

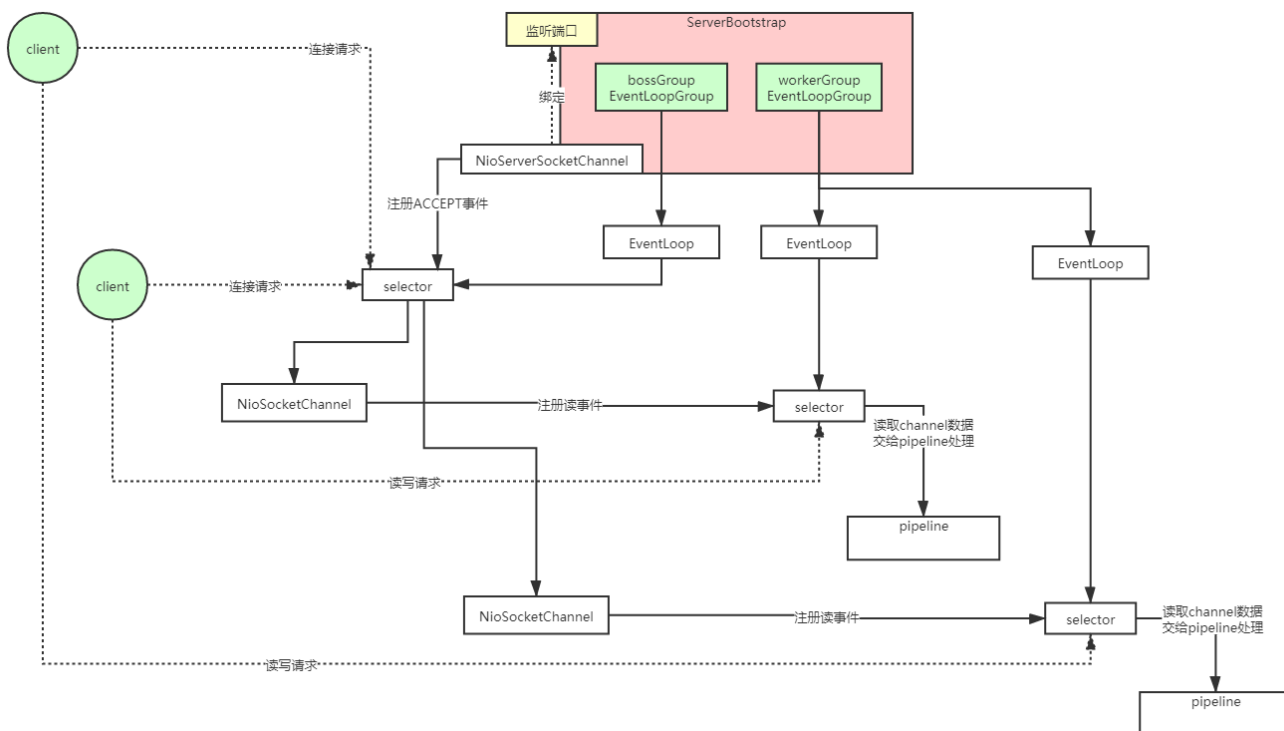
为什么要看源码：

- 1、**提升技术功底**：学习源码里的优秀设计思想，比如一些疑难问题的解决思路，还有一些优秀的设计模式，整体提升自己的技术功底
- 2、**深度掌握技术框架**：源码看多了，对于一个新技术或框架的掌握速度会有大幅提升，看下框架demo大致就能知道底层的实现，技术框架更新再快也不怕
- 3、**快速定位线上问题**：遇到线上问题，特别是框架源码里的问题(比如bug)，能够快速定位，这就是相比其他没看过源码的人的优势
- 4、**对面试大有裨益**：面试一线互联网公司对于框架技术一般都会问到源码级别的实现
- 5、**知其然知其所以然**：对技术有追求的人必做之事，使用了一个好的框架，很想知道底层是如何实现的
- 6、**拥抱开源社区**：参与到开源项目的研发，结识更多大牛，积累更多优质人脉

看源码方法(凭经验去猜)：

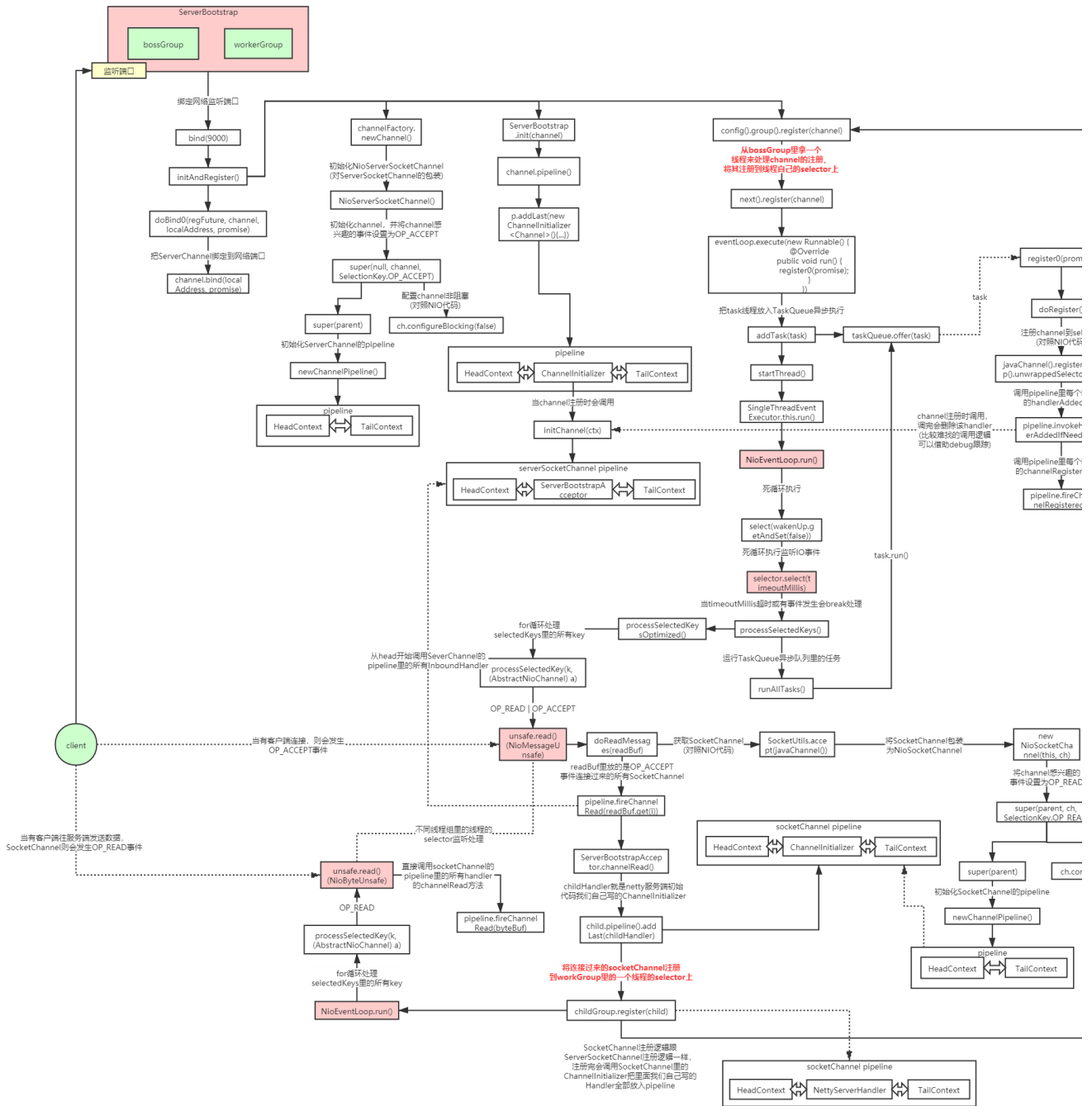
- 1、**先使用**：先看官方文档快速掌握框架的基本使用
- 2、**抓主线**：找一个demo入手，顺藤摸瓜快速静态看一遍框架的主线源码(抓大放小)，画出源码主流程图，切勿一开始就陷入源码的细枝末节，否则会把自已绕晕
- 3、**画图做笔记**：总结框架的一些核心功能点，从这些功能点入手深入到源码的细节，**边看源码边画源码走向图**，并对关键源码的理解做笔记，把源码里的闪光点都记录下来，后续借鉴到工作项目中，理解能力强的可以直接看静态源码，也可以边看源码边debug源码执行过程，观察一些关键变量的值
- 4、**整合总结**：所有功能点的源码都分析完后，回到主流程图再梳理一遍，争取把自己画的所有图都在脑袋里做一个整合

Netty线程模型图



Netty线程模型源码剖析图

图链接: <https://www.processon.com/view/link/5dee0943e4b079080a26c2ac>



Netty高并发高性能架构设计精髓

- 主从Reactor线程模型
- NIO多路复用非阻塞
- 无锁串行化设计思想
- 支持高性能序列化协议
- 零拷贝(直接内存的使用)
- ByteBuf内存池设计
- 灵活的TCP参数配置能力
- 并发优化

无锁串行化设计思想

在大多数场景下，并行多线程处理可以提升系统的并发性能。但是，如果对于共享资源的并发访问处理不当，会带来严重的锁竞争，这最终会导致性能的下降。为了尽可能的避免锁竞争带来的性能损耗，可以通过串行化设计，即消息的处理尽可能在同一个线程内完成，期间不进行线程切换，这样就避免了多线程竞争和同步锁。NIO的多路复用就是一种无锁串行化的设计思想(理解下Redis和Netty的线程模型)

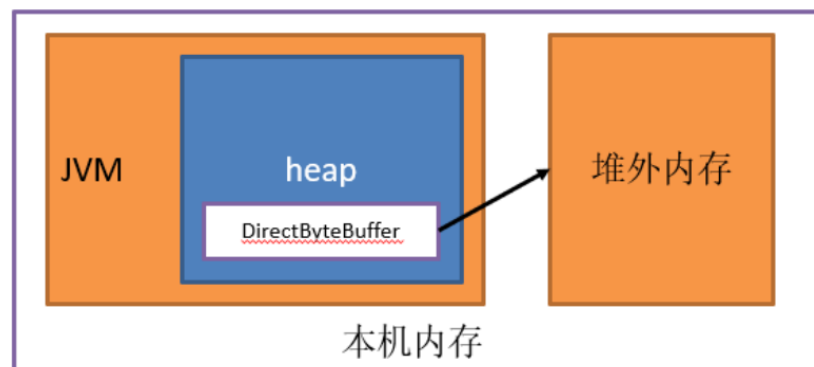
为了尽可能提升性能，Netty采用了串行无锁化设计，在IO线程内部进行串行操作，避免多线程竞争导致的性能下降。表面上看，串行化设计似乎CPU利用率不高，并发程度不够。但是，通过调整NIO线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部

无锁化的串行线程设计相比一个队列-多个工作线程模型性能更优。

Netty的NioEventLoop读取到消息之后，直接调用ChannelPipeline的fireChannelRead(Object msg)，只要用户不主动切换线程，一直会由NioEventLoop调用到用户的Handler，期间不进行线程切换，这种串行化处理方式避免了多线程操作导致的锁的竞争，从性能角度看是最优的。

直接内存

直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，某些情况下这部分内存也会被频繁地使用，而且也可能导致OutOfMemoryError异常出现。Java里用DirectByteBuffer可以分配一块直接内存(堆外内存)，元空间对应的内存也叫作直接内存，它们对应的都是机器的物理内存。



```
1 /**
2  * 直接内存与堆内存的区别
3  */
4 public class DirectMemoryTest {
5
6     public static void heapAccess() {
7         long startTime = System.currentTimeMillis();
8         //分配堆内存
9         ByteBuffer buffer = ByteBuffer.allocate(1000);
10        for (int i = 0; i < 100000; i++) {
11            for (int j = 0; j < 200; j++) {
12                buffer.putInt(j);
13            }
14            buffer.flip();
15            for (int j = 0; j < 200; j++) {
16                buffer.getInt();
17            }
18            buffer.clear();
19        }
20        long endTime = System.currentTimeMillis();
21        System.out.println("堆内存访问:" + (endTime - startTime) + "ms");
22    }
23
24    public static void directAccess() {
25        long startTime = System.currentTimeMillis();
26        //分配直接内存
27        ByteBuffer buffer = ByteBuffer.allocateDirect(1000);
28        for (int i = 0; i < 100000; i++) {
29            for (int j = 0; j < 200; j++) {
30                buffer.putInt(j);
31            }
32            buffer.flip();
33            for (int j = 0; j < 200; j++) {
34                buffer.getInt();
35            }
36            buffer.clear();
37        }
38        long endTime = System.currentTimeMillis();
```

```

39 System.out.println("直接内存访问:" + (endTime - startTime) + "ms");
40 }
41
42 public static void heapAllocate() {
43     long startTime = System.currentTimeMillis();
44     for (int i = 0; i < 100000; i++) {
45         ByteBuffer.allocate(100);
46     }
47     long endTime = System.currentTimeMillis();
48     System.out.println("堆内存申请:" + (endTime - startTime) + "ms");
49 }
50
51 public static void directAllocate() {
52     long startTime = System.currentTimeMillis();
53     for (int i = 0; i < 100000; i++) {
54         ByteBuffer.allocateDirect(100);
55     }
56     long endTime = System.currentTimeMillis();
57     System.out.println("直接内存申请:" + (endTime - startTime) + "ms");
58 }
59
60 public static void main(String args[]) {
61     for (int i = 0; i < 10; i++) {
62         heapAccess();
63         directAccess();
64     }
65
66     System.out.println();
67
68     for (int i = 0; i < 10; i++) {
69         heapAllocate();
70         directAllocate();
71     }
72 }
73 }

```

运行结果:

```

76 堆内存访问:44ms
77 直接内存访问:29ms
78 堆内存访问:33ms
79 直接内存访问:19ms
80 堆内存访问:55ms
81 直接内存访问:38ms
82 堆内存访问:39ms
83 直接内存访问:20ms
84 堆内存访问:38ms
85 直接内存访问:18ms
86 堆内存访问:36ms
87 直接内存访问:19ms
88 堆内存访问:34ms
89 直接内存访问:19ms
90 堆内存访问:40ms
91 直接内存访问:20ms
92 堆内存访问:37ms
93 直接内存访问:24ms
94 堆内存访问:59ms
95 直接内存访问:25ms
96
97 堆内存申请:11ms
98 直接内存申请:36ms
99 堆内存申请:13ms

```

```
100 直接内存申请:52ms
101 堆内存申请:62ms
102 直接内存申请:40ms
103 堆内存申请:2ms
104 直接内存申请:37ms
105 堆内存申请:1ms
106 直接内存申请:81ms
107 堆内存申请:2ms
108 直接内存申请:23ms
109 堆内存申请:1ms
110 直接内存申请:31ms
111 堆内存申请:2ms
112 直接内存申请:32ms
113 堆内存申请:7ms
114 直接内存申请:41ms
115 堆内存申请:8ms
116 直接内存申请:142ms
```

从程序运行结果看出直接内存申请较慢，但访问效率高。在java虚拟机实现上，本地IO一般会直接操作直接内存（直接内存=>系统调用=>硬盘/网卡），而非直接内存则需要二次拷贝（堆内存=>直接内存=>系统调用=>硬盘/网卡）。

直接内存分配源码分析：

```
1 public static ByteBuffer allocateDirect(int capacity) {
2     return new DirectByteBuffer(capacity);
3 }
4
5
6 DirectByteBuffer(int cap) { // package-private
7     super(-1, 0, cap, cap);
8     boolean pa = VM.isDirectMemoryPageAligned();
9     int ps = Bits.pageSize();
10    long size = Math.max(1L, (long)cap + (pa ? ps : 0));
11    //判断是否有足够的直接内存空间分配，可通过-XX:MaxDirectMemorySize=<size>参数指定直接内存最大可分配空间，如果不指定默认为最大堆内存大小，
12    //在分配直接内存时如果发现空间不够会显示调用System.gc()触发一次full gc回收掉一部分无用的直接内存的引用对象，同时直接内存也会被释放掉
13    //如果释放完分配空间还是不够会抛出异常java.lang.OutOfMemoryError
14    Bits.reserveMemory(size, cap);
15
16    long base = 0;
17    try {
18        // 调用unsafe本地方法分配直接内存
19        base = unsafe.allocateMemory(size);
20    } catch (OutOfMemoryError x) {
21        // 分配失败，释放内存
22        Bits.unreserveMemory(size, cap);
23        throw x;
24    }
25    unsafe.setMemory(base, size, (byte) 0);
26    if (pa && (base % ps != 0)) {
27        // Round up to page boundary
28        address = base + ps - (base & (ps - 1));
29    } else {
30        address = base;
31    }
32
33    // 使用Cleaner机制注册内存回收处理函数，当直接内存引用对象被GC清理掉时，
34    // 会提前调用这里注册的释放直接内存的Deallocator线程对象的run方法
35    cleaner = Cleaner.create(this, new Deallocator(base, size, cap));
36    att = null;
37 }
```

```

38
39
40 // 申请一块本地内存。内存空间是未初始化的，其内容是无法预期的。
41 // 使用freeMemory释放内存，使用reallocMemory修改内存大小
42 public native long allocateMemory(long bytes);
43
44 // openjdk8/hotspot/src/share/vm/prims/unsafe.cpp
45 UNSAFE_ENTRY(jlong, Unsafe_AllocateMemory(JNIEnv *env, jobject unsafe, jlong size))
46   UnsafeWrapper("Unsafe_AllocateMemory");
47   size_t sz = (size_t)size;
48   if (sz != (julong)size || size < 0) {
49     THROW_0(vmSymbols::java_lang_IllegalArgumentException());
50   }
51   if (sz == 0) {
52     return 0;
53   }
54   sz = round_to(sz, HeapWordSize);
55   // 调用os::malloc申请内存，内部使用malloc这个C标准库的函数申请内存
56   void* x = os::malloc(sz, mtInternal);
57   if (x == NULL) {
58     THROW_0(vmSymbols::java_lang_OutOfMemoryError());
59   }
60   //Copy::fill_to_words((HeapWord*)x, sz / HeapWordSize);
61   return addr_to_java(x);
62 UNSAFE_END

```

使用直接内存的优缺点：

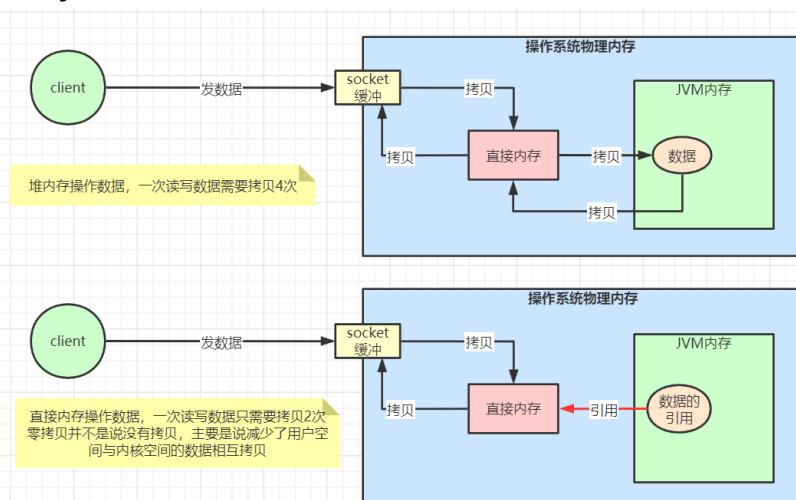
优点：

- 不占用堆内存空间，减少了发生GC的可能
- java虚拟机实现上，本地IO会直接操作直接内存（直接内存=>系统调用=>硬盘/网卡），而非直接内存则需要二次拷贝（堆内存=>直接内存=>系统调用=>硬盘/网卡）

缺点：

- 初始分配较慢
- 没有JVM直接帮助管理内存，容易发生内存溢出。为了避免一直没有FULL GC，最终导致直接内存把物理内存耗完。我们可以指定直接内存的最大值，通过-XX: MaxDirectMemorySize来指定，当达到阈值的时候，调用system.gc来进行一次FULL GC，间接把那些没有被使用的直接内存回收掉。

Netty零拷贝



Netty的接收和发送ByteBuf采用DIRECT BUFFERS，使用堆外**直接内存**进行Socket读写，不需要进行字节缓冲区的二次拷贝。

如果使用传统的JVM堆内存（HEAP BUFFERS）进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中，然后才能写入Socket中。JVM堆内存的数据是不能直接写入Socket中的。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。

可以看看netty的读写源码，比如read源码NioByteUnsafe.read()

```

java x AbstractNioByteChannel.java x DefaultMaxBytesRecvByteBufAllocator.java x RecvByteBufAllocator.java x
@Override
public final void read() {
    final ChannelConfig config = config();
    if (shouldBreakReadReady(config)) {
        clearReadPending();
        return;
    }
    final ChannelPipeline pipeline = pipeline();
    final ByteBufAllocator allocator = config.getAllocator();
    final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();
    allocHandle.reset(config);

    ByteBuf byteBuf = null;
    boolean close = false;
    try {
        do {
            byteBuf = allocHandle.allocate(allocator);
            allocHandle.lastBytesRead(doReadBytes(byteBuf));
            if (allocHandle.lastBytesRead() <= 0) {
                // nothing was read. release the buffer.
                byteBuf.release();
                byteBuf = null;
            }
        } while (true);
    } catch (IOException e) {
        close = true;
    }

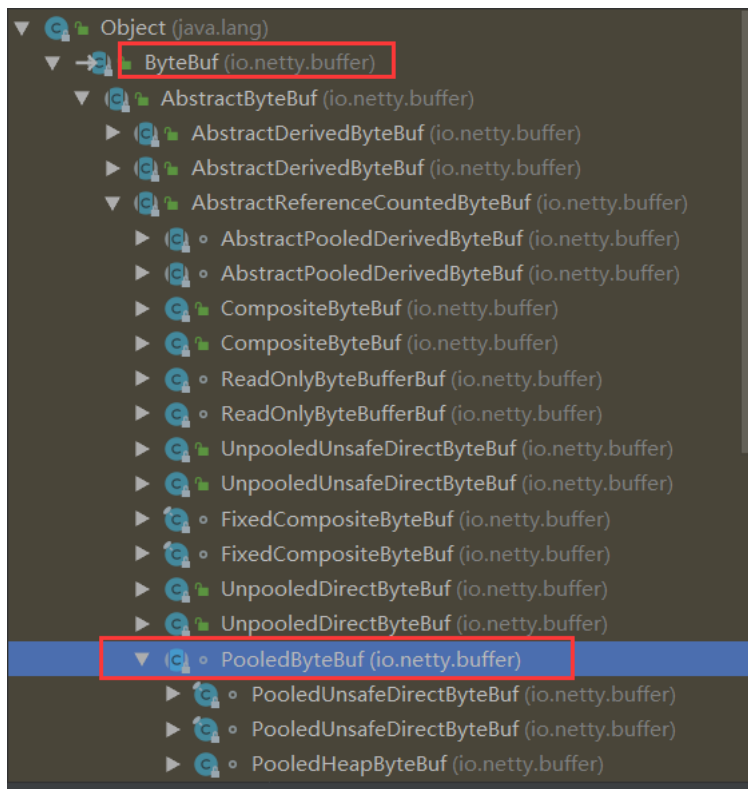
    @Override
    public ByteBuf ioBuffer(int initialCapacity) {
        if (PlatformDependent.hasUnsafe()) {
            return directBuffer(initialCapacity);
        }
        return heapBuffer(initialCapacity);
    }

    private void initMemoryAddress() {
        memoryAddress = PlatformDependent.directBufferAddress(memory) + offset;
    }
}

```

ByteBuf内存池设计

随着JVM虚拟机和JIT即时编译技术的发展，对象的分配和回收是个非常轻量级的工作。但是对于缓冲区Buffer(相当于一个内存块)，情况却稍有不同，特别是对于堆外直接内存的分配和回收，是一件耗时的操作。为了尽量重用缓冲区，Netty提供了基于ByteBuf内存池的缓冲区重用机制。需要的时候直接从池子里获取ByteBuf使用即可，使用完毕之后就重新放回到池子里去。下面我们一起看下Netty ByteBuf的实现：



可以看下netty的读写源码里面用到的ByteBuf内存池，比如read源码NioByteUnsafe.read()

```
java x AbstractNioByteChannel.java x DefaultMaxBytesRecvByteBufAllocator.java x RecvByteBufAllocator.java x
@Override
public final void read() {
    final ChannelConfig config = config();
    if (shouldBreakReadReady(config)) {
        clearReadPending();
        return;
    }
    final ChannelPipeline pipeline = pipeline();
    final ByteBufAllocator allocator = config.getAllocator();
    final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();
    allocHandle.reset(config);

    ByteBuf byteBuf = null;
    boolean close = false;
    try {
        do {
            byteBuf = allocHandle.allocate(allocator);
            allocHandle.lastBytesRead(doReadBytes(byteBuf));
            if (allocHandle.lastBytesRead() <= 0) {
                // nothing was read. release the buffer.
                byteBuf.release();
                byteBuf = null;
            }
        } while (true);
    } catch (IOException e) {
        close = true;
    }

    if (close) {
        byteBuf.release();
    }
}

@Override
public ByteBuf ioBuffer(int initialCapacity) {
    if (PlatformDependent.hasUnsafe()) {
        return directBuffer(initialCapacity);
    }
    return heapBuffer(initialCapacity);
}
```



```

@Override
public ByteBuf directBuffer(int initialCapacity, int maxCapacity) {
    if (initialCapacity == 0 && maxCapacity == 0) {
        return emptyBuf;
    }
    validate(initialCapacity, maxCapacity);
    return newDirectBuffer(initialCapacity, maxCapacity);
}

```

继续看newDirectBuffer方法，我们发现它是一个抽象方法，由AbstractByteBufAllocator的子类负责具体实现，代码如下：

```

return newDirectBuffer(initialCapacity, maxCapacity);
Choose Implementation of AbstractByteBufAllocator.newDirectBu
PooledByteBufAllocator (io.netty.buffer) Maven: io.netty:netty-al
UnpooledByteBufAllocator (io.netty.buffer) Maven: io.netty:netty-al

```

代码跳转到PooledByteBufAllocator的newDirectBuffer方法，从Cache中获取内存区域PoolArena，调用它的allocate方法进行内存分配：

```

@Override
protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {
    PoolThreadCache cache = threadCache.get();
    PoolArena<ByteBuf> directArena = cache.directArena;

    final ByteBuf buf;
    if (directArena != null) {
        buf = directArena.allocate(cache, initialCapacity, maxCapacity);
    } else {
        buf = PlatformDependent.hasUnsafe() ?
            UnsafeByteBufUtil.newUnsafeDirectByteBuf(alloc, this, initialCapacity, maxCapacity) :
            new UnpooledDirectByteBuf(alloc, this, initialCapacity, maxCapacity);
    }

    return toLeakAwareBuffer(buf);
}

```

PoolArena的allocate方法如下：

```

PooledByteBuf<T> allocate(PoolThreadCache cache, int reqCapacity, int maxCapacity) {
    PooledByteBuf<T> buf = newByteBuf(maxCapacity);
    allocate(cache, buf, reqCapacity);
    return buf;
}

```

我们重点分析newByteBuf的实现，它同样是个抽象方法，由子类DirectArena和HeapArena来实现不同类型的缓冲区分配

```

PooledByteBuf<T> allocate(PoolThreadCache cache, int reqCapacity, int maxCapacity) {
    PooledByteBuf<T> buf = newByteBuf(maxCapacity);
    Choose Implementation of PoolArena.newByteBuf(int) (2 found)
    DirectArena in PoolArena (io.netty.buffer) Maven: io.netty:netty-all:4.1.35.Final (netty-all)
    HeapArena in PoolArena (io.netty.buffer) Maven: io.netty:netty-all:4.1.35.Final (netty-all)

```

我们这里使用的是直接内存，因此重点分析DirectArena的实现

```

@Override
protected PooledByteBuf<ByteBuf> newByteBuf(int maxCapacity) {
    if (HAS_UNSAFE) {
        return PooledUnsafeDirectByteBuf.newInstance(maxCapacity);
    } else {
        return PooledDirectByteBuf.newInstance(maxCapacity);
    }
}

```

最终执行了PooledUnsafeDirectByteBuf的newInstance方法，代码如下：

```

static PooledUnsafeDirectByteBuf newInstance(int maxCapacity) {
    PooledUnsafeDirectByteBuf buf = RECYCLER.get();
    buf.reuse(maxCapacity);
    return buf;
}

```

通过RECYCLER的get方法循环使用ByteBuf对象，如果是非内存池实现，则直接创建一个新的ByteBuf对象。

灵活的TCP参数配置能力

合理设置TCP参数在某些场景下对于性能的提升可以起到显著的效果，例如接收缓冲区SO_RCVBUF和发送缓冲区SO_SNDBUF。如果设置不当，对性能的影响是非常大的。通常建议值为128K或者256K。

Netty在启动辅助类ChannelOption中可以灵活的配置TCP参数，满足不同的用户场景。

```

IP_MULTICAST_IF: ChannelOption<NetworkInterface> = valueOf(...)
IP_MULTICAST_LOOP_DISABLED: ChannelOption<Boolean> = valueOf(...)
IP_MULTICAST_TTL: ChannelOption<Integer> = valueOf(...)
IP_TOS: ChannelOption<Integer> = valueOf(...)
MAX_MESSAGES_PER_READ: ChannelOption<Integer> = valueOf(...)
MESSAGE_SIZE_ESTIMATOR: ChannelOption<MessageSizeEstimator> = valueOf(...)
pool: ConstantPool<ChannelOption<Object>> = new ConstantPool<ChannelOption<Object>>()
RCVBUF_ALLOCATOR: ChannelOption<RecvByteBufAllocator> = valueOf(...)
SINGLE_EVENTEXECUTOR_PER_GROUP: ChannelOption<Boolean> = valueOf(...)
SO_BACKLOG: ChannelOption<Integer> = valueOf(...)
SO_BROADCAST: ChannelOption<Boolean> = valueOf(...)
SO_KEEPALIVE: ChannelOption<Boolean> = valueOf(...)
SO_LINGER: ChannelOption<Integer> = valueOf(...)
SO_RCVBUF: ChannelOption<Integer> = valueOf(...)
SO_REUSEADDR: ChannelOption<Boolean> = valueOf(...)
SO_SNDBUF: ChannelOption<Integer> = valueOf(...)
SO_TIMEOUT: ChannelOption<Integer> = valueOf(...)
TCP_NODELAY: ChannelOption<Boolean> = valueOf(...)
WRITE_BUFFER_HIGH_WATER_MARK: ChannelOption<Integer> = valueOf(...)
WRITE_BUFFER_LOW_WATER_MARK: ChannelOption<Integer> = valueOf(...)
WRITE_BUFFER_WATER_MARK: ChannelOption<WriteBufferWaterMark> = valueOf(...)
WRITE_SPIN_COUNT: ChannelOption<Integer> = valueOf(...)

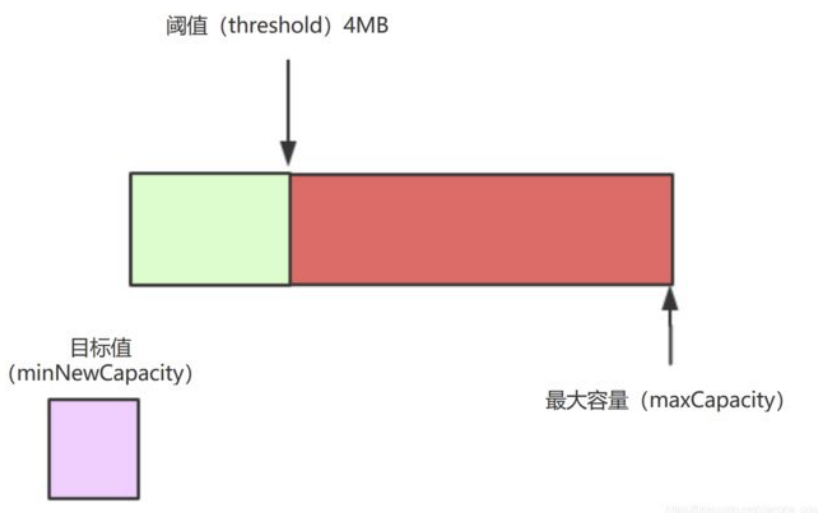
```

并发优化

- volatile的大量、正确使用;
- CAS和原子类的广泛使用;
- 线程安全容器的使用;
- 通过读写锁提升并发性能。

ByteBuf扩容机制

如果我们需要了解ByteBuf的扩容,我们需要先了解ByteBuf中定义的几个成员变量,再从源码的角度来分析扩容。



- minNewCapacity: 表用户需要写入的值大小
- threshold: 阈值, 为Bytebuf内部设定容量的最大值
- maxCapacity: Netty最大能接受的容量大小, 一般为int的最大值

ByteBuf核心扩容方法

进入ByteBuf源码中, 深入分析其扩容方法: idea源码进入: ByteBuf.writeByte()->AbstractByteBuf->calculateNewCapacity

1. 判断目标值与阈值threshold (4MB) 的大小关系, 等于直接返回阈值

```

@Override
public int calculateNewCapacity(int minNewCapacity, int maxCapacity) {
    checkPositiveOrZero(minNewCapacity, name: "minNewCapacity");
    if (minNewCapacity > maxCapacity) {
        throw new IllegalArgumentException(String.format(
            "minNewCapacity: %d (expected: not greater than maxCapacity(%d)",
            minNewCapacity, maxCapacity));
    }

    final int threshold = CALCULATE_THRESHOLD; // 4 MiB page

    if (minNewCapacity == threshold) {
        return threshold;
    }
}

```

等于返回阈值

比最大的容量还大的话，就抛出异常

2. 采用步进4MB的方式完成扩容

```

// If over threshold, do not double but just increase by threshold.
if (minNewCapacity > threshold) {
    int newCapacity = minNewCapacity / threshold * threshold;
    if (newCapacity > maxCapacity - threshold) {
        newCapacity = maxCapacity;
    } else {
        newCapacity += threshold;
    }
    return newCapacity;
}

```

3. 采用64为基数，做倍增的方式完成扩容

```

// Not over threshold. Double up to 4 MiB, starting from 64.
int newCapacity = 64;
while (newCapacity < minNewCapacity) {
    newCapacity <<= 1;
}

```

总结：Netty的ByteBuf需要动态扩容来满足需要，扩容过程：默认门限阈值为4MB(这个阈值是一个经验值，不同场景，可能取值不同)，当需要的容量等于门限阈值，使用阈值作为新的缓存区容量 目标容量，如果大于阈值，采用每次步进4MB的方式进行内存扩张 ((需要扩容值/4MB)*4MB)，扩张后需要和最大内存 (maxCapacity) 进行比较，大于maxCapacity的话就用 maxCapacity, 否则使用扩容值 目标容量，如果小于阈值，采用倍增的方式，以64 (字节) 作为基本数值，每次翻倍增长64 --> 128 --> 256，直到倍增后的结果大于或等于需要的容量值。

补充：handler的生命周期回调接口调用顺序

```

1
2 /**
3  * 在channel的pipeline里如下handler: ch.pipeline().addLast(new LifecycleInBoundHandler());
4  * handler的生命周期回调接口调用顺序:
5  * handlerAdded -> channelRegistered -> channelActive -> channelRead -> channelReadComplete
6  * -> channelInactive -> channelUnRegistered -> handlerRemoved
7  *
8  * handlerAdded: 新建立的连接会按照初始化策略，把handler添加到该channel的pipeline里面，也就是channel.pipeline.addLast(new LifecycleInBoundHandler)执行完成后的回调;
9  * channelRegistered: 当该连接分配到具体的worker线程后，该回调会被调用。
10 * channelActive: channel的准备工作已经完成，所有的pipeline添加完成，并分配到具体的线上上，说明该channel准备就绪，可以使用了。
11 * channelRead: 客户端向服务端发来数据，每次都会回调此方法，表示有数据可读;
12 * channelReadComplete: 服务端每次读完一次完整的数据之后，回调该方法，表示数据读取完毕;
13 * channelInactive: 当连接断开时，该回调会被调用，说明这时候底层的TCP连接已经被断开了。
14 * channelUnRegistered: 对应channelRegistered，当连接关闭后，释放绑定的workder线程;
15 * handlerRemoved: 对应handlerAdded，将handler从该channel的pipeline移除后的回调方法。
16 */
17 public class LifecycleInBoundHandler extends ChannelInboundHandlerAdapter {
18     @Override
19     public void channelRegistered(ChannelHandlerContext ctx)

```

```

20 throws Exception {
21     System.out.println("channelRegistered: channel注册到NioEventLoop");
22     super.channelRegistered(ctx);
23 }
24
25 @Override
26 public void channelUnregistered(ChannelHandlerContext ctx)
27 throws Exception {
28     System.out.println("channelUnregistered: channel取消和NioEventLoop的绑定");
29     super.channelUnregistered(ctx);
30 }
31
32 @Override
33 public void channelActive(ChannelHandlerContext ctx)
34 throws Exception {
35     System.out.println("channelActive: channel准备就绪");
36     super.channelActive(ctx);
37 }
38
39 @Override
40 public void channelInactive(ChannelHandlerContext ctx)
41 throws Exception {
42     System.out.println("channelInactive: channel被关闭");
43     super.channelInactive(ctx);
44 }
45
46 @Override
47 public void channelRead(ChannelHandlerContext ctx, Object msg)
48 throws Exception {
49     System.out.println("channelRead: channel中有可读的数据" );
50     super.channelRead(ctx, msg);
51 }
52
53 @Override
54 public void channelReadComplete(ChannelHandlerContext ctx)
55 throws Exception {
56     System.out.println("channelReadComplete: channel读数据完成");
57     super.channelReadComplete(ctx);
58 }
59
60 @Override
61 public void handlerAdded(ChannelHandlerContext ctx)
62 throws Exception {
63     System.out.println("handlerAdded: handler被添加到channel的pipeline");
64     super.handlerAdded(ctx);
65 }
66
67 @Override
68 public void handlerRemoved(ChannelHandlerContext ctx)
69 throws Exception {
70     System.out.println("handlerRemoved: handler从channel的pipeline中移除");
71     super.handlerRemoved(ctx);
72 }
73 }

```

1 文档: [04-Netty核心源码剖析.note](#)

2 链接: <http://note.youdao.com/noteshare?id=ab45dc97644411c44fbd27ee95d8244e&sub=AD6E37D6A0A242E2B43BAAA8952102CA>