t).length;
```

```
    }
}
```

为了方便调用和序列化封装为工具类，同时对代码通过**FastThreadLocal进行优化，提升性能**。

```
public class ProtostuffUtils {

    /**
     * 避免每次序列化都重新申请Buffer空间，提升性能
     */
    private static final FastThreadLocal<LinkedBuffer> bufferPool = new FastThreadLocal<LinkedBuffer>() {
        @Override
        protected LinkedBuffer initialValue() throws Exception {
            return LinkedBuffer.allocate(4 * 2 * LinkedBuffer.DEFAULT_BUFFER_SIZE);
        }
    };

    /**
     * 缓存Schema
     */
    private static Map<Class<?>, Schema<?>> schemaCache = new ConcurrentHashMap<>();

    /**
     * 序列化方法，把指定对象序列化成字节数组
     */
    @SuppressWarnings("unchecked")
    public static <T> byte[] serialize(T obj) {
        Class<T> clazz = (Class<T>) obj.getClass();
        Schema<T> schema = getSchema(clazz);
        byte[] data;
        LinkedBuffer linkedBuffer = null;
        try {
            linkedBuffer = bufferPool.get();
            data = ProtostuffIOUtil.toByteArray(obj, schema, linkedBuffer);
        } finally {
            if (Objects.nonNull(linkedBuffer)) {
                linkedBuffer.clear();
            }
        }
```

```
        return data;
    }

    /**
     * 反序列化方法，将字节数组反序列化成指定Class类型
     */
    public static <T> T deserialize(byte[] data, Class<T> clazz) {
        Schema<T> schema = getSchema(clazz);
        T obj = schema.newMessage();
        ProtostuffIOUtil.mergeFrom(data, obj, schema);
        return obj;
    }

    @SuppressWarnings("unchecked")
    private static <T> Schema<T> getSchema(Class<T> clazz) {
        Schema<T> schema = (Schema<T>) schemaCache.get(clazz);
        if (Objects.isNull(schema)) {
            schema = RuntimeSchema.getSchema(clazz);
            if (Objects.nonNull(schema)) {
                schemaCache.put(clazz, schema);
            }
        }

        return schema;
    }
}
```

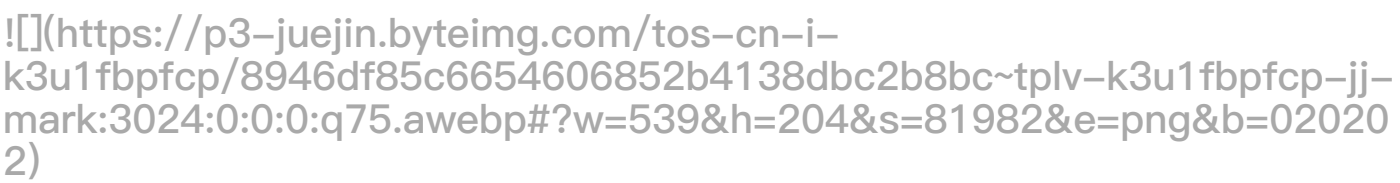
3.压测及参数调整

通过压测并逐步调整OHC配置常见参数（**segmentCount、hashTableSize、eviction，参数含义见附录**）

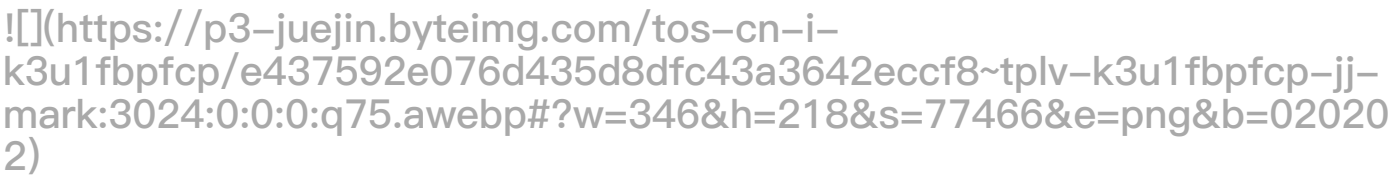
MAX对比降低10倍

GC时间对比降低10倍

优化前



优化后



4.OHC缓存状态监控

OHC缓存的**命中次数、内存使用状态等存储在OHCacheStats中，可以通过OHCache.stats()获取**。

OHCacheStates信息：

> `hitCount`：缓存命中次数，表示从缓存中成功获取数据的次数。`missCount`：缓存未命中次数，表示尝试从缓存中获取数据但未找到的次数。`evictionCount`：缓存驱逐次数，表示因为缓存空间不足而从缓存中移除的数据项数量。`expireCount`：缓存过期次数，表示因为缓存数据过期而从缓存中移除的数据项数量。`size`：缓存当前存储的数据项数量。`capacity`：缓存的最大容量，表示缓存可以存储的最大数据项数量。`free`：缓存剩余空闲容量，表示缓存中未使用的可用空间。`rehashCount`：重新哈希次数，表示进行哈希表重新分配的次数。`put(add/replace/fail)`：数据项添加/替换/失败的次数。`removeCount`：缓存移除次数，表示从缓存中移除数据项的次数。`segmentSizes(#/min/max/avg)`：段大小统计信息，包括段的数量、最小大小、最大大小和平均大小。`totalAllocated`：已分配的总内存大小，表示为负数时表示未知。`lruCompactions`：LRU 压缩次数，表示进行 LRU 压缩的次数。

通过**定期采集**OHCacheStates信息，来**监控本地缓存数据、** 命中率=[命中次数 / (命中次数 + 未命中次数)]等 **，并添加相关报警**。同时通过缓存状态信息，来判断过期策略、段数、容量等设置是否合理，命中率是否符合预期等。

四.剖析根源见真谛-OHC原理
==========

堆外缓存框架（Off-Heap Cache）是将缓存数据存储在 JVM 堆外的内存区域，而不是存储在 JVM 堆中。在 OHC（Off-Heap Cache）中，数据也是存储在堆外的内存区域。

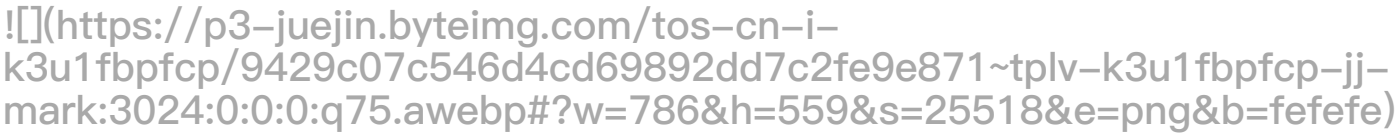
具体来说，OHC 使用 DirectByteBuffer 来分配堆外内存，并将缓存数据存储在这些 DirectByteBuffer 中。

DirectByteBuffer 在 JVM 堆外的内存区域中分配一块连续的内存空间，缓存数据被存储在这个内存区域中。这使得 OHC 在处理大量数据时具有更高的性能和效率，因为它可以**避免 JVM 堆的垃圾回收和堆内存限制**。

OHC 核心OHCache接口提供了两种实现：

•**OHCacheLinkedImpl**： 实现为每个条目单独分配堆外内存，最适合中型和大型条目。

•OHCacheChunkedImpl：实现为每个散列段作为一个整体分配堆外内存，并且适用于小条目。（实验性的，不推荐,不做）



可以看到OHCacheLinkedImpl中包含多个段，每个段用OffHeapLinkedMap来表示。同时，OHCacheLinkedImpl将Java对象序列化成字节数组存储在堆外，在该过程中需要使用用户自定义的CacheSerializer。

OHCacheLinkedImpl的主要工作流程如下：

1.计算key的hash值，根据hash值计算段号，确定其所处的OffHeapLinkedMap

2.从OffHeapLinkedMap中获取该键值对的堆外内存地址(指针)

3.对于get操作，从指针所指向的堆外内存读取byte[]，把byte[]反序列化成对象

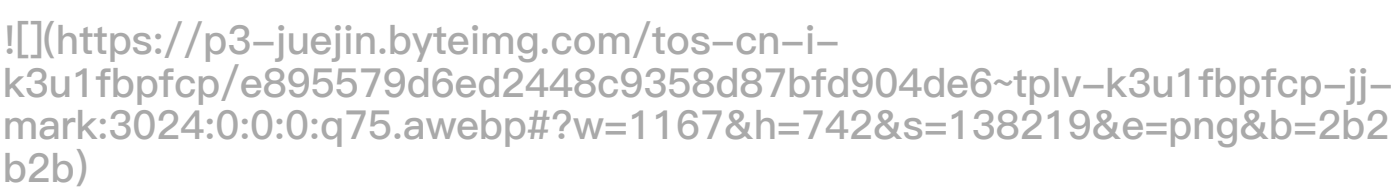
4.对于put操作，把对象序列化成byte[]，并写入指针所指向的堆外内存

可以将OHC理解为一个key-value结果的map,只不过这个map数据存储是指向堆外内存的内存指针。

指针在堆内，指针指向的缓存数据存储在堆外。那么OHC最核心的其实就是**对堆外内存的地址引用的put和get以及发生在其中内存空间**的操作了。

对OHCacheLinkedImpl的put、get本地调试

1.put



put核心操作就是

1.申请堆外内存

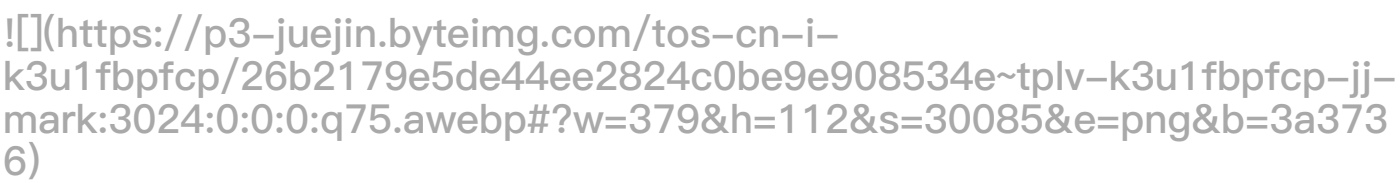
2.将申请地址存入map;

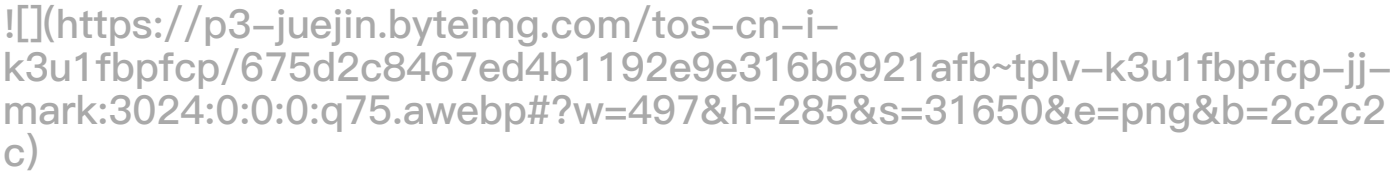
3.异常时释放内存

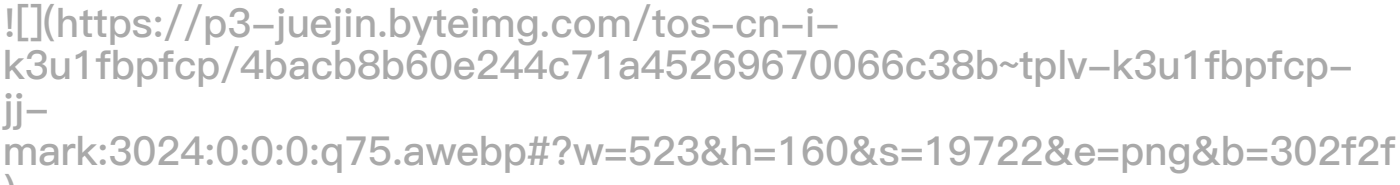
第2步其实就是map数据更新、扩容等的一些实现这里不在，我们重点怎么申请和释放内存的

1.申请内存

通过深入代码发现是调用的IAllocator接口的JNANativeAllocator实现类，最后调用的是**com.sun.jna.Native#malloc**实现

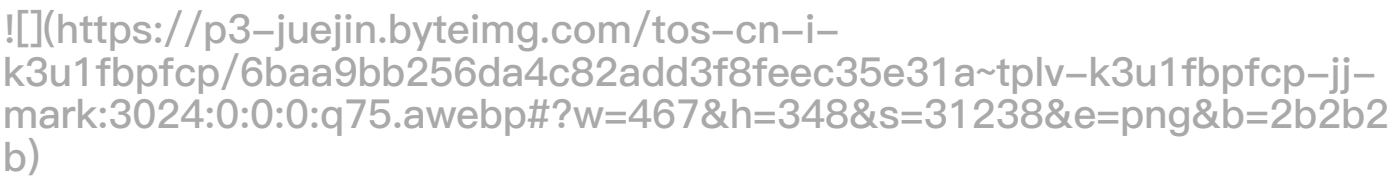




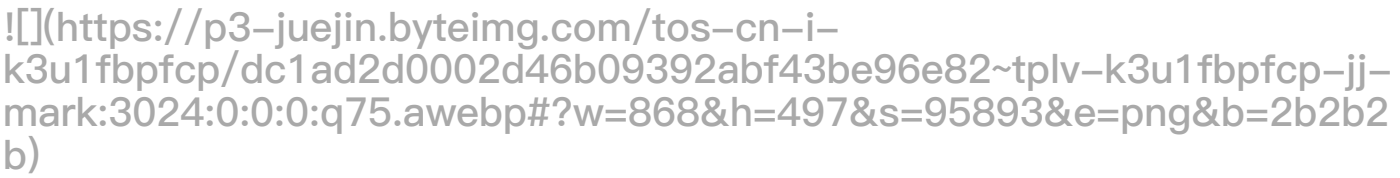


2.释放内存

通过上面可知释放内存操作的代码



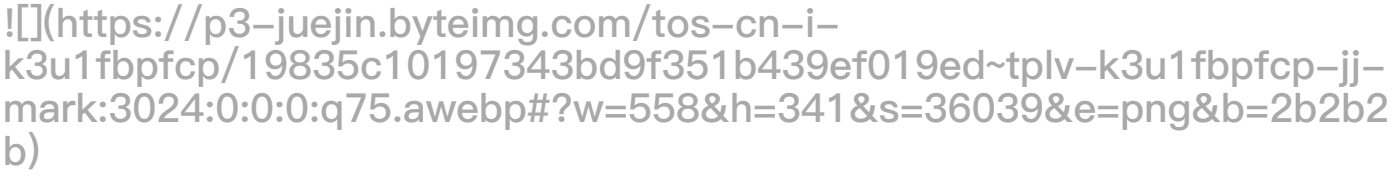
2.get



3.Q&A

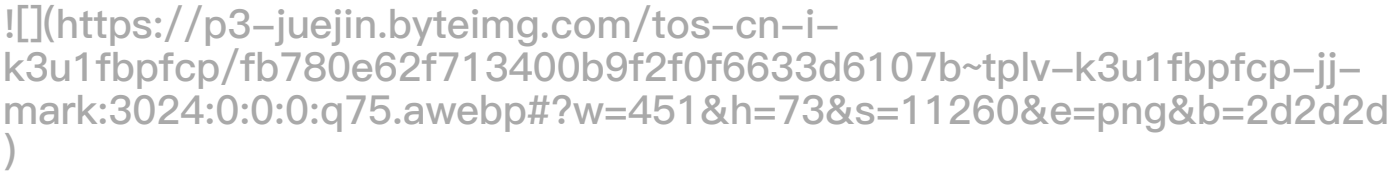
在put操作时，上面看到IAllocator有两个实现类，**JNANativeAllocator和UnsafeAllocator两个实现类，他们有什么区别？为什么使用JNANativeAllocator?**

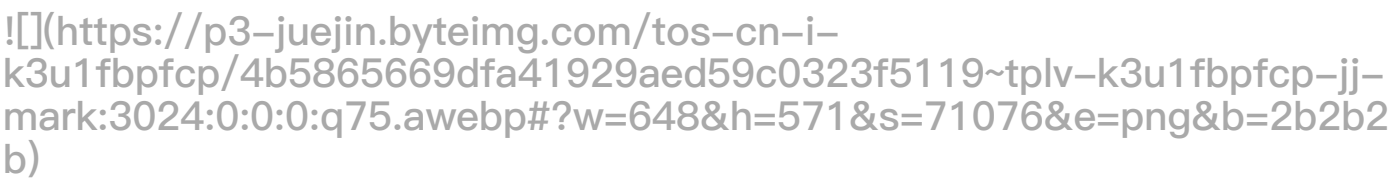
区别：UnsafeAllocator对内存操作使用的是**Unsafe**类



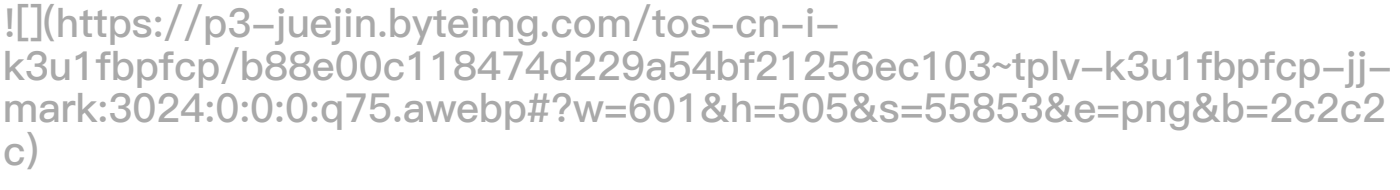
为什么使用JNANativeAllocator：**Native比Unsafe性能更好，差3倍左右，OHC默认使用JNANativeAllocator**；

在日常我们知道通过ByteBuffer#allocateDirect(int capacity)也可以直接申请堆外内存，通过ByteBuffer源码可以看到内部使用的就是Unsafe类





可以看到，同时DirectByteBuffer内部会调用 Bits.reserveMemory(size, cap);



Bits.reserveMemory方法中，当内存不足时可能会触发fullgc，多个申请内存的线程同时遇到这种情况时，对于服务来说是不能接受的，所以这也是OHC自己进行堆外内存管理的原因。

如果自己进行实现堆外缓存框架，要考虑上面这种情况。

五.总结

1.OHC使用建议

1.对于OHC的参数配置、序列化器的选择，没有固定的推荐。可以**通过压测逐步调整到最优**。

2.由于OHC需要把key和value序列化成字节数组存储到堆外，因此需要**选择合适的序列化工具**。

3.在存储每个键值对时，会调用CacheSerializer#serializedSize计算序列化后的内存空间占用，从而申请堆外内存。另外，在真正写入堆外时，会调用CacheSerializer#serialize真正进行序列化。因此，务必在这两个方法中**使用相同的序列化方法，防止序列化的大小与计算出来的大小不一致，导致内存存不下或者多申请，浪费内存空间**。

2.缓存优化建议

1.当本地缓存影响GC时，可以考虑**使用OHC减少本地缓存对GC的影响**；

2.区分热点数据，对**缓存数据进行多级拆分**，如堆内->堆外->分布式缓存(Reids)等；

3.将较大**缓存对象拆分**或者按照业务维度将不同热点数据缓存到不同介质中，减少单一存储介质压力；

4.**减小缓存对象大小**，如缓存JSON字符，可对字段名进行缩写 ,减少存储数据量，降低传输数据量，同时也能保证数据一定的私密性。

> 对象：{”paramID”:1,”paramName”:”John Doe”} 正常JSON字符串：{”paramID”:1,”paramName”:”John Doe”} 压缩字段名JSON字符串：{”a”:1,”b”:”John Doe”}

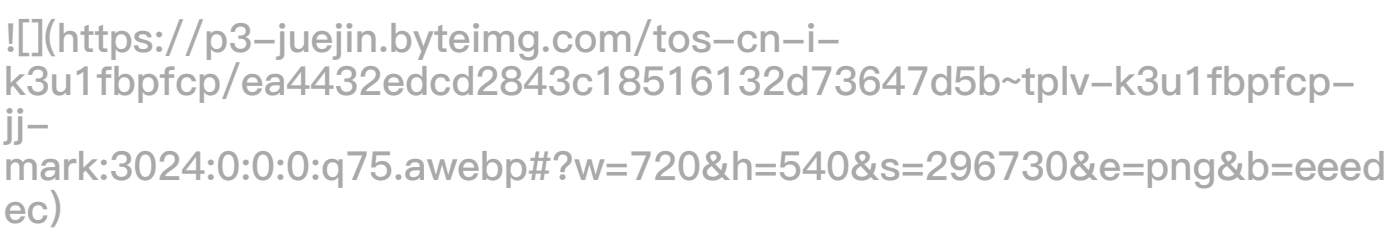
Hold hold , One more thing....

==========

在使用Guava时，存储**25w**个缓存对象数据占用空间**485M**

使用OHCache时，储存**60w**个缓存对象数据占用数据**387M**

为什么存储空间差别那么多呐?

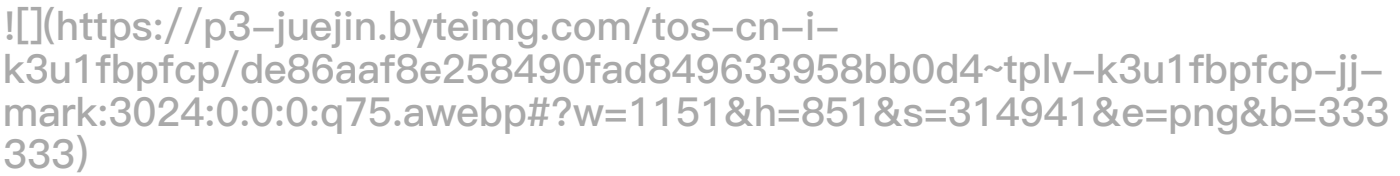


Guava 存储的对象是在堆内存中的，对象在 JVM 堆中存储时，它们会占用一定的内存空间，并且会包含对象头、实例数据和对齐填充等信息。对象的大小取决于其成员变量的类型和数量，以及可能存在的对齐需求。同时当对象被频繁创建和销毁时，可能会产生内存碎片。

而 OHC 它将对象存储在 JVM 堆外的直接内存中。由于堆外内存不受 Java 堆内存大小限制，OHC 可以更有效地管理和利用内存。此外，OHC 底层存储字节数组，存储字节数组相对于直接存储对象，可以减少对象的创建和销毁，在一些场景下，直接操作字节数组可能比操作对象更高效，因为**它避免了对象的额外开销，如对象头和引用，减少额外的开销**。同时将对象序列化为二进制数组存储，**内存更加紧凑，减少内存碎片的产生**。

综上所述，OHC 在存储大量对象时能够更有效地利用内存空间，相对于 Guava 在内存占用方面具有优势。

另外一个原因，不同序列化框架性能不同，**将对象序列化后的占用空间的大小也不同**。



参考及附录
=====

1.OHC常见参数

name	**默认值**	**描述**
keySerializer	需要开发者实现	Key序列化实现
valueSerializer	需要开发者实现	Value序列化实现
capacity	64MB	缓存容量单位B

segmentCount	2倍CPU核心数	分段数量
hashTableSize	8192	哈希表的大小
loadFactor	0.75	负载因子
maxEntrySize	capacity/segmentCount	缓存项最大字节限制
throwOOME	false	内存不足是否抛出OOM
hashAlgorighm	MURMUR3	hash算法，可选性MURMUR3、 CRC32, CRC32C （Jdk9以上支持）
unlocked	false	读写数据是否加锁，默认是加锁
eviction	LRU	驱逐策略，可选项：LRU、W_TINY_LFU、NONE
frequencySketchSize	hashTableSize数量	W_TINY_ LFU frequency sketch 的大小
edenSize	0.2	W_TINY_LFU 驱逐策略下使用

2.JNI faster than Unsafe?

[mail.openjdk.org/pipermail/h...](http://cxyroad.com/”https://mail.openjdk.org/pipermail/hotspot-dev/2015-February/017089.html”)

3.OHC源码

[github.com/snazy/ohc](http://cxyroad.com/”https://github.com/snazy/ohc”)

4.参考文档

•序列化框架对比

•[Java堆外缓存OHC在马蜂窝推荐引擎的应用](http://cxyroad.com/”https://mp.weixin.qq.com/s/PEqWHct0K4LvzgeYJobWpA”)

•“堆外缓存”这玩意是真不错，我要写进简历了。

作者：京东零售 赵雪召

来源：京东云开发者社区

原文链接: https://juejin.cn/post/7366899841455423498