Netty的4个重要内容

1.Reactor线程模型：高性能多线程设计思路

2.Netty中自己定义的channel概念：增强版的NIOchannel

3.ChannelPipeline责任链设计模式：事件处理机制

4.内存管理：增强型byteBuf缓冲区

Netty整体结构图

在这里插入图片描述

这张图来自官网，可以看出三大模块：

1.支持socket等多种传输方式

2.提供了多种协议编码实现；

3.核心设计包含：事件处理模型、API的使用、byteBuffer的增强

官网地址：netty.io

本章讲解最简单的echo实例

Netty的线程模型

在这里插入图片描述

Netty有一个标准的模板工具类ServerBootstrap只需要按照要求配置group即可。server端使用eventloopgroup（事件驱动器）来分发处理事件，handler就是指定谁来处理请求的，childHandler就是处理客户端连接过来的请求的处理器。channel是用来创建具体通道的实例。optoin是一些配置。

public final class EchoServer {

static final int PORT = Integer.parseInt(System.getProperty("port", "8080"));

public static void main(String[] args) throws Exception {

// Configure the server.

// 创建EventLoopGroup accept线程组 NioEventLoop

EventLoopGroup bossGroup = new NioEventLoopGroup(1);

// 创建EventLoopGroup I/O线程组

EventLoopGroup workerGroup2 = new NioEventLoopGroup(1);

try {

// 服务端启动引导工具类

ServerBootstrap b = new ServerBootstrap();

// 配置服务端处理的reactor线程组以及服务端的其他配置

b.group(bossGroup, workerGroup2).channel(NioServerSocketChannel.class).option(ChannelOption.SO\_BACKLOG, 100)

.handler(new LoggingHandler(LogLevel.DEBUG)).childHandler(new ChannelInitializer<SocketChannel>() {

@Override

public void initChannel(SocketChannel ch) throws Exception {

ChannelPipeline p = ch.pipeline();

p.addLast(new EchoServerHandler());

}

});

// 通过bind启动服务

ChannelFuture f = b.bind(PORT).sync();

// 阻塞主线程，知道网络服务被关闭

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

} finally {

// 关闭线程组

bossGroup.shutdownGracefully();

workerGroup2.shutdownGracefully();

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

EventLoopGroup

初始化过程

在这里插入图片描述

Netty启动的时候会构建多个EventLoopGroup。构建以批次为单位，一次构建多组，分别处理accept、io不同事件。

在这里插入图片描述

main对应的accept；sub对应的是IO；

EventLoopGroup实现了EventExcutor接口，通过上层父类MultithreadEventExcutorGroup的构造方法创建事件执行器，使用其中的excutor来创建线程并且执行。Excutor线程池本质上就是创建多个NioEventLoop时间执行器。NioEventLoop的实现就是NIO中的selector增强，本质还是selector，而NioEventLoop是不能创建线程的，这里用到了Excutor来创建线程。在线程run方法中会轮训执行selector.select方法和taskqueue里的内容，每个EventLoop对应一个轮询线程。eventLoop是不会自己执行线程的，只有当有任务提交时才触发eventloop，如果eventloop想要自己执行任务，那么就需要使用execute方法来提交一个runnable

protected MultithreadEventExecutorGroup(int nThreads, Executor executor,

EventExecutorChooserFactory chooserFactory, Object... args) {

if (nThreads <= 0) {

throw new IllegalArgumentException(String.format("nThreads: %d (expected: > 0)", nThreads));

}

if (executor == null) {// 如果执行器为空，则创建一个

executor = new ThreadPerTaskExecutor(newDefaultThreadFactory());

}

children = new EventExecutor[nThreads];

for (int i = 0; i < nThreads; i ++) {

boolean success = false;

try {

children[i] = newChild(executor, args);

success = true;

} catch (Exception e) {

// TODO: Think about if this is a good exception type

throw new IllegalStateException("failed to create a child event loop", e);

} finally {

if (!success) {

for (int j = 0; j < i; j ++) {

children[j].shutdownGracefully();

}

for (int j = 0; j < i; j ++) {

EventExecutor e = children[j];

try {

while (!e.isTerminated()) {

e.awaitTermination(Integer.MAX\_VALUE, TimeUnit.SECONDS);

}

} catch (InterruptedException interrupted) {

// Let the caller handle the interruption.

Thread.currentThread().interrupt();

break;

}

}

}

}

}

chooser = chooserFactory.newChooser(children);

final FutureListener<Object> terminationListener = new FutureListener<Object>() {

@Override

public void operationComplete(Future<Object> future) throws Exception {

if (terminatedChildren.incrementAndGet() == children.length) {

terminationFuture.setSuccess(null);

}

}

};

for (EventExecutor e: children) {

e.terminationFuture().addListener(terminationListener);

}

Set<EventExecutor> childrenSet = new LinkedHashSet<EventExecutor>(children.length);

Collections.addAll(childrenSet, children);

readonlyChildren = Collections.unmodifiableSet(childrenSet);

}

//eventloop创建多个线程

@Override

protected EventLoop newChild(Executor executor, Object... args) throws Exception {

return new NioEventLoop(this, executor, (SelectorProvider) args[0],

((SelectStrategyFactory) args[1]).newSelectStrategy(), (RejectedExecutionHandler) args[2]);

}

//executor

@Override

public void execute(Runnable task) {

if (task == null) {

throw new NullPointerException("task");

}

// 判断execute方法的调用者是不是EventLoop同一个线程

boolean inEventLoop = inEventLoop();

addTask(task);// 增加到任务队列

if (!inEventLoop) {// 不是同一个线程，则调用启动方法

startThread();//开启eventloop执行

if (isShutdown()) {

boolean reject = false;

try {

if (removeTask(task)) {

reject = true;

}

} catch (UnsupportedOperationException e) {

// The task queue does not support removal so the best thing we can do is to just move on and

// hope we will be able to pick-up the task before its completely terminated.

// In worst case we will log on termination.

}

if (reject) {

reject();

}

}

}

if (!addTaskWakesUp && wakesUpForTask(task)) {

wakeup(inEventLoop);

}

}

private void startThread() {

if (state == ST\_NOT\_STARTED) {

if (STATE\_UPDATER.compareAndSet(this, ST\_NOT\_STARTED, ST\_STARTED)) {

try {

doStartThread();// 未启动，则触发启动

} catch (Throwable cause) {

STATE\_UPDATER.set(this, ST\_NOT\_STARTED);

PlatformDependent.throwException(cause);

}

}

}

}

private void doStartThread() {

assert thread == null;

executor.execute(new Runnable() {// 这里的executor是初始化EventLoop的时候传进来的

@Override

public void run() {

thread = Thread.currentThread();

if (interrupted) {

thread.interrupt();

}

boolean success = false;

updateLastExecutionTime();

try {// 创建线程开始执行run方法，所以，每个EventLoop都是执行run

SingleThreadEventExecutor.this.run();

success = true;

} catch (Throwable t) {

logger.warn("Unexpected exception from an event executor: ", t);

} finally {

for (;;) {

int oldState = state;

if (oldState >= ST\_SHUTTING\_DOWN || STATE\_UPDATER.compareAndSet(

SingleThreadEventExecutor.this, oldState, ST\_SHUTTING\_DOWN)) {

break;

}

}

// Check if confirmShutdown() was called at the end of the loop.

if (success && gracefulShutdownStartTime == 0) {

if (logger.isErrorEnabled()) {

logger.error("Buggy " + EventExecutor.class.getSimpleName() + " implementation; " +

SingleThreadEventExecutor.class.getSimpleName() + ".confirmShutdown() must " +

"be called before run() implementation terminates.");

}

}

try {

// Run all remaining tasks and shutdown hooks.

for (;;) {

if (confirmShutdown()) {

break;

}

}

} finally {

try {

cleanup();

} finally {

STATE\_UPDATER.set(SingleThreadEventExecutor.this, ST\_TERMINATED);

threadLock.release();

if (!taskQueue.isEmpty()) {

if (logger.isWarnEnabled()) {

logger.warn("An event executor terminated with " +

"non-empty task queue (" + taskQueue.size() + ')');

}

}

terminationFuture.setSuccess(null);

}

}

}

}

});

}

//SingleThreadEventExecutor.this.run();线程内的执行

@Override

protected void run() {// 有任务提交后，被触发执行

for (;;) {// 执行两件事selector,select的事件 和 taskQueue里面的内容

try {

try {

switch (selectStrategy.calculateStrategy(selectNowSupplier, hasTasks())) {

case SelectStrategy.CONTINUE:

continue;

case SelectStrategy.BUSY\_WAIT:

// fall-through to SELECT since the busy-wait is not supported with NIO

case SelectStrategy.SELECT:

select(wakenUp.getAndSet(false));

// 'wakenUp.compareAndSet(false, true)' is always evaluated

// before calling 'selector.wakeup()' to reduce the wake-up

// overhead. (Selector.wakeup() is an expensive operation.)

//

// However, there is a race condition in this approach.

// The race condition is triggered when 'wakenUp' is set to

// true too early.

//

// 'wakenUp' is set to true too early if:

// 1) Selector is waken up between 'wakenUp.set(false)' and

// 'selector.select(...)'. (BAD)

// 2) Selector is waken up between 'selector.select(...)' and

// 'if (wakenUp.get()) { ... }'. (OK)

//

// In the first case, 'wakenUp' is set to true and the

// following 'selector.select(...)' will wake up immediately.

// Until 'wakenUp' is set to false again in the next round,

// 'wakenUp.compareAndSet(false, true)' will fail, and therefore

// any attempt to wake up the Selector will fail, too, causing

// the following 'selector.select(...)' call to block

// unnecessarily.

//

// To fix this problem, we wake up the selector again if wakenUp

// is true immediately after selector.select(...).

// It is inefficient in that it wakes up the selector for both

// the first case (BAD - wake-up required) and the second case

// (OK - no wake-up required).

if (wakenUp.get()) {

selector.wakeup();

}

// fall through

default:

}

} catch (IOException e) {

// If we receive an IOException here its because the Selector is messed up. Let's rebuild

// the selector and retry. https://github.com/netty/netty/issues/8566

rebuildSelector0();

handleLoopException(e);

continue;

}

cancelledKeys = 0;

needsToSelectAgain = false;

final int ioRatio = this.ioRatio;

if (ioRatio == 100) {

try {// 处理事件

processSelectedKeys();

} finally {

// Ensure we always run tasks.

runAllTasks();

}

} else {

final long ioStartTime = System.nanoTime();

try {

processSelectedKeys();

} finally {

// Ensure we always run tasks.

final long ioTime = System.nanoTime() - ioStartTime;

runAllTasks(ioTime \* (100 - ioRatio) / ioRatio);

}

}

} catch (Throwable t) {

handleLoopException(t);

}

// Always handle shutdown even if the loop processing threw an exception.

try {

if (isShuttingDown()) {

closeAll();

if (confirmShutdown()) {

return;

}

}

} catch (Throwable t) {

handleLoopException(t);

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274

275

EventLoop启动图

在这里插入图片描述

bind绑定端口过程图

在这里插入图片描述

服务端代码需要调用bind进行绑定（ChannelFuture f = b.bind(PORT).sync();这一行），bind进入之后是dobind

private ChannelFuture doBind(final SocketAddress localAddress) {

final ChannelFuture regFuture = initAndRegister();// 创建/初始化ServerSocketChannel对象，并注册到Selector

final Channel channel = regFuture.channel();

if (regFuture.cause() != null) {

return regFuture;

}

// 等注册完成之后，再绑定端口。 防止端口开放了，却不能处理请求

if (regFuture.isDone()) {

// At this point we know that the registration was complete and successful.

ChannelPromise promise = channel.newPromise();

doBind0(regFuture, channel, localAddress, promise);// 实际操作绑定端口的代码

return promise;

} else {

// Registration future is almost always fulfilled already, but just in case it's not.

final PendingRegistrationPromise promise = new PendingRegistrationPromise(channel);

regFuture.addListener(new ChannelFutureListener() {

@Override

public void operationComplete(ChannelFuture future) throws Exception {

Throwable cause = future.cause();

if (cause != null) {

// Registration on the EventLoop failed so fail the ChannelPromise directly to not cause an

// IllegalStateException once we try to access the EventLoop of the Channel.

promise.setFailure(cause);

} else {

// Registration was successful, so set the correct executor to use.

// See https://github.com/netty/netty/issues/2586

promise.registered();

doBind0(regFuture, channel, localAddress, promise);

}

}

});

return promise;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

其中initregist初始化,创建了一个netty的通道

final ChannelFuture initAndRegister() {

Channel channel = null;

try {

channel = channelFactory.newChannel();

init(channel);

} catch (Throwable t) {

if (channel != null) {

// channel can be null if newChannel crashed (eg SocketException("too many open files"))

channel.unsafe().closeForcibly();

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(channel, GlobalEventExecutor.INSTANCE).setFailure(t);

}

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(new FailedChannel(), GlobalEventExecutor.INSTANCE).setFailure(t);

}

// （一开始初始化的group）MultithreadEventLoopGroup里面选择一个eventLoop进行绑定

ChannelFuture regFuture = config().group().register(channel);

if (regFuture.cause() != null) {

if (channel.isRegistered()) {

channel.close();

} else {

channel.unsafe().closeForcibly();

}

}

// If we are here and the promise is not failed, it's one of the following cases:

// 1) If we attempted registration from the event loop, the registration has been completed at this point.

// i.e. It's safe to attempt bind() or connect() now because the channel has been registered.

// 2) If we attempted registration from the other thread, the registration request has been successfully

// added to the event loop's task queue for later execution.

// i.e. It's safe to attempt bind() or connect() now:

// because bind() or connect() will be executed \*after\* the scheduled registration task is executed

// because register(), bind(), and connect() are all bound to the same thread.

return regFuture;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

而注册的本质是NIO中把channel与selector进行绑定，selector是在之前的eventloop中，所以ChannelFuture regFuture = config().group().register(channel);才会从group（eventloopgroup）对象中注册channel。

register是由父类实现的

@Override

public ChannelFuture register(Channel channel) {

return next().register(channel);// 根据选择器，选择一个合适的NioEventLoop进行注册(SingleThreadEventLoop)

}

1

2

3

4

next是轮训找到一个合适的，register方法还是通过父类实现

@Override

public ChannelFuture register(Channel channel) {

return register(new DefaultChannelPromise(channel, this));// 执行注册相关的逻辑

}

@Override

public ChannelFuture register(final ChannelPromise promise) {

ObjectUtil.checkNotNull(promise, "promise");

promise.channel().unsafe().register(this, promise);// 调用channel中unsafe对象的注册方法(AbstractUnsafe)

return promise;

}

1

2

3

4

5

6

7

8

9

10

11

这里的register方法还要继续向上层找

@Override

public final void register(EventLoop eventLoop, final ChannelPromise promise) {

if (eventLoop == null) {

throw new NullPointerException("eventLoop");

}

if (isRegistered()) {

promise.setFailure(new IllegalStateException("registered to an event loop already"));

return;

}

if (!isCompatible(eventLoop)) {

promise.setFailure(

new IllegalStateException("incompatible event loop type: " + eventLoop.getClass().getName()));

return;

}

AbstractChannel.this.eventLoop = eventLoop;

// 如果调用register方法的线程和EventLoop执行线程不是同一个线程，则以任务形式提交绑定操作

if (eventLoop.inEventLoop()) {

register0(promise);// 实际就是调用这个方法

} else {

try {

eventLoop.execute(new Runnable() {

@Override

public void run() {

register0(promise);

}

});

} catch (Throwable t) {

logger.warn(

"Force-closing a channel whose registration task was not accepted by an event loop: {}",

AbstractChannel.this, t);

closeForcibly();

closeFuture.setClosed();

safeSetFailure(promise, t);

}

}

}

private void register0(ChannelPromise promise) {

try {

// check if the channel is still open as it could be closed in the mean time when the register

// call was outside of the eventLoop

if (!promise.setUncancellable() || !ensureOpen(promise)) {

return;

}

boolean firstRegistration = neverRegistered;

doRegister();// NIOchannel中，将Channel和NioEventLoop里面的Selector进行绑定

neverRegistered = false;

registered = true;

// Ensure we call handlerAdded(...) before we actually notify the promise. This is needed as the

// user may already fire events through the pipeline in the ChannelFutureListener.

pipeline.invokeHandlerAddedIfNeeded();

safeSetSuccess(promise);

pipeline.fireChannelRegistered();// 传播通道完成注册的事件

// Only fire a channelActive if the channel has never been registered. This prevents firing

// multiple channel actives if the channel is deregistered and re-registered.

if (isActive()) {// ServerSocketChannel服务端完成bind之后，才会变成active。

if (firstRegistration) {// 如果是socketChannel，active的判断就是是否连接、是否开启

pipeline.fireChannelActive();

} else if (config().isAutoRead()) {

// This channel was registered before and autoRead() is set. This means we need to begin read

// again so that we process inbound data.

//

// See https://github.com/netty/netty/issues/4805

beginRead();// 如果是取消register，再重新绑定的，就会直接注册到OP\_READ

}

}

} catch (Throwable t) {

// Close the channel directly to avoid FD leak.

closeForcibly();

closeFuture.setClosed();

safeSetFailure(promise, t);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

从这段代码可以看出register有两个入参，一个是eventloop一个是channel，就是这个方法调用doregister（）把eventloop与channel进行了注册

@Override

protected void doRegister() throws Exception {

boolean selected = false;

for (;;) {

try {

selectionKey = javaChannel().register(eventLoop().unwrappedSelector(), 0, this);

return;

} catch (CancelledKeyException e) {

if (!selected) {

// Force the Selector to select now as the "canceled" SelectionKey may still be

// cached and not removed because no Select.select(..) operation was called yet.

eventLoop().selectNow();

selected = true;

} else {

// We forced a select operation on the selector before but the SelectionKey is still cached

// for whatever reason. JDK bug ?

throw e;

}

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

到这里就是完整的channel初始化到注册的过程。

回到bind过程代码,在代码中dobind实际是调用了dobind0

private static void doBind0(

final ChannelFuture regFuture, final Channel channel,

final SocketAddress localAddress, final ChannelPromise promise) {

// This method is invoked before channelRegistered() is triggered. Give user handlers a chance to set up

// the pipeline in its channelRegistered() implementation.

channel.eventLoop().execute(new Runnable() {

@Override

public void run() {// 这里向EventLoop提交任务，一旦有任务提交则会触发EventLoop的轮询

if (regFuture.isSuccess()) {// 本质又绕回到channel的bind方法上面。

channel.bind(localAddress, promise).addListener(ChannelFutureListener.CLOSE\_ON\_FAILURE);

} else {

promise.setFailure(regFuture.cause());

}

}

});

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

这段代码就是channel的eventloop提交任务。当eventloop轮询检测到有任务时就会提交任务，如果当前channel有任务正在执行，那么就把新任务加入等待队列taskqueue中，等待执行。当初始化成功后，这里的bind方法会给当前的端口绑定一个职责连handlers

@Override

public ChannelFuture bind(SocketAddress localAddress, ChannelPromise promise) {// 核心的绑定实现在这个channel抽象类中

return pipeline.bind(localAddress, promise);// 这里触发一个netty职责链中的bind事件，由应用代码发起到底层，属于outBound

}

1

2

3

4

Channel

pipeline DefualtChannelPipeline 通道内事件处理链路

eventLoop 绑定的eventLoop，用于执行操作

unsafe 提供IO操作的封装

config（） Channelconfig 返回通道配置信息

read（） Channel 开始读，触发读取链路

write（Object o） ChannelFuture 写，触发写链路

bind（socketaddress s） ChannelFuture 绑定

pipeline职责连

为请求创建一个处理对象的链

在这里插入图片描述

实现责任链的4个要素

1.处理器抽象类

2.具体处理实现类

3.保存处理器信息

4.处理执行

集合形势存储处理器

//处理器抽象类

class AbstractHandler {void doHandler(Object arg0)}

//处理器实现类

class Handler1 extends AbstractHandler { assert coutinue;}

class Handler2 extends AbstractHandler { assert coutinue;}

class Handler3 extends AbstractHandler { assert coutinue;}

//创建集合并存储处理器实例

List handlers = new List();

handlers.add(Handler1,Handler2,Handler3)

//处理请求，调用处理器

void Process（request）{

for（handler in handlers）{

handler.doHandler(request);

}

}

//发起请求吊用，通过责任链处理请求

call.process(request);

链表形势存储处理器

//处理器抽象类

class AbstractHandler{

AbstractHandler next；//下一个节点

void doHandler（Object arg0）

}

//处理器实现类

class Handler1 extends AbstractHandler { assert coutinue;}

class Handler2 extends AbstractHandler { assert coutinue;}

class Handler3 extends AbstractHandler { assert coutinue;}

//将处理器串成链表存储

pipeline = head- >{Handler1->Handler2->Handler3}-> end

//处理请求，调用处理器（从头到尾）

void Process(request){

handler = pipeline.findOne;//查找第一个

while(handler!=null){

handler.doHandler(request);

handler = handler.next();

}

}

链式处理器责任链简单demo

public class PipelineDemo {

/\*\*

\* 初始化的时候造一个head，作为责任链的开始，但是并没有具体的处理

\*/

public HandlerChainContext head = new HandlerChainContext(new AbstractHandler() {

@Override

void doHandler(HandlerChainContext handlerChainContext, Object arg0) {

handlerChainContext.runNext(arg0);

}

});

public void requestProcess(Object arg0) {

this.head.handler(arg0);

}

public void addLast(AbstractHandler handler) {

HandlerChainContext context = head;

while (context.next != null) {

context = context.next;

}

context.next = new HandlerChainContext(handler);

}

public static void main(String[] args) {

System.out.println("1111");

PipelineDemo pipelineChainDemo = new PipelineDemo();

pipelineChainDemo.addLast(new Handler2());

pipelineChainDemo.addLast(new Handler1());

pipelineChainDemo.addLast(new Handler1());

pipelineChainDemo.addLast(new Handler2());

// 发起请求

pipelineChainDemo.requestProcess("火车呜呜呜~~");

}

}

/\*\*

\* handler上下文，我主要负责维护链，和链的执行

\*/

class HandlerChainContext {

HandlerChainContext next; // 下一个节点

AbstractHandler handler;

public HandlerChainContext(AbstractHandler handler) {

this.handler = handler;

}

void handler(Object arg0) {

this.handler.doHandler(this, arg0);

}

/\*\*

\* 继续执行下一个

\*/

void runNext(Object arg0) {

if (this.next != null) {

this.next.handler(arg0);

}

}

}

// 处理器抽象类

abstract class AbstractHandler {

/\*\*

\* 处理器，这个处理器就做一件事情，在传入的字符串中增加一个尾巴..

\*/

abstract void doHandler(HandlerChainContext handlerChainContext, Object arg0); // handler方法

}

// 处理器具体实现类

class Handler1 extends AbstractHandler {

@Override

void doHandler(HandlerChainContext handlerChainContext, Object arg0) {

arg0 = arg0.toString() + "..handler1的小尾巴.....";

System.out.println("我是Handler1的实例，我在处理：" + arg0);

// 继续执行下一个

handlerChainContext.runNext(arg0);

}

}

// 处理器具体实现类

class Handler2 extends AbstractHandler {

@Override

void doHandler(HandlerChainContext handlerChainContext, Object arg0) {

arg0 = arg0.toString() + "..handler2的小尾巴.....";

System.out.println("我是Handler2的实例，我在处理：" + arg0);

// 继续执行下一个

handlerChainContext.runNext(arg0);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

责任链开始执行之后，对于是否执行下一个handler，一般是在handler中定义的，用处理器来控制下一个执行哪一个处理器。

责任链模型

出站和入站事件是以socket为标准，从socket向上的是入站，从上向socket的是出站

在这里插入图片描述

在入站事件和出站事件时会触发处理器。

pipeline初始化的实在创建channel时做的，pipeline会规定一个head和一个end，即头尾两个固定处理器，这两个处理器只是固定了首尾上下文，没有任何实际操作。

入站事件：通常指IO线程生成了入站数据。

从socket底层自己往上冒上来的事件都是入站。如：EventLoop收到selector的OP\_READ事件，入站处理器调用socketChannel.read(ByteBuffer)接收到数据后，ChannelPipeline中的下一个节点的channelRead方法将被调用。

出站事件：通常指IO线程执行实际的输出操作。

想主动往socket底层操作的事件都是出站。如：bind方法是将请求server socket绑定到给定的SocketAddress，ChannelPipeline中的下一个节点中的bind方法被调用

pipeline的方法

在这里插入图片描述

handler的三种作用：处理IO（读写）、拦截IO（accept）、传入下一个handler（runnext）

register入站方法源码跟踪，首先从上面代码的bind开始，一直跟到ServerBootstrap类的init方法

@Override

void init(Channel channel) throws Exception {

final Map<ChannelOption<?>, Object> options = options0();

synchronized (options) {

setChannelOptions(channel, options, logger);

}

final Map<AttributeKey<?>, Object> attrs = attrs0();

synchronized (attrs) {

for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {

@SuppressWarnings("unchecked")

AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();

channel.attr(key).set(e.getValue());

}

}

ChannelPipeline p = channel.pipeline();

final EventLoopGroup currentChildGroup = childGroup;

final ChannelHandler currentChildHandler = childHandler;

final Entry<ChannelOption<?>, Object>[] currentChildOptions;

final Entry<AttributeKey<?>, Object>[] currentChildAttrs;

synchronized (childOptions) {

currentChildOptions = childOptions.entrySet().toArray(newOptionArray(0));

}

synchronized (childAttrs) {

currentChildAttrs = childAttrs.entrySet().toArray(newAttrArray(0));

}

// ChannelInitializer是一个特殊的handler，一般就是在registered之后，执行一次，然后销毁。用于初始化channel

p.addLast(new ChannelInitializer<Channel>() {

@Override// 触发ChannelInitializer时，收到注册成功的事件后，就会执行initChannel方法

public void initChannel(final Channel ch) throws Exception {

final ChannelPipeline pipeline = ch.pipeline();

ChannelHandler handler = config.handler();

if (handler != null) {

pipeline.addLast(handler);

}

ch.eventLoop().execute(new Runnable() {

@Override

public void run() {

pipeline.addLast(new ServerBootstrapAcceptor(

ch, currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));

}

});

}

});

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

上述代码走到p.add是添加了一个ChannelInitializer方法的通道

@Sharable

public abstract class ChannelInitializer<C extends Channel> extends ChannelInboundHandlerAdapter {

private static final InternalLogger logger = InternalLoggerFactory.getInstance(ChannelInitializer.class);

// We use a ConcurrentMap as a ChannelInitializer is usually shared between all Channels in a Bootstrap /

// ServerBootstrap. This way we can reduce the memory usage compared to use Attributes.

private final ConcurrentMap<ChannelHandlerContext, Boolean> initMap = PlatformDependent.newConcurrentHashMap();

/\*\*

\* This method will be called once the {@link Channel} was registered. After the method returns this instance

\* will be removed from the {@link ChannelPipeline} of the {@link Channel}.

\*

\* @param ch the {@link Channel} which was registered.

\* @throws Exception is thrown if an error occurs. In that case it will be handled by

\* {@link #exceptionCaught(ChannelHandlerContext, Throwable)} which will by default close

\* the {@link Channel}.

\*/

protected abstract void initChannel(C ch) throws Exception;

@Override

@SuppressWarnings("unchecked")

public final void channelRegistered(ChannelHandlerContext ctx) throws Exception {

// Normally this method will never be called as handlerAdded(...) should call initChannel(...) and remove

// the handler.

if (initChannel(ctx)) {// 收到注册成功的事件，先执行initChannel(ctx)，在执行方法重载的initChannel(ch)

// we called initChannel(...) so we need to call now pipeline.fireChannelRegistered() to ensure we not

// miss an event.

ctx.pipeline().fireChannelRegistered();

} else {

// Called initChannel(...) before which is the expected behavior, so just forward the event.

ctx.fireChannelRegistered();

}

}

/\*\*

\* Handle the {@link Throwable} by logging and closing the {@link Channel}. Sub-classes may override this.

\*/

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {

if (logger.isWarnEnabled()) {

logger.warn("Failed to initialize a channel. Closing: " + ctx.channel(), cause);

}

ctx.close();

}

/\*\*

\* {@inheritDoc} If override this method ensure you call super!

\*/

@Override

public void handlerAdded(ChannelHandlerContext ctx) throws Exception {

if (ctx.channel().isRegistered()) {

// This should always be true with our current DefaultChannelPipeline implementation.

// The good thing about calling initChannel(...) in handlerAdded(...) is that there will be no ordering

// surprises if a ChannelInitializer will add another ChannelInitializer. This is as all handlers

// will be added in the expected order.

initChannel(ctx);

}

}

@SuppressWarnings("unchecked")

private boolean initChannel(ChannelHandlerContext ctx) throws Exception {

if (initMap.putIfAbsent(ctx, Boolean.TRUE) == null) { // Guard against re-entrance.

try {// 这个init方法一般就是创建channel时，实现的那个initchannel方法

initChannel((C) ctx.channel());

} catch (Throwable cause) {

// Explicitly call exceptionCaught(...) as we removed the handler before calling initChannel(...).

// We do so to prevent multiple calls to initChannel(...).

exceptionCaught(ctx, cause);

} finally {// ChannelInitializer执行结束之后，会把自己从pipeline中删除掉，避免重复初始化

remove(ctx);

}

return true;

}

return false;

}

private void remove(ChannelHandlerContext ctx) {

try {

ChannelPipeline pipeline = ctx.pipeline();

if (pipeline.context(this) != null) {

pipeline.remove(this);

}

} finally {

initMap.remove(ctx);

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

这个init方法就是创建channel的同时实现initchannel，而这个initchannel的处理器是一个特殊的处理器，因为是初始化，所以只能执行一次，所以当执行完成后需要最后remove掉，这就是pipeline动态的添加和删除实现。

在init方法的匿名函数中,会定义一个配置中的handler和一个pipeline，然后将handler加入pipeline。而eventLoop执行的线程中也增加了一个handler（new ServerBootstrapAcceptor（…））从名字就看到是处理accept连接事件的。

@Override// 触发ChannelInitializer时，收到注册成功的事件后，就会执行initChannel方法

public void initChannel(final Channel ch) throws Exception {

final ChannelPipeline pipeline = ch.pipeline();

ChannelHandler handler = config.handler();

if (handler != null) {

pipeline.addLast(handler);

}

ch.eventLoop().execute(new Runnable() {

@Override

public void run() {

pipeline.addLast(new ServerBootstrapAcceptor(

ch, currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));

}

});

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

registered入站事件处理图

在这里插入图片描述

bind出站事件处理图

在这里插入图片描述

accept入站事件处理分析

在这里插入图片描述

read入站事件处理分析

在这里插入图片描述

以上就是pipeline的分析

一般典型的服务器在每个通道的管道中都有以下处理程序，但是根据协议和业务逻辑的复杂性和特征，会有所不同：

1.协议解码器–将二进制数据转为java对象

2.协议编码器–将java对象转换为二进制

3.业务逻辑处理程序–执行实际业务逻辑（如数据库访问）

netty的高度可拓展性就是因为有了pipeline责任链的设计模式。

Netty底层ByteBuf原理

ByteBuf是为了解决Java NIO ByteBuffer的问题和满足网络应用程序开发人员日常需求而设计的。

JDK ByteBuffer缺点：

1.无法动态扩容，长度固定，不能动态拓展和收缩，当数据大于ByteBuffer容量时，会引发索引越界。

2.API使用复杂

读写切换的时候需要手动调用flip和rewind等方法，需要谨慎使用，否则容易出错。

ByteBuf对于ByteBuffer的增强

1.API操作便捷性

2.动态扩容

3.多种ByteBuf实现

4.高效零拷贝机制

ByteBuf三个重要特性：capacity（容量）、readerindex（读取位置）、writeindex（写入位置）。readerindex和writeindex这两个指针可以支持顺序读写操作。

在这里插入图片描述

常用方法

随机访问索引getByte

顺序读取read\*

顺序写入write\*

清除已读内容discardReadsByte

清除缓冲区clear

搜索操作

标记和重置

引用计数和释放

实际用法

ByteBuf使用的时候，netty不推荐直接new，而是通过Unpooled.buffer来获取实例创建对象

代码示例

public class ByteBufDemo {

@Test

public void apiTest() {

// +-------------------+------------------+------------------+

// | discardable bytes | readable bytes | writable bytes |

// | | (CONTENT) | |

// +-------------------+------------------+------------------+

// | | | |

// 0 <= readerIndex <= writerIndex <= capacity

// 1.创建一个非池化的ByteBuf，大小为10个字节

ByteBuf buf = Unpooled.buffer(10);

System.out.println("原始ByteBuf为====================>" + buf.toString());

System.out.println("1.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 2.写入一段内容

byte[] bytes = {1, 2, 3, 4, 5};

buf.writeBytes(bytes);

System.out.println("写入的bytes为====================>" + Arrays.toString(bytes));

System.out.println("写入一段内容后ByteBuf为===========>" + buf.toString());

System.out.println("2.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 3.读取一段内容

byte b1 = buf.readByte();

byte b2 = buf.readByte();

System.out.println("读取的bytes为====================>" + Arrays.toString(new byte[]{b1, b2}));

System.out.println("读取一段内容后ByteBuf为===========>" + buf.toString());

System.out.println("3.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 4.将读取的内容丢弃

buf.discardReadBytes();

System.out.println("将读取的内容丢弃后ByteBuf为========>" + buf.toString());

System.out.println("4.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 5.清空读写指针

buf.clear();

System.out.println("将读写指针清空后ByteBuf为==========>" + buf.toString());

System.out.println("5.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 6.再次写入一段内容，比第一段内容少

byte[] bytes2 = {1, 2, 3};

buf.writeBytes(bytes2);

System.out.println("写入的bytes为====================>" + Arrays.toString(bytes2));

System.out.println("写入一段内容后ByteBuf为===========>" + buf.toString());

System.out.println("6.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 7.将ByteBuf清零

buf.setZero(0, buf.capacity());

System.out.println("将内容清零后ByteBuf为==============>" + buf.toString());

System.out.println("7.ByteBuf中的内容为================>" + Arrays.toString(buf.array()) + "\n");

// 8.再次写入一段超过容量的内容

byte[] bytes3 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

buf.writeBytes(bytes3);

System.out.println("写入的bytes为====================>" + Arrays.toString(bytes3));

System.out.println("写入一段内容后ByteBuf为===========>" + buf.toString());

System.out.println("8.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 随机访问索引 getByte

// 顺序读 read\*

// 顺序写 write\*

// 清除已读内容 discardReadBytes

// 清除缓冲区 clear

// 搜索操作

// 标记和重置

// 完整代码示例：参考

// 搜索操作 读取指定位置 buf.getByte(1);

//

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

写入代码

@Override

public ByteBuf writeByte(int value) {

ensureWritable0(1);

\_setByte(writerIndex++, value);

return this;

}

final void ensureWritable0(int minWritableBytes) {

ensureAccessible();// 检查ByteBuf对象的引用计数，如果为0，则不允许再进行操作

if (minWritableBytes <= writableBytes()) {

return;

}

if (checkBounds) {

if (minWritableBytes > maxCapacity - writerIndex) {

throw new IndexOutOfBoundsException(String.format(

"writerIndex(%d) + minWritableBytes(%d) exceeds maxCapacity(%d): %s",

writerIndex, minWritableBytes, maxCapacity, this));

}

}

// 扩容计算，当前容量扩容至2的幂次方大小。

// Normalize the current capacity to the power of 2.

int newCapacity = alloc().calculateNewCapacity(writerIndex + minWritableBytes, maxCapacity);

// Adjust to the new capacity.

capacity(newCapacity);// 设置新的容量值

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

扩容方法

@Override

public int calculateNewCapacity(int minNewCapacity, int maxCapacity) {

if (minNewCapacity < 0) {

throw new IllegalArgumentException("minNewCapacity: " + minNewCapacity + " (expected: 0+)");

}

if (minNewCapacity > maxCapacity) {

throw new IllegalArgumentException(String.format(

"minNewCapacity: %d (expected: not greater than maxCapacity(%d)",

minNewCapacity, maxCapacity));

}// 阈值4兆。 这个阈值的用意：容量要求4兆以内，每次扩容以2的倍数进行计算。超过4兆容量，另外的计算方式

final int threshold = CALCULATE\_THRESHOLD; // 4 MiB page

if (minNewCapacity == threshold) {// 新容量的最小要求，如果等于阈值，则立刻返回

return threshold;

}

// If over threshold, do not double but just increase by threshold.

if (minNewCapacity > threshold) {// 如果新容量的最小要求大于阈值

int newCapacity = minNewCapacity / threshold \* threshold;// 新容量 = 新容量最小要求/阈值 \* 阈值

if (newCapacity > maxCapacity - threshold) {// 大于 max(默认Integer.MAX\_VALUE)，则返回最大限制值

newCapacity = maxCapacity;

} else {

newCapacity += threshold;// 否则新容量 = 新容量最小要求/阈值 \* 阈值 + 阈值

}

return newCapacity;

}

// 如果容量要求没超过阈值，则从64字节开始，不断增加一倍，直至满足新容量最小要求

// Not over threshold. Double up to 4 MiB, starting from 64.

int newCapacity = 64;

while (newCapacity < minNewCapacity) {

newCapacity <<= 1;

}

return Math.min(newCapacity, maxCapacity);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

capacity默认256字节、最大值：Integer.MAX\_VALUE(2GB)

write\*方法调用时，通过AbstractByteBuf类的ensureWritable0方法进行检查。

容量计算方法：AbstractByteBufAllocator类的calculateNewCapacity(新capacity的最小值，capacity的最大值)

根据新的capacity最小值要求，对应两套算法：

1.没超过4M：从64byte开始，每次乘2，直到计算出newCapacity满足新容量的最小值。如：原256，已写250，继续写10，需要的容量最小值是261，新capacity是64 \* 2 \* 2 \* 2 = 512

2.新capacity = newcapacity最小值/4M \* 4M + 4M。如原3M，已写3M，继续写2M，需要的容量最小值时5M，则容量时9M。

4M的来源是一个固定的阈值AbstractByteBufAllocator类static final int CALCULATE\_THRESHOLD = 1048576 \* 4; // 4 MiB page

bytebuf的使用实现方式

在这里插入图片描述

在使用中，都是通过ByteBufAllocator接口实现的分配器进行申请，同时分配器具备内存管理功能。

堆内内存

由UnpooledHeapByteBuf实现

private final ByteBufAllocator alloc;

byte[] array;

private ByteBuffer tmpNioBuf;

本质是数组，对于数组的一个封装

之前的代码实例就是堆内内存，这里不做重述。

堆外内存

由UnpooledDirectByteBuf实现

private final ByteBufAllocator alloc;

private ByteBuffer buffer;

private ByteBuffer tmpNioBuf;

private int capacity;

private boolean doNotFree;

本质是NIO的bytebuffer

堆外内存输出内容时不能使用buf.array()方法，netty和javaNIO都没有实现此方法。其他方法使用都与堆内内存一样。

代码实例

public class DirectByteBufDemo {

@Test

public void apiTest() {

// +-------------------+------------------+------------------+

// | discardable bytes | readable bytes | writable bytes |

// | | (CONTENT) | |

// +-------------------+------------------+------------------+

// | | | |

// 0 <= readerIndex <= writerIndex <= capacity

// 1.创建一个非池化的ByteBuf，大小为10个字节

ByteBuf buf = Unpooled.directBuffer(10);

System.out.println("原始ByteBuf为====================>" + buf.toString());

// System.out.println("1.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 2.写入一段内容

byte[] bytes = {1, 2, 3, 4, 5};

buf.writeBytes(bytes);

System.out.println("写入的bytes为====================>" + Arrays.toString(bytes));

System.out.println("写入一段内容后ByteBuf为===========>" + buf.toString());

//System.out.println("2.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 3.读取一段内容

byte b1 = buf.readByte();

byte b2 = buf.readByte();

System.out.println("读取的bytes为====================>" + Arrays.toString(new byte[]{b1, b2}));

System.out.println("读取一段内容后ByteBuf为===========>" + buf.toString());

//System.out.println("3.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 4.将读取的内容丢弃

buf.discardReadBytes();

System.out.println("将读取的内容丢弃后ByteBuf为========>" + buf.toString());

//System.out.println("4.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 5.清空读写指针

buf.clear();

System.out.println("将读写指针清空后ByteBuf为==========>" + buf.toString());

//System.out.println("5.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 6.再次写入一段内容，比第一段内容少

byte[] bytes2 = {1, 2, 3};

buf.writeBytes(bytes2);

System.out.println("写入的bytes为====================>" + Arrays.toString(bytes2));

System.out.println("写入一段内容后ByteBuf为===========>" + buf.toString());

// System.out.println("6.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 7.将ByteBuf清零

buf.setZero(0, buf.capacity());

System.out.println("将内容清零后ByteBuf为==============>" + buf.toString());

// System.out.println("7.ByteBuf中的内容为================>" + Arrays.toString(buf.array()) + "\n");

// 8.再次写入一段超过容量的内容

byte[] bytes3 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

buf.writeBytes(bytes3);

System.out.println("写入的bytes为====================>" + Arrays.toString(bytes3));

System.out.println("写入一段内容后ByteBuf为===========>" + buf.toString());

// System.out.println("8.ByteBuf中的内容为===============>" + Arrays.toString(buf.array()) + "\n");

// 随机访问索引 getByte

// 顺序读 read\*

// 顺序写 write\*

// 清除已读内容 discardReadBytes

// 清除缓冲区 clear

// 搜索操作

// 标记和重置

// 完整代码示例：参考

// 搜索操作 读取指定位置 buf.getByte(1);

//

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

Unsafe的实现

unsafe代表不安全的操作。但是更底层的操作会带来性能的提升和特殊功能，netty中会尽力使用unsafe。java的特性是“一处编写到处运行”，所以他针对底层的内存或其他操作做了很多封装。而unsafe提供了一系列操作底层的方法，可能会导致不兼容和不可知异常。所以在不是完全掌握的情况下不推荐使用unsafe。

1.info. 仅返回一些低级的内存信息：addressSize、pageSize；

2.Objects. 提供用于操作对象及其字段的方法：allocateInstance、objectFieldOffset；

3.Classes. 提供用于操作类及其静态字段的方法：staticFieldOffset、defineClass。defineAnonymousClass、ensureClassInitialized；

4.Synchronization. 低级的同步原语：monitorEnter、tyrMonitorEnter、monitorExit、compareAndSwapInt、putOrderedInt；

5.Memory. 直接访问内存方法：allocateMemory、copyMemory、freeMemory、getAddress、getInt、putInt；

6.Arrays. 操作数组：arrayBaseOffset、arrayIndexScale

内存复用

普通申请堆外内存时，直接new一个堆外内存不会有内存复用，当使用pool来申请bytebuf时会触发内存复用。

实例代码

@Override

protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {

PoolThreadCache cache = threadCache.get();// 获取cache对象

PoolArena<ByteBuffer> directArena = cache.directArena; // 从cache对象中去除arena

// arena可以理解为一个netty提供的实际进行buf的分配和管理工具

final ByteBuf buf;

if (directArena != null) {

buf = directArena.allocate(cache, initialCapacity, maxCapacity);

} else {// 如果没有arena，就用unpool了

buf = PlatformDependent.hasUnsafe() ?

UnsafeByteBufUtil.newUnsafeDirectByteBuf(this, initialCapacity, maxCapacity) :

new UnpooledDirectByteBuf(this, initialCapacity, maxCapacity);

}

return toLeakAwareBuffer(buf);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

newByteBuf代码

内存复用在这里newInstance实现

@Override

protected PooledByteBuf<ByteBuffer> newByteBuf(int maxCapacity) {

if (HAS\_UNSAFE) {// 能拿到unsafe对象，就用unsafe。此处有复用的实现

return PooledUnsafeDirectByteBuf.newInstance(maxCapacity);

} else {

return PooledDirectByteBuf.newInstance(maxCapacity);

}

}

/PooledUnsafeDirectByteBuf中

static PooledUnsafeDirectByteBuf newInstance(int maxCapacity) {

PooledUnsafeDirectByteBuf buf = RECYCLER.get();// 从名字就能看出来，这是一种buf对象的复用机制

buf.reuse(maxCapacity);// 取出来的可能是之前buf，使用前清理一下

return buf;

}

///RECYCLER中

/\*

\* 当获取对象时，首先从线程变量里取一个栈，从栈里面取handle变量，从handle获取value；

\* value就是我们的buffer对象，如果已经有buffer对象那么就拿来用，如果没有就创建一个newObject（handle）

\*/

@SuppressWarnings("unchecked")

public final T get() {

if (maxCapacityPerThread == 0) {

return newObject((Handle<T>) NOOP\_HANDLE);

}// 在线程变量中维护了一个栈，弹出一个handle对象

Stack<T> stack = threadLocal.get();

DefaultHandle<T> handle = stack.pop();

if (handle == null) {// 如果没有则创建一个新的handle

handle = stack.newHandle();

handle.value = newObject(handle);

}// 返回已经存在的buf对象

return (T) handle.value;// value是当buf回收时调用deallocate，存放进来的。

}

//newObject在PooledUnsafeDirectByteBuf中

private static final Recycler<PooledUnsafeDirectByteBuf> RECYCLER = new Recycler<PooledUnsafeDirectByteBuf>() {

@Override

protected PooledUnsafeDirectByteBuf newObject(Handle<PooledUnsafeDirectByteBuf> handle) {

return new PooledUnsafeDirectByteBuf(handle, 0);// 没有可复用的才会创建新的

}

};

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

内存回收

有了复用就要有回收，通过get方法可以取到栈里的对象，那么回收时是如何把对象放入栈的，这里使用的是Recycle的recycle方法。

@Deprecated

public final boolean recycle(T o, Handle<T> handle) {

if (handle == NOOP\_HANDLE) {

return false;

}

DefaultHandle<T> h = (DefaultHandle<T>) handle;

if (h.stack.parent != this) {

return false;

}

h.recycle(o);

return true;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

recycle（）将在ByteBuf调用之后再调用，用于释放bytebuf。下面来看一下netty的bytebuf是如何使用的

PooledByteBuf类继承了AbstractReferenceCountedByteBuf类，实现了引用计数功能。每个buf的使用都会计数，当计数为0时就会释放，有点类似GC。而释放的方法是release调用release0，然后再调用deallocate。

// 这是netty自己的一种buffer释放机制。概念上和GC类似

public abstract class AbstractReferenceCountedByteBuf extends AbstractByteBuf {// 这个要看，实现了引用计数

private static final long REFCNT\_FIELD\_OFFSET;

private static final AtomicIntegerFieldUpdater<AbstractReferenceCountedByteBuf> refCntUpdater =

AtomicIntegerFieldUpdater.newUpdater(AbstractReferenceCountedByteBuf.class, "refCnt");

// even => "real" refcount is (refCnt >>> 1); odd => "real" refcount is 0

@SuppressWarnings("unused")

private volatile int refCnt = 2;

static {

long refCntFieldOffset = -1;

try {

if (PlatformDependent.hasUnsafe()) {

refCntFieldOffset = PlatformDependent.objectFieldOffset(

AbstractReferenceCountedByteBuf.class.getDeclaredField("refCnt"));

}

} catch (Throwable ignore) {

refCntFieldOffset = -1;

}

REFCNT\_FIELD\_OFFSET = refCntFieldOffset;

}

private static int realRefCnt(int rawCnt) {

return (rawCnt & 1) != 0 ? 0 : rawCnt >>> 1;

}

protected AbstractReferenceCountedByteBuf(int maxCapacity) {

super(maxCapacity);

}

private int nonVolatileRawCnt() {

// TODO: Once we compile against later versions of Java we can replace the Unsafe usage here by varhandles.

return REFCNT\_FIELD\_OFFSET != -1 ? PlatformDependent.getInt(this, REFCNT\_FIELD\_OFFSET)

: refCntUpdater.get(this);

}

@Override

int internalRefCnt() {

// Try to do non-volatile read for performance as the ensureAccessible() is racy anyway and only provide

// a best-effort guard.

return realRefCnt(nonVolatileRawCnt());

}

/\*\* 返回该对象的引用计数。如果为0，则表示该对象已解除分配，是时候释放了。 \*/

@Override

public int refCnt() {

return realRefCnt(refCntUpdater.get(this));

}

/\*\*

\* An unsafe operation intended for use by a subclass that sets the reference count of the buffer directly

\*/

protected final void setRefCnt(int newRefCnt) {

refCntUpdater.set(this, newRefCnt << 1); // overflow OK here

}

/\*\* 引用计数+1 \*/

@Override

public ByteBuf retain() {

return retain0(1);

}

@Override

public ByteBuf retain(int increment) {

return retain0(checkPositive(increment, "increment"));

}

private ByteBuf retain0(final int increment) {

// all changes to the raw count are 2x the "real" change

int adjustedIncrement = increment << 1; // overflow OK here

int oldRef = refCntUpdater.getAndAdd(this, adjustedIncrement);

if ((oldRef & 1) != 0) {

throw new IllegalReferenceCountException(0, increment);

}

// don't pass 0!

if ((oldRef <= 0 && oldRef + adjustedIncrement >= 0)

|| (oldRef >= 0 && oldRef + adjustedIncrement < oldRef)) {

// overflow case

refCntUpdater.getAndAdd(this, -adjustedIncrement);

throw new IllegalReferenceCountException(realRefCnt(oldRef), increment);

}

return this;

}

@Override

public ByteBuf touch() {

return this;

}

@Override

public ByteBuf touch(Object hint) {

return this;

}

/\*\* 将引用计数减少1，并在引用计数达到0时释放该对象。 \*/

@Override

public boolean release() {

return release0(1);

}

@Override

public boolean release(int decrement) {

return release0(checkPositive(decrement, "decrement"));

}

private boolean release0(int decrement) {

int rawCnt = nonVolatileRawCnt(), realCnt = toLiveRealCnt(rawCnt, decrement);

if (decrement == realCnt) {// 如果要减的数量和目前的引用数量相等，代表要释放了

if (refCntUpdater.compareAndSet(this, rawCnt, 1)) {// cas无锁机制，保证线程安全

deallocate();// 执行释放

return true;

}

return retryRelease0(decrement);// 自旋锁重试

}

return releaseNonFinal0(decrement, rawCnt, realCnt);

}

private boolean releaseNonFinal0(int decrement, int rawCnt, int realCnt) {

if (decrement < realCnt

// all changes to the raw count are 2x the "real" change

&& refCntUpdater.compareAndSet(this, rawCnt, rawCnt - (decrement << 1))) {

return false;

}

return retryRelease0(decrement);

}

private boolean retryRelease0(int decrement) {

for (;;) {// 自旋锁

int rawCnt = refCntUpdater.get(this), realCnt = toLiveRealCnt(rawCnt, decrement);

if (decrement == realCnt) {

if (refCntUpdater.compareAndSet(this, rawCnt, 1)) {

deallocate();

return true;

}

} else if (decrement < realCnt) {

// all changes to the raw count are 2x the "real" change

if (refCntUpdater.compareAndSet(this, rawCnt, rawCnt - (decrement << 1))) {

return false;

}

} else {

throw new IllegalReferenceCountException(realCnt, -decrement);

}

Thread.yield(); // this benefits throughput under high contention

}

}

/\*\*

\* Like {@link #realRefCnt(int)} but throws if refCnt == 0

\*/

private static int toLiveRealCnt(int rawCnt, int decrement) {

if ((rawCnt & 1) == 0) {

return rawCnt >>> 1;

}

// odd rawCnt => already deallocated

throw new IllegalReferenceCountException(0, -decrement);

}

/\*\*

\* Called once {@link #refCnt()} is equals 0.// 执行释放，不同的ByteBuf类型，回收的方式不同

\*/

protected abstract void deallocate();

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

deallocate是一个abstract方法，这就意味着不同的buf对象会自己实现deallocate方法。看一下poolByteBuf实现

@Override

protected final void deallocate() {

if (handle >= 0) {

final long handle = this.handle;

this.handle = -1;

memory = null;

tmpNioBuf = null;

chunk.arena.free(chunk, handle, maxLength, cache);

chunk = null;

recycle();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

在最后调用了recycle方法。再netty中释放buf是在责任链中实现的，我们可以直接手动释放((ByteBuf) msg).release();也可以继续向后传播ctx.fireChannelRead(msg);直到最后tailContext会调用release释放buf。注意：如果责任链的handler实现时直接使用新buf，这将会导致原有的buf不释放，所以不要手动添加新buf，使用netty定义的buf会更快更高效的实现buf的复用

public class EchoServerHandler extends ChannelInboundHandlerAdapter {

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) {

System.out.println("收到数据：" + ((ByteBuf)msg).toString(Charset.defaultCharset()));

//ctx.write(Unpooled.wrappedBuffer("98877".getBytes()));

// ((ByteBuf) msg).release();

ctx.fireChannelRead(msg);

}

@Override

public void channelReadComplete(ChannelHandlerContext ctx) {

ctx.flush();

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {

// Close the connection when an exception is raised.

cause.printStackTrace();

ctx.close();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

在上述获取新实例的代码中，调用了buf.reuse清理方法,将buf的属性重新初始化

static PooledUnsafeDirectByteBuf newInstance(int maxCapacity) {

PooledUnsafeDirectByteBuf buf = RECYCLER.get();// 从名字就能看出来，这是一种buf对象的复用机制

buf.reuse(maxCapacity);// 取出来的可能是之前buf，使用前清理一下

return buf;

}

1

2

3

4

5

内存分配

allocate代码

PoolThreadCache缓存池中，netty将内存分为不同层次，如16M、8M、4M、2M、1M、512K等等，根据申请时capacity的不同而选择不同内存区块

directArena.allocate代码，如果PoolThreadCache满了之后，netty会向JVM申请新的内存，但是这些内存不会存在缓存池中。

PooledByteBuf<T> allocate(PoolThreadCache cache, int reqCapacity, int maxCapacity) {

PooledByteBuf<T> buf = newByteBuf(maxCapacity);// 获取到一个buf对象

allocate(cache, buf, reqCapacity);// 开始分配内存

return buf;

}

private void allocate(PoolThreadCache cache, PooledByteBuf<T> buf, final int reqCapacity) {

final int normCapacity = normalizeCapacity(reqCapacity);

if (isTinyOrSmall(normCapacity)) { // capacity < pageSize // 如果需要的容量小于pageSize

int tableIdx;

PoolSubpage<T>[] table;

boolean tiny = isTiny(normCapacity);

if (tiny) { // < 512 // 小于512字节，分配一个tiny缓存

if (cache.allocateTiny(this, buf, reqCapacity, normCapacity)) {

// was able to allocate out of the cache so move on

return;

}

tableIdx = tinyIdx(normCapacity);

table = tinySubpagePools;

} else {

if (cache.allocateSmall(this, buf, reqCapacity, normCapacity)) {

// was able to allocate out of the cache so move on

return;

}

tableIdx = smallIdx(normCapacity);

table = smallSubpagePools;

}

final PoolSubpage<T> head = table[tableIdx];

/\*\*

\* Synchronize on the head. This is needed as {@link PoolChunk#allocateSubpage(int)} and

\* {@link PoolChunk#free(long)} may modify the doubly linked list as well.

\*/

synchronized (head) {

final PoolSubpage<T> s = head.next;

if (s != head) {

assert s.doNotDestroy && s.elemSize == normCapacity;

long handle = s.allocate();

assert handle >= 0;

s.chunk.initBufWithSubpage(buf, handle, reqCapacity);

incTinySmallAllocation(tiny);

return;

}

}

synchronized (this) {

allocateNormal(buf, reqCapacity, normCapacity);

}

incTinySmallAllocation(tiny);

return;

}

if (normCapacity <= chunkSize) {

if (cache.allocateNormal(this, buf, reqCapacity, normCapacity)) {

// was able to allocate out of the cache so move on

return;

}

synchronized (this) {

allocateNormal(buf, reqCapacity, normCapacity);

++allocationsNormal;

}

} else {

// Huge allocations are never served via the cache so just call allocateHuge

allocateHuge(buf, reqCapacity);// 分配超大内存，不会进行缓存

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

小结

PoolThreadCache ：PooledbyteBufAllocate实例维护的一个线程变量。

多种分类的“MomeryRegistorCache数组”用做内存缓存，MomeryRegistorCache内部是链表，队列里面存Chunk。

PoolChunk里面维护了内存引用，内存复用的做法就是buf的momery指向Chunk的memory。

PooledByteBufAllocator.ioBuffer运作过程图

在这里插入图片描述

所以需要注意，一定要释放缓存，如果不释放就不会被回收，从而造成内存泄漏。netty中默认使用的是pooledUnsafeDirectByteBuf，pool目的是为了复用，Unsafe的目的是为了性能提升，Direct还是为了性能提升，所以netty才能实现高效高性能的特性。而netty建议开发者使用unpooledHeapByteBuf。

零拷贝机制

零拷贝机制是一种应用层的实现。和JVM、操作系统内存机制并无过多关联。

1.CompositeByteBuf，将多个ByteBuf合并为一个逻辑上的ByteBuf，避免了各个ByteBuf之间的拷贝。

在这里插入图片描述

2.wrapedBuffer（）方法，将byte[]数组包装成ByteBuf对象。

在这里插入图片描述

3.slice（）方法，将一个ByteBuf对象切分成多个ByteBuf对象

在这里插入图片描述

所谓零拷贝就是不改变原buf只是逻辑上做拆分、合并、包装。减少了大量的内存复制，由此提升性能。

代码示例

public class ZeroCopyTest {

@org.junit.Test

public void wrapTest() {

byte[] arr = {1, 2, 3, 4, 5};

ByteBuf byteBuf = Unpooled.wrappedBuffer(arr);

System.out.println(byteBuf.getByte(4));

arr[4] = 6;

System.out.println(byteBuf.getByte(4));

}

@org.junit.Test

public void sliceTest() {

ByteBuf buffer1 = Unpooled.wrappedBuffer("hello".getBytes());

ByteBuf newBuffer = buffer1.slice(1, 2);

newBuffer.unwrap();

System.out.println(newBuffer.toString());

}

@org.junit.Test

public void compositeTest() {

ByteBuf buffer1 = Unpooled.buffer(3);

buffer1.writeByte(1);

ByteBuf buffer2 = Unpooled.buffer(3);

buffer2.writeByte(4);

CompositeByteBuf compositeByteBuf = Unpooled.compositeBuffer();

CompositeByteBuf newBuffer = compositeByteBuf.addComponents(true, buffer1, buffer2);

System.out.println(newBuffer);

}

}