# 一、Reference counted objects

来自Netty的官方文档：

<http://netty.io/wiki/reference-counted-objects.html>

翻译:

<http://damacheng009.iteye.com/blog/2013657>

自Netty4开始，一些对象的生命周期通过引用计数器进行管理，而不通过GC来进行管理（对象不可达来判定是否回收）。

这样Netty不使用这些对象的时候能快速把它们返回到对象池中。

引用计数器通过稍微不方便的语法来实现。通过语法实现，说明Netty把对象回收交给了程序员来主动释放，像C中的free一样。ByteBuf就是这样的对象。

## Basics of reference counting

ByteBuf buf = ctx.alloc().directBuffer(); //在刚一分配的时候，引用数会设置为1

assert buf.refCnt() == 1;

// release() returns true only if the reference count becomes 0.

boolean destroyed = buf.release();//并不是释放内存，而是更改引用计数器的值；如果为0，则deallocate()

assert destroyed;

assert buf.refCnt() == 0;

### Dangling reference

在使用ByteBuf的时候，都需要检查引用计数器的值，如果为0，则抛出异常。



### Increasing the reference count

buf.retain();//将引用计数器的值加1

### Derived buffers

ByteBuf.duplicate(), ByteBuf.slice() ，ByteBuf.order(ByteOrder) 的返回值是原来buffer上衍生出来的逻辑buffer，shares the memory region of the parent buffer。

引用计数和不变，使用parent buffer的引用计数器。因此在使用他们的时候，如果把他们作为参数传递给程序的组件，要手动把计数器加1.

 ByteBuf.copy() and ByteBuf.readBytes(int) 这样的api并不是子缓冲，而是生成一个新的对象。

## Reference-counting in ChannelHandler

### Inbound messages

1 使用Netty时，在事件循环中，一定要记得在finally中 释放Bytebuf，或者把它发送到下一个时间处理器。

2 ByteBuf并不是Netty中唯一的引用计数器对象，由解码器（decoder）生成的消息（messages）对象，这些对象很可能也是引用计数对象。

3 可以调用 ReferenceCountUtil.release(msg) api来释放。

4 SimpleChannelHandler 在接收消息的地方都调用了释放api， 可以继承它来简化代码。

### Outbound messages

一般Netty在消息发送之后，会自动释放。但如果有了组件传递，就需要手动释放了。

## Troubleshooting buffer leaks

# 二、ReferenceCounted interface

what:

主要的api是 增加引用计数器、减少引用计数器、获取引用计数器的值。

how:

retain()、retain(int)、release()、release(int) 、refCnt():int

why:

垃圾回收

# 三、ByteBUf

what:

1 继承了ReferenceCounted接口，说明其生命周期的管理由程序员来实现。

2 分为池化的和非池化的两种类型：

UnpooledHeapByteBuf、UnpooledDirectByteBuf ：能够依靠jvm来内存回收，也能主动回收。

PooledHeapByteBuf、PooledDirectByteBuf 只能主动回收，否则会OOM。

# 四、ChannelHandler组件

# SimpleChannelInboundHandler组件

what： SimpleChannelInboundHandler<T>是个泛型抽象类。如果要实现我们的hander，可以继承这个类。

How：

1 如何使用？

需要重写的方法：channelRead0（）

2 与上下环节的联系:

channelRead()函数。可以看到，它自己释放 引用计数器对象。

# 五、Channel

1 是socket的封装，每个Channel对象中都会有一个ChannelPipeline

（AbstractChannel、DefaultChannelPipeline）

# 六、ChannelPipeline

1 主要存储HandlerContext的链表。

2 链表的首节点是： AbstractChannelHandlerContext head，既是一个OutboundHandler，同时也是一个HandlerContext

链表的尾节点是： AbstractChannelHandlerContext tail，即是一个InboundHandler，同时也是一个HandlerContext

# 七、ServerBootstrap

## 1 bind 流程

a 执行initAndRegister： 在这里面获得Channel，并对Channel做init操作。



b ServerBoostrap init()

b ChannelInitializer 也是ChannelInboundHandlerAdapter

## 2 设置tcp参数

tcp的参数也可以在socket中设置。

如： accept队列的大小； 心跳监控；是否允许延迟发送

<http://moonmonster.iteye.com/blog/2278937>

<http://11838641.blog.51cto.com/11828641/1864664>

<http://www.jianshu.com/p/e6f2036621f4>



# 八、自己的项目

1 服务端 是如何阻塞的

2 handler 中out，直接使用ProtobufEncoder。Netty 是如何自动实现的。

3 自定义解码继承自ByteToMessageDecoder， 是不是还有其他类继承，有什么区别。

# 九、服务端代码解析

1 ·

// Bind and start to accept incoming connections.

ChannelFuture f = b.bind(port).sync(); // (7)

// Wait until the server socket is closed.

// In this example, this does not happen, but you can do that to gracefully

// shut down your server.

f.channel().closeFuture().sync();