Netty

[1. 内存管理 1](#_102)

[1.1. 零拷贝 1](#_101)

[1.2. 规格 1](#_103)

[1.3. 内存分配管理器 1](#_126)

[1.3.1. Arena 1](#_131)

[1.3.2. PoolThreadCache 2](#_136)

[1.4. 申请内存流程 3](#_130)

[2. NioEventLoop 3](#_104)

[2.1. 多生产者单消费者 3](#_105)

[2.2. 选择器 3](#_106)

[2.2.1. 普通：按顺序分配 3](#_107)

[2.2.2. 优化：加1 &（NioEventLoop个数-1） 3](#_108)

[2.3. 运行 3](#_109)

[2.3.1. 监测是否有IO事件 3](#_110)

[2.3.2. 处理IO事件 4](#_111)

[2.3.3. 处理异步任务队列，taskQueue，外部线程提交的任务 4](#_112)

[3. 新连接接入 4](#_119)

[3.1. 检测新连接 4](#_120)

[3.2. 创建NioSocketChannel 4](#_121)

[3.3. 分配线程并注册Selector 4](#_122)

[3.4. 向Selector注册读事件 5](#_123)

[4. FastThreadLocal 5](#_124)

[5. Pipeline 5](#_125)

[6. 对象池Recycler 5](#_127)

[7. 问题 5](#_128)

[7.1. 需要多个Acceptor线程 5](#_129)

[7.2. Netty中的线程池大小 6](#_138)

# 内存管理

## 零拷贝

与操作系统的零拷贝不太一样，使用堆外内存

## 规格

tiny：0~512B，间隔为16B

small：512B~8KB，间隔为512B

page/subpage：8K

chunk：16M

## 内存分配管理器

### Arena

是线程共有的。

通过伙伴算法管理Chunk，以平衡二叉树的形式管理page。

#### page分配

对于page的分配：

当我们要申请大小为 16K 的内存时，我们会从根节点开始不断寻找可用的节点，一直到第 10 层。那么如何判断一个节点是否可用呢？Netty 会在每个节点内部保存一个值，这个值代表这个节点之下的第几层还存在未分配的节点。比如第 9 层的节点的值如果为 9，就代表这个节点本身到下面所有的子节点都未分配；如果第 9 层的节点的值为 10，代表它本身不可被分配，但第 10 层有子节点可以被分配；如果第 9 层的节点的值为 12，此时可分配节点的深度大于了总深度，代表这个节点及其下面的所有子节点都不可被分配。

#### subpage分配

对于subpage的分配：

首先会从arena中取出一个Subpage池的头节点，然后从Chunk中找到一个可分配的Page，Netty会先检查Page是否已被转化成Subpage，如果没有的话，会先创建Subpage，然后再从Subpage分配内存。创建好的Subpage被分成多个element，我们只会用到其中的一段，剩下都是未分配的。而这些还有element未分配的Subpage会被放入到Arena的Subpage池中。Arena中按不同内存规格大小分成了两组Subpage池，分别是tinySubpagePools和smallSubpagePools。还未被完全分配的Subpage会被放入Subpage池中，这样下次用户申请相同规格的内存时就可以直接从Subpage池中找到Subpage进行分配。被分配完的 Subpage会被移除。

#### chunk管理

随着 Chunk 中内存的不断分配和释放（内存释放后会先进入 Cache，如果 Cache 满了会归还给 Chunk），会导致很多碎片内存段，大大增加了之后分配一段连续内存的失败率，针对这种情况，Netty 按内存使用率把 Chunk 分成了几个等级，分配时按一定的次序进行分配，减少内存的碎片化。

而当我们申请内存时，会以 q050->q025->q000->qInit->q075 的次序申请

### PoolThreadCache

#### 缓存内存

Arena 中分配的内存使用完毕以后，不会立刻归还，而是会在 PoolThreadCache 中缓存起来。当用户下一次申请的时候，会先在 PoolThreadCache 中寻找有没有匹配的内存。

PoolThreadCache 是线程私有的，内部通过 MemoryRegionCache 的方式缓存内存。MemoryRegionCache 的内部实际上是一个队列。和 Arena 中的 Subpage 池一样，PoolThreadCache 也把内存按不同的规格保存

￼

#### 释放内存

内存池不会预置内存块到线程缓存中，在线程申请到内存使用完成之后归还内存时优先把内存块缓存到线程中，除非该内存块不适合缓存在线程中（内存太大），假如当前线程内存分配动作非常活跃时，这样会明显的提高分配效率，但是当它不活跃时对内存又是极大的浪费，所以内存池会监控该线程，随时做好把内存从线程缓存中删除的准备。删除缓存的逻辑在 trim() 方法中，这个方法在 allocate() 中被调用，默认每进行 8192 次 allocate() 会调用一次 trim()

## 申请内存流程

初次申请内存时，Netty会从一整块内存（Chunk）中分出一部分给用户使用。这部分工作是由Arena完成的。当用户使用完释放完毕内存，分出来的内存会按照不同规格大小放在PoolThreadCache中缓存起来。当下次要申请内存的时候，优先从PoolThreadCache中找。

具体申请过程如下：

1. 对请求内存做规整化处理

2. 当请求内存小于8K时，走subpage逻辑，先从PoolThreadCache中找，再从Subpage池中找，最后向Arena申请分配

3. 当请求内存大于8k小于16M时，走Chunk逻辑，先从PoolThreadCache中找，再向Arena申请分配

4. 如果请求内存大于16M时，不走内存池，直接分配

# NioEventLoop

如果调用Channel操作的线程是当前线程，那么直接执行；否则会将操作封装成任务放入NioEventLoop的任务队列中。

## 多生产者单消费者

## 选择器

### 普通：按顺序分配

### 优化：加1 &（NioEventLoop个数-1）

## 运行

### 监测是否有IO事件

判断轮询过程是否有任务加入，是的话终止本次轮询

不需要的话，进行一次定时轮询任务

解决JDK空轮询bug：当检测到多次空轮询时，会创建一个新的Selector。

### 处理IO事件

取出IO事件对应的channel。

处理channel，如果用户移除了一定数量的Channel之后，会丢掉原有的key，重新select()一批出来

#### 未优化的SelectedKeys

#### 优化后的SelectedKeys

使用自定义Set（实质是数组）替换Selector内部的SelectedKeys和publicSelectedKeys（两个HashSet）

### 处理异步任务队列，taskQueue，外部线程提交的任务

#### 两个异步队列

定时任务队列和普通任务队列。

#### 合并定时任务到普通任务队列

#### 处理任务：每执行64个任务就会检查一下是否超时

# 新连接接入

## 检测新连接

maxMessagePerRead控制了每次调用read()方法接收的新连接数量；如果超过，会停止接收新连接，一般设置为16.

还可以进行收尾工作，可通过可通过SingleThreadEventLoop中的executeAfterEventLoopIterator（Runnable task)添加收尾任务

## 创建NioSocketChannel

## 分配线程并注册Selector

由服务器启动时注册的ServerBootstrapAcceptor处理，创建相应的worker channel，并注册到Selector中去。

## 向Selector注册读事件

# FastThreadLocal

# Pipeline

1. 可通过@Sharable共享handler。

2. 可通过@Skip跳过某些方法，ChannelInboundHandlerAdapter的每个方法都加了@Skip，也就是说只有用户重写的方法才是关心的。

3. 可动态插拔handler。

inbound事件和onbound事件的区别：

inbound事件主要发生在数据被接收或Channel状态变化时，被动接收。

outbound事件主要发生在数据发送或用户对Channel进行操作时，是主动发起的。

# 对象池Recycler

1. Stack作为本线程对象池的核心

2. 在回收线程1的对象时，线程1会放到Stack中；其他线程会放到WeakOrderQueue，并通过链表形式与线程1的Stack相连。

3. WeakOrderQueue以Link管理对象，每个Link可存放16个对象。

4. 默认情况下一个线程最多持有2CPU个WeakOrderQueue。也就是说一个线程最多可以帮2CPU个外部线程回收对象。

5. 当前线程从对象池拿对象时，先从Stack中拿，然后从别的线程的WeakOrderQueue中转移对象到Stack中。如果存在，则从Stack获取对象；如果没有回收到就创建对象。

6. 一个对象池中最多存放4K个对象。

# 问题

## 需要多个Acceptor线程

在个别特殊场景中，一个NIO线程负责监听和处理所有的客户端连接可能会存在性能问题，例如并发百万客户端连接、或者服务端需要对客户端握手进行安全认证。但是认证本身非常损耗性能，在这类场景下，单独一个Acceptor线程可能会存在性能不足的问题，为了解决性能问题，即主从React多线程模型（多个Acceptor线程）。

## Netty中的线程池大小

其实是根据公式得到的：（io等待时间/CPU时间 + 1）\* Ncpu

Netty默认ioRatio=50，即io等待时间 = CPU时间，所以等于2Ncpu