
目录

Introduction	1.1
Bootstrap	1.2
类AbstractBootstrap	1.2.1
类Bootstrap	1.2.2
类ServerBootstrap	1.2.3
Eventloop	1.3
接口EventExecutor	1.3.1
接口EventLoop	1.3.2
NIO EventLoop	1.3.3
温习JDK Executor	1.3.3.1
类AbstractEventExecutor	1.3.3.2
类AbstractScheduledEventExecutor	1.3.3.3
类SingleThreadEventExecutor	1.3.3.4
类SingleThreadEventLoop	1.3.3.5
类NioEventLoop	1.3.3.6
Channel	1.4
接口Channel	1.4.1
类AbstractChannel	1.4.1.1
Channel Factory	1.4.2
Channel Unsafe	1.4.3
Channel Handler	1.4.4
ChannelHandler相关接口	1.4.4.1
ChannelHandlerAdapter相关类	1.4.4.2
类ChannelInitializer	1.4.4.3
类ChannelHandlerAppender	1.4.4.4
Channel Handler Context	1.4.5
接口ChannelHandlerContext	1.4.5.1

类AbstractChannelHandlerContext	1.4.5.2
类DefaultChannelHandlerContext	1.4.5.3
Channel Pipeline	1.4.6
接口ChannelPipeline	1.4.6.1
类DefaultChannelPipeline	1.4.6.2
Buffer	1.5
ByteBuf继承结构	1.5.1
ByteBuf重要特性	1.5.2
Pooled buffer	1.5.2.1
Reference Count	1.5.2.2
接口ReferenceCounted	1.5.2.2.1
类AbstractReferenceCountedByteBuf	1.5.2.2.2
buffer的释放	1.5.2.2.3
类UnreleasableByteBuf	1.5.2.2.4
buffer泄露检测	1.5.2.2.5
类ResourceLeakDetector	1.5.2.2.6
类LeakAwareByteBuf	1.5.2.2.7
Zero Copy	1.5.2.3
类SlicedByteBuf	1.5.2.3.1
方法Unpooled.wrappedBuffer()	1.5.2.3.2
类CompositeByteBuf	1.5.2.3.3
codec	1.6
decoder	1.6.1
decoder	1.6.1.1
http	1.7
http object	1.7.1
http类实现	1.7.1.1
http header	1.7.1.2
http codec	1.7.2
类HttpServerCodec	1.7.2.1

Netty学习笔记

Netty学习笔记, 包括代码学习, 源码分析.基于netty4.1, 用于记录Netty学习过程的各种信息和心得.

Netty的代码结构

netty源码的组织结构和包含的package如下：

1. common
 - `io.netty.util`
 - `io.netty.util.concurrent`
2. resolver
 - `io.netty.resolver`
3. resolver-dns
 - `io.netty.resolver.dns`
4. buffer
 - `io.netty.buffer`
5. transport
 - `io.netty.bootstrap`
 - `io.netty.channel`
 - `io.netty.channel.EventLoop`
6. transport-native-epoll
 - `io.netty.channel.epoll`
 - `io.netty.channel.unix`
7. transport-rxtx
 - `io.netty.channel.rxtx`
8. transport-sctp
 - `io.netty.channel.sctp`
 - `com.sun.nio.sctp`
 - `io.netty.handler.codec.sctp`
9. transport-udt
 - `io.netty.channel.udt`
10. handler

- `io.netty.handler.ipfilter`
- `io.netty.handler.logging`
- `io.netty.handler.ssl`
- `io.netty.handler.stream`
- `io.netty.handler.timeout`
- `io.netty.handler.traffic`

11. handler-proxy

- `io.netty.handler.proxy`

12. codec

- `codec: io.netty.handler.codec`
- `dns: io.netty.handler.codec.dns`
- `haproxy: io.netty.handler.codec.haproxy`
- `http: io.netty.handler.codec.http`
- `http2: io.netty.handler.codec.http2`
- `memcache: io.netty.handler.codec.memcache`
- `mqtt: io.netty.handler.codec.mqtt`
- `socks: io.netty.handler.codec.socksx`
- `xml: io.netty.handler.codec.xml`

学习重点和进度

1. resolver

- `io.netty.resolver`

2. buffer

- `io.netty.buffer` (Ongoing)

3. transport

- `io.netty.bootstrap` (Done)
- `io.netty.channel` (Done)
- `io.netty.channel.EventLoop` (Done)

4. handler

5. codec

- `codec: io.netty.handler.codec`
- `http: io.netty.handler.codec.http`
- `http2: io.netty.handler.codec.http2`

Bootstrap

Bootstrap的作用

Bootstrap的作用可以参考AbstractBootstrap的javadoc:

AbstractBootstrap is a helper class that makes it easy to **bootstrap a Channel**.

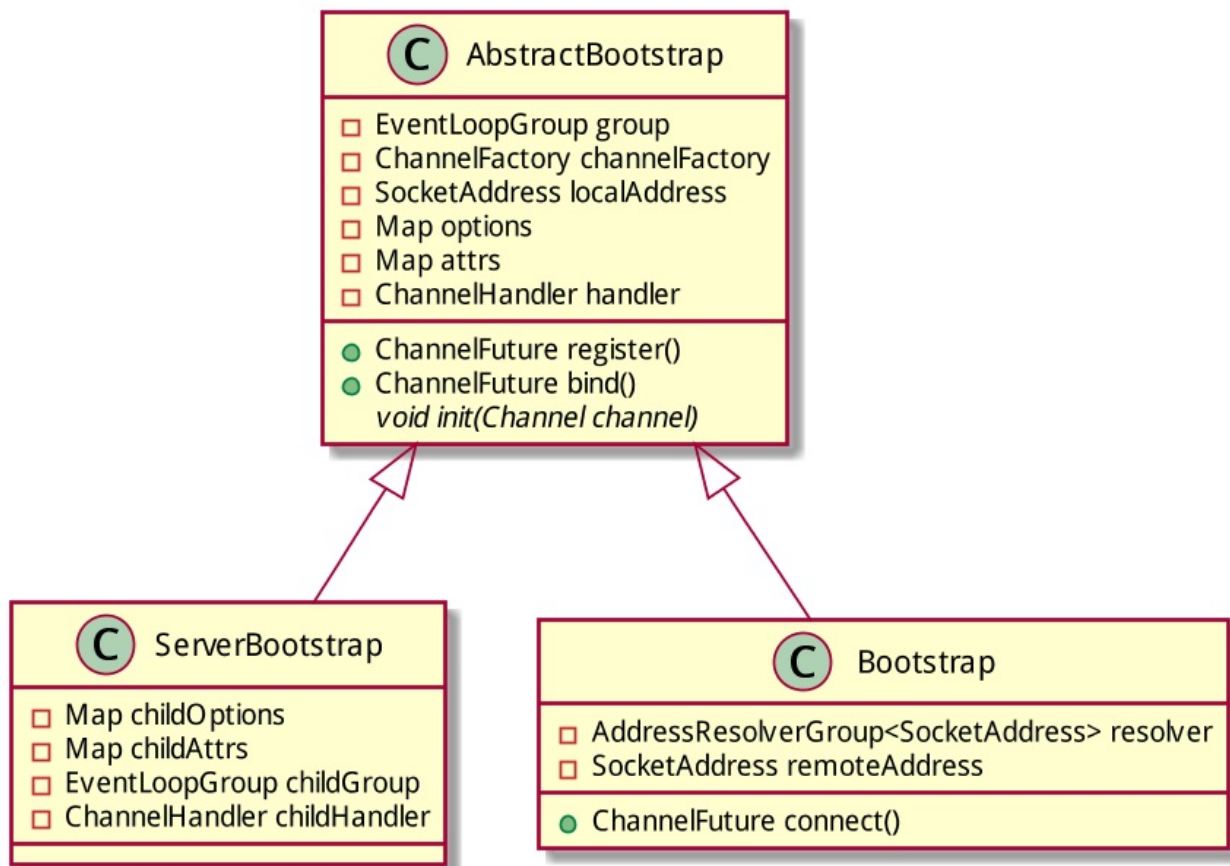
Bootstrap存在的意义就是为了方便的"引导"Channel.

在netty中, 存在两种类型的Channel, 因此也对应有两种Bootstrap:

channel类型	用于引导的bootstrap实现类
ServerChannel	ServerBootstrap
Channel	Bootstrap

Bootstrap的继承结构

在netty的代码中, 类ServerBootstrap和类Bootstrap都继承自基类AbstractBootstrap:



类AbstractBootstrap

类定义

AbstractBootstrap是Bootstrap的基类, 类定义如下:

```
package io.netty.bootstrap;

public abstract class AbstractBootstrap
<B extends AbstractBootstrap<B, C>, C extends Channel>
implements Cloneable {}
```

类定义中的泛型B要求是AbstractBootstrap的子类, 而泛型C要求是Channel的子类.

注意这里的泛型的使用, 非常的巧妙。

成员变量

group属性

```
volatile EventLoopGroup group;

public B group(EventLoopGroup group) {
    if (group == null) {
        throw new NullPointerException("group");
    }
    if (this.group != null) {
        throw new IllegalStateException("group set already");
    }
    this.group = group;
    return (B) this;
}

public EventLoopGroup group() {
    return group;
}
```

注意this.group只能设置一次, 这意味着group(group)方法只能被调用一次.

localAddress属性

localAddress用于绑定本地终端, 有多个设值的方法:

```
private volatile SocketAddress localAddress;

public B localAddress(SocketAddress localAddress) {
    this.localAddress = localAddress;
    return (B) this;
}

public B localAddress(int inetPort) {
    return localAddress(new InetSocketAddress(inetPort));
}

public B localAddress(String inetHost, int inetPort) {
    return localAddress(new InetSocketAddress(inetHost, inetPort));
}

public B localAddress(InetAddress inetHost, int inetPort) {
    return localAddress(new InetSocketAddress(inetHost, inetPort));
}

final SocketAddress localAddress() {
    return localAddress;
}
```

这些重载的localAddress(), 最终都指向了InetSocketAddress的几个构造函数.

options 属性

options属性是一个LinkedHashMap, option()方法用于设置单个的key/value, 如果value为null则删除该key.

```
private final Map<ChannelOption<?>, Object> options = new Linked
HashMap<ChannelOption<?>, Object>();

public <T> B option(ChannelOption<T> option, T value) {
    if (option == null) {
        throw new NullPointerException("option");
    }
    if (value == null) {
        synchronized (options) {
            options.remove(option);
        }
    } else {
        synchronized (options) {
            options.put(option, value);
        }
    }
    return (B) this;
}

final Map<ChannelOption<?>, Object> options() {
    return options;
}
```

attrs属性

attrs和options属性类似。

```
private final Map<AttributeKey<?>, Object> attrs = new LinkedHas  
hMap<AttributeKey<?>, Object>();  
  
public <T> B attr(AttributeKey<T> key, T value) {  
    if (key == null) {  
        throw new NullPointerException("key");  
    }  
    if (value == null) {  
        synchronized (attrs) {  
            attrs.remove(key);  
        }  
    } else {  
        synchronized (attrs) {  
            attrs.put(key, value);  
        }  
    }  
    return (B) this;  
}  
  
final Map<AttributeKey<?>, Object> attrs() {  
    return attrs;  
}
```

handler属性

```
private volatile ChannelHandler handler;

public B handler(ChannelHandler handler) {
    if (handler == null) {
        throw new NullPointerException("handler");
    }
    this.handler = handler;
    return (B) this;
}

final ChannelHandler handler() {
    return handler;
}
```

channelFactory属性

新旧两个ChannelFactory

channelFactory这个属性有点麻烦, 根源在于ChannelFactory这个类, netty中有新旧两个ChannelFactory, 具体介绍见 [Channel Factory](#)

混合使用

但是现在的情况是内部已经转为使用新类, 对外的接口还是继续保持使用原来的旧类, 因此代码有些混乱:

```
// 这里的channelFactory的类型定义用的是旧类, 因此需要加SuppressWarnings
@SuppressWarnings("deprecation")
private volatile ChannelFactory< ? extends C> channelFactory;

// 返回的channelFactory也是用的旧类, 没的说, 继续SuppressWarnings
@SuppressWarnings("deprecation")
final ChannelFactory< ? extends C> channelFactory() {
    return channelFactory;
}

// 这个方法的参数是旧的"io.netty.bootstrap.ChannelFactory", 已经被标志
// 为"@Deprecated", 尽量用下面的方法
@Deprecated
public B channelFactory(ChannelFactory< ? extends C> channelFact
ory) {
    if (channelFactory == null) {
        throw new NullPointerException("channelFactory");
    }
    if (this.channelFactory != null) {
        throw new IllegalStateException("channelFactory set alre
ady");
    }

    this.channelFactory = channelFactory;
    return (B) this;
}

// 这个方法是现在推荐使用的设置channelFactory的方法, 使用新类"io.netty.
// channel.ChannelFactory"
@SuppressWarnings({ "unchecked", "deprecation" })
public B channelFactory(io.netty.channel.ChannelFactory< ? exten
ds C> channelFactory) {
    // 但是底层的实现还是调用回上面被废弃的channelFactory()方法
    // 因为新类是继承自旧类的, 所有只要简单转一下类型就好
    return channelFactory((ChannelFactory<C>) channelFactory);
}
```

此外还有一个channel()方法可以非常方便的设置channelFactory:

```
public B channel(Class< ? extends C> channelClass) {
    if (channelClass == null) {
        throw new NullPointerException("channelClass");
    }
    return channelFactory(new ReflectiveChannelFactory<C>(channelClass));
}
```

类方法

validate()

validate()用于检验所有的参数, 实际代码中检查的是group和channelFactory两个参数, 这两个参数必须设置不能为空:

```
public B validate() {
    if (group == null) {
        throw new IllegalStateException("group not set");
    }
    if (channelFactory == null) {
        throw new IllegalStateException("channel or channelFactory not set");
    }
    return (B) this;
}
```

register()

register()方法创建一个新的Channel并将它注册到EventLoop, 在执行前会调用validate()方法做前置检查:

```
public ChannelFuture register() {
    validate();
    return initAndRegister();
}
```


initAndRegister()是关键代码, 细细读一下:

```
final ChannelFuture initAndRegister() {
    // 创建一个新的Channel
    final Channel channel = channelFactory().newChannel();
    try {
        // 调用抽象方法, 子类来做初始化
        init(channel);
    } catch (Throwable t) {
        // 如果出错, 强行关闭这个channel
        channel.unsafe().closeForcibly();
        // as the Channel is not registered yet we need to force
        the usage of the GlobalEventExecutor
        return new DefaultChannelPromise(channel, GlobalEventExe
cutor.INSTANCE).setFailure(t);
    }

    // 创建成功则将这个channel注册到eventloop中
    ChannelFuture regFuture = group().register(channel);
    // 如果注册出错
    if (regFuture.cause() != null) {
        // 判断是否已经注册
        if (channel.isRegistered()) {
            // channel已经注册的就关闭
            channel.close();
        } else {
            // 还没有注册的就强行关闭
            channel.unsafe().closeForcibly();
        }
    }
}

// 如果代码走到这里而且promise没有失败, 那么是下面两种情况之一:
// 1) 如果我们尝试了从event loop中注册, 那么现在注册已经完成
// 现在可以安全的尝试 bind()或者connect(), 因为channel已经注册
成功
// 2) 如果我们尝试了从另外一个线程中注册, 注册请求已经成功添加到event
loop的任务队列中等待后续执行
// 现在可以安全的尝试 bind()或者connect():
// 因为 bind() 或 connect() 会在安排的注册任务之后执行
// 而register(), bind(), 和 connect() 都被确认是同一个线程
```

```
        return regFuture;
    }
```

中途调用的init()方法定义如下, 后面看具体子类代码时再展开.

```
abstract void init(Channel channel) throws Exception;
```

bind()

bind()方法有多个重载, 差异只是bind操作所需的InetSocketAddress参数从何而来而已:

1. 从属性 `this.localAddress` 来

这个时候bind()方法无需参数, 直接使用属性this.localAddress, 当前调用之前 `this.localAddress` 必须有赋值(通过函数localAddress()):

```
public ChannelFuture bind() {
    validate();
    SocketAddress localAddress = this.localAddress;
    if (localAddress == null) {
        throw new IllegalStateException("localAddress not set");
    }
    return doBind(localAddress);
}
```

2. 从bind()方法的输入参数中来

在输入参数中来直接指定localAddress:

```
public ChannelFuture bind(SocketAddress localAddress) {
    validate();
    if (localAddress == null) {
        throw new NullPointerException("localAddress");
    }
    return doBind(localAddress);
}
```

另外为了方便, 重载了下面三个方法, 用不同的方式来创建InetSocketAddress而已:

```
public ChannelFuture bind(int inetPort) {
    return bind(new InetSocketAddress(inetPort));
}

public ChannelFuture bind(String inetHost, int inetPort) {
    return bind(new InetSocketAddress(inetHost, inetPort));
}

public ChannelFuture bind(InetAddress inetHost, int inetPort) {
    return bind(new InetSocketAddress(inetHost, inetPort));
}
```

注: 使用带参数的bind()方法, 忽略了localAddress()设定的参数. 而且也没有设置localAddress属性. 这里的差异, 后面留意.

继续看doBind()方法的细节, 这个依然是这个类的核心内容:

```
private ChannelFuture doBind(final SocketAddress localAddress) {
    // 调用initAndRegister()方法, 先初始化channel, 并注册到event loop
    final ChannelFuture regFuture = initAndRegister();
    final Channel channel = regFuture.channel();
    // 检查注册的channel是否出错
    if (regFuture.cause() != null) {
        return regFuture;
    }
}
```

```

// 检查注册操作是否完成
if (regFuture.isDone()) {
    // 如果完成
    // 在这个点上我们知道注册已经完成并且成功
    // 继续bind操作, 创建一个ChannelPromise
    ChannelPromise promise = channel.newPromise();
    // 调用doBind0()方法来继续真正的bind操作
    doBind0(regFuture, channel, localAddress, promise);
    return promise;
} else {
    // 通常这个时候注册的future应该都已经完成, 但是万一没有, 我们也需要处理
    // 为这个channel创建一个PendingRegistrationPromise
    final PendingRegistrationPromise promise = new PendingRegistrationPromise(channel);
    // 然后给注册的future添加一个listener, 在operationComplete()回调时
    regFuture.addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture future) throws Exception {
            Throwable cause = future.cause();
            // 检查是否出错
            if (cause != null) {
                // 在event loop上注册失败, 因此直接让ChannelPromise失败, 避免一旦我们试图访问这个channel的eventloop导致IllegalStateException
                promise.setFailure(cause);
            } else {
                // 注册已经成功, 因此设置正确的executor以便使用
                // 注: 这里已经以前有过一个bug, 有issue记录
                // See https://github.com/netty/netty/issues/2586
                promise.executor = channel.eventLoop();
            }
            // 调用doBind0()方法来继续真正的bind操作
            doBind0(regFuture, channel, localAddress, promise);
        }
    });
}
});

```

```
        return promise;
    }
}
```

关键的doBind0()方法:

```
private static void doBind0(final ChannelFuture regFuture, final
    Channel channel, final SocketAddress localAddress, final ChannelFuture
    lPromise promise) {

    // 这个方法在channelRegistered()方法触发前被调用。
    // 让handler有机会在它的channelRegistered()实现中构建pipeline
    // 给channel的event loop增加一个一次性任务
    channel.eventLoop().execute(new OneTimeTask() {
        @Override
        public void run() {
            // 检查注册是否成功
            if (regFuture.isSuccess()) {
                // 如果成功则绑定localAddress到channel
                channel.bind(localAddress, promise).addListener(
ChannelFutureListener.CLOSE_ON_FAILURE);
            } else {
                // 如果不成功则设置错误到promise
                promise.setFailure(regFuture.cause());
            }
        }
    });
}
```

类Bootstrap

类Bootstrap用于帮助客户端引导Channel。

bind()方法用于无连接传输如datagram (UDP)。对于常规TCP链接，用connect()方法。

类定义

```
package io.netty.bootstrap;

public class Bootstrap extends AbstractBootstrap<Bootstrap, Channel> {}
```

类成员

resolver属性

resolver默认设置为DefaultAddressResolverGroup.INSTANCE，可以通过resolver()方法来赋值：

```
private static final AddressResolverGroup< ? > DEFAULT_RESOLVER
= DefaultAddressResolverGroup.INSTANCE;
private volatile AddressResolverGroup<SocketAddress> resolver =
(AddressResolverGroup<SocketAddress>) DEFAULT_RESOLVER;

public Bootstrap resolver(AddressResolverGroup< ? > resolver) {
    if (resolver == null) {
        throw new NullPointerException("resolver");
    }
    this.resolver = (AddressResolverGroup<SocketAddress>) resolver;
    return this;
}
```

remoteAddress 属性

remoteAddress 可以通过 remoteAddress() 方法赋值, 有多个重载方法:

```
private volatile SocketAddress remoteAddress;

public Bootstrap remoteAddress(SocketAddress remoteAddress) {
    this.remoteAddress = remoteAddress;
    return this;
}
public Bootstrap remoteAddress(String inetHost, int inetPort) {
    remoteAddress = InetSocketAddress.createUnresolved(inetHost,
inetPort);
    return this;
}
public Bootstrap remoteAddress(InetAddress inetHost, int inetPort) {
    remoteAddress = new InetSocketAddress(inetHost, inetPort);
    return this;
}
```

类方法

validate() 方法

重写了 validate() 方法, 在调用 AbstractBootstrap 的 validate() 方法(检查 group 和 channelFactory)外, 增加了对 handler 的检查:

```
@Override
public Bootstrap validate() {
    super.validate();
    if (handler() == null) {
        throw new IllegalStateException("handler not set");
    }
    return this;
}
```

connect()方法

有多个connect()方法重载，功能都是一样，拿到输入的remoteAddress然后调用doResolveAndConnect()方法：

```
private ChannelFuture doResolveAndConnect(SocketAddress remoteAddress, final SocketAddress localAddress) {
    // 先初始化channel并注册到event loop
    final ChannelFuture regFuture = initAndRegister();
    if (regFuture.cause() != null) {
        // 如果注册失败则退出
        return regFuture;
    }

    final Channel channel = regFuture.channel();
    final EventLoop eventLoop = channel.eventLoop();
    final AddressResolver<SocketAddress> resolver = this.resolver.getResolver(eventLoop);

    if (!resolver.isSupported(remoteAddress) || resolver.isResolved(remoteAddress)) {
        // Resolver 不知道该怎么处理给定的远程地址，或者已经解析
        return doConnect(remoteAddress, localAddress, regFuture, channel.newPromise());
    }

    // 开始解析远程地址
    final Future<SocketAddress> resolveFuture = resolver.resolve(remoteAddress);
    final Throwable resolveFailureCause = resolveFuture.cause();

    if (resolveFailureCause != null) {
        // 如果地址解析失败，则立即失败
        channel.close();
        return channel.newFailedFuture(resolveFailureCause);
    }

    if (resolveFuture.isDone()) {
        // 理解成功的解析了远程地址，开始做连接
        return doConnect(resolveFuture.getNow(), localAddress, r
```



```
egFuture, channel.newPromise());
    }

    // 地址解析还没有完成, 只能等待完成后在做connectio, 增加一个promise
    来操作
    final ChannelPromise connectPromise = channel.newPromise();
    resolveFuture.addListener(new FutureListener<SocketAddress>(
    ) {
        @Override
        public void operationComplete(Future<SocketAddress> futu
re) throws Exception {
            if (future.cause() != null) {
                channel.close();
                connectPromise.setFailure(future.cause());
            } else {
                doConnect(future.getNow(), localAddress, regFutu
re, connectPromise);
            }
        }
    });

    return connectPromise;
}
```

doConnect()方法中才是真正的开始处理连接操作, 但是还是需要检查注册操作是否完成:

```
private static ChannelFuture doConnect(
    final SocketAddress remoteAddress, final SocketAddress l
ocalAddress,
    final ChannelFuture regFuture, final ChannelPromise conn
ectPromise) {
    // 判断一下前面的注册操作是否已经完成
    // 因为注册操作是异步操作，前面只是返回一个future，代码执行到这里时，
    // 可能完成，也可能还在进行中
    if (regFuture.isDone()) {
        // 如果注册已经完成，可以执行连接了
        doConnect0(remoteAddress, localAddress, regFuture, conn
ectPromise);
    } else {
        // 如果注册还在进行中，增加一个ChannelFutureListener，等操作完
        // 成之后再再在回调方法中执行连接操作
        regFuture.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future)
throws Exception {
                doConnect0(remoteAddress, localAddress, regFutur
e, connectPromise);
            }
        });
    }

    return connectPromise;
}
```

异步操作就是这点比较麻烦，总是需要一个一个future的做判断/处理，如果没有完成还的加promise/future来依靠回调函数继续工作处理流程。

终于到了最后的doConnect0()方法，总算可以真的连接了：

```

private static void doConnect0(
    final SocketAddress remoteAddress, final SocketAddress l
ocalAddress, final ChannelFuture regFuture,
    final ChannelPromise connectPromise) {
    // 这个方法在channelRegistered()方法被触发前调用。
    // 给我们的handler一个在它的channelRegistered()实现中构建pipeline
    的机会
    final Channel channel = connectPromise.channel();
    // 取当前channel的eventloop, 执行一个一次性任务
    channel.eventLoop().execute(new OneTimeTask() {
        @Override
        public void run() {
            if (regFuture.isSuccess()) {
                // 如果注册成功
                if (localAddress == null) {
                    channel.connect(remoteAddress, connectPromis
e);
                } else {
                    channel.connect(remoteAddress, localAddress,
connectPromise);
                }
                connectPromise.addListener(ChannelFutureListener
.CLOSE_ON_FAILURE);
            } else {
                connectPromise.setFailure(regFuture.cause());
            }
        }
    });
}

```

init(channel)方法

前面看基类AbstractBootstrap时看到过, 这个init()方法是一个模板方法, 需要子类做具体实现.

看看Bootstrap是怎么做channel初始化的:

```
@Override
@SuppressWarnings("unchecked")
void init(Channel channel) throws Exception {
    // 取channel的ChannelPipeline
    ChannelPipeline p = channel.pipeline();
    // 增加当前Bootstrap的handle到ChannelPipeline中
    p.addLast(handler());

    // 取当前Bootstrap设置的options, 逐个设置到channel中
    final Map<ChannelOption< ? >, Object> options = options();
    synchronized (options) {
        for (Entry<ChannelOption< ? >, Object> e: options.entrySet()) {
            try {
                if (!channel.config().setOption((ChannelOption<Object>) e.getKey(), e.getValue())) {
                    logger.warn("Unknown channel option: " + e);
                }
            } catch (Throwable t) {
                logger.warn("Failed to set a channel option: " +
                    channel, t);
            }
        }
    }

    // 同样取当前Bootstrap的attrs, 逐个设置到channel中
    final Map<AttributeKey< ? >, Object> attrs = attrs();
    synchronized (attrs) {
        for (Entry<AttributeKey< ? >, Object> e: attrs.entrySet()) {
            channel.attr((AttributeKey<Object>) e.getKey()).set(
                e.getValue());
        }
    }
}
```

总结上在init()方法中, Bootstrap只做了一个事情: 将**Bootstrap**中保存的信息 (handle/options/attrs)设置到新创建的**channel**.

clone()

深度克隆当前Bootstrap对象，有完全一样的配置，但是使用给定的EventLoopGroup。

这个方法适合用相同配置创建多个Channel。

```
public Bootstrap clone(EventLoopGroup group) {  
    Bootstrap bs = new Bootstrap(this);  
    bs.group = group;  
    return bs;  
}
```

类ServerBootstrap

类ServerBootstrap用于帮助服务器端引导ServerChannel.

ServerBootstrap除了处理ServerChannel外, 还需要处理从ServerChannel下创建的Channel.Netty中称这两个关系为parent和child.

类定义

```
public class ServerBootstrap extends AbstractBootstrap<ServerBootstrap, ServerChannel> {}
```

类属性

childGroup属性

childGroup属性用于指定处理客户端连接的EventLoopGroup, 设置的方式有两种:

1. group(parentGroup, childGroup)方法用于单独设置parentGroup, childGroup, 分别用于处理ServerChannel和Channel.
2. group(group)方法设置parentGroup, childGroup为使用同一个EventLoopGroup. 注意这个方法覆盖了基类的方法.

```
private volatile EventLoopGroup childGroup;

@Override
public ServerBootstrap group(EventLoopGroup group) {
    return group(group, group);
}
public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup) {
    super.group(parentGroup);
    if (childGroup == null) {
        throw new NullPointerException("childGroup");
    }
    if (this.childGroup != null) {
        throw new IllegalStateException("childGroup set already");
    }
    this.childGroup = childGroup;
    return this;
}
public EventLoopGroup childGroup() {
    return childGroup;
}
```

childOptions/childAttrs/childHandler属性

这三个属性和parent的基本对应, 设值方法和检验都是一模一样的:

```
private final Map<ChannelOption<?>, Object> childOptions = new LinkedHashMap<ChannelOption<?>, Object>();

private final Map<AttributeKey<?>, Object> childAttrs = new LinkedHashMap<AttributeKey<?>, Object>();

private volatile ChannelHandler childHandler;
```

类方法

init() 方法

ServerBootstrap的init(channel)方法相比Bootstrap的要复杂一些, 除了设置options/attrs/handler到channel外, 还需要为child设置childGroup, childHandler, childOptions, childAttrs:

```
@Override
void init(Channel channel) throws Exception {
    final Map<ChannelOption< ? >, Object> options = options();
    synchronized (options) {
        channel.config().setOptions(options);
    }

    final Map<AttributeKey< ? >, Object> attrs = attrs();
    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(childOptions.size()));
    }
    synchronized (childAttrs) {
        currentChildAttrs = childAttrs.entrySet().toArray(newAttrArray(childAttrs.size()));
    }
}
```



```

p.addLast(new ChannelInitializer<Channel>() {
    @Override
    public void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        ChannelHandler handler = handler();
        if (handler != null) {
            pipeline.addLast(handler);
        }
        pipeline.addLast(new ServerBootstrapAcceptor(
            currentChildGroup, currentChildHandler, curr
entChildOptions, currentChildAttrs));
    }
});
}

```

ServerBootstrapAcceptor的实现, 主要看channelRead()方法:

```

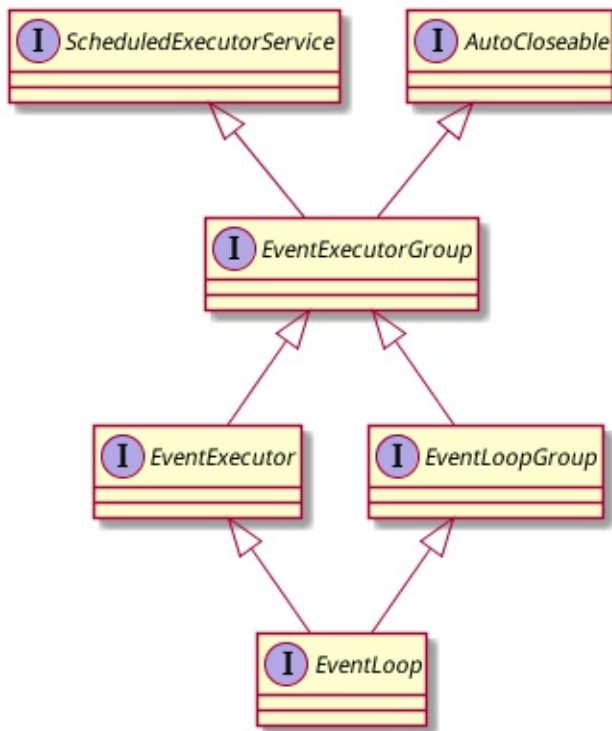
private static class ServerBootstrapAcceptor extends ChannelHand
lerAdapter {
    public void channelRead(ChannelHandlerContext ctx, Object ms
g) {
        // 获取child channel
        final Channel child = (Channel) msg;
        // 设置childHandler到child channel
        child.pipeline().addLast(childHandler);
        // 设置childOptions到child channel
        for (Entry<ChannelOption< ? >, Object> e: childOptions)
        {
            try {
                if (!child.config().setOption((ChannelOption<Obj
ect>) e.getKey(), e.getValue())) {
                    logger.warn("Unknown channel option: " + e);
                }
            } catch (Throwable t) {
                logger.warn("Failed to set a channel option: " +
child, t);
            }
        }
    }
}

```

```
// 设置childAttrs到child channel
for (Entry<AttributeKey< ? >, Object> e: childAttrs) {
    child.attr((AttributeKey<Object>) e.getKey()).set(e.
getValue());
}

// 将child channel注册到childGroup
try {
    childGroup.register(child).addListener(new ChannelFu
tureListener() {
        @Override
        public void operationComplete(ChannelFuture futu
re) throws Exception {
            if (!future.isSuccess()) {
                forceClose(child, future.cause());
            }
        }
    });
} catch (Throwable t) {
    forceClose(child, t);
}
}
```

EventLoop和EventLoopGroup这两个接口的定义有些复杂,不仅它们之间有令人疑惑的继承关系,而且它们还各自从EventExecutor和EventExecutorGroup中继承:



先分析EventExecutor和EventExecutorGroup,然后再继续看EventLoop和EventLoopGroup.

EventExecutorGroup

接口定义

先看EventExecutorGroup接口的定义,注意这个类在package "io.netty.util.concurrent"下:

```
package io.netty.util.concurrent

public interface EventExecutorGroup extends ScheduledExecutorService, AutoCloseable {}
```

EventExecutorGroup继承自jdk java.util.concurrent包下的ScheduledExecutorService, 这意味着EventExecutorGroup本身就是一个标准的jdk executor, 提供定时任务的支持.

增强shutdown()方法

EventExecutorGroup中针对ExecutorService的shutdown()提供了增强, 提供了优雅关闭的方法shutdownGracefully().

从代码上看,具体做法是EventExecutorGroup覆盖了executor的标准方法shutdown()和shutdownNow(), 加上了@Deprecated标记. 在javadoc中要求使用者不要调用这两个方法, 改为调用EventExecutorGroup接口中定义的shutdownGracefully()方法. 另外增加了一个isShuttingDown()方法来检查是否正在关闭:

```

/**
 * 当且仅当这个EventExecutorGroup管理的所有EventExecutor正在被shut
 * downGracefully()方法优雅关闭或被已经被关闭。
 */
boolean isShuttingDown();

/**
 * 等同于以合理的默认参数调用shutdownGracefully(quietPeriod, tim
 * eout, unit)方法
 */
Future<?> shutdownGracefully();

/**
 * 发信号给这个executor,告之调用者希望这个executor关闭。一旦这个方法
 * 被调用, isShuttingDown()方法就将开始返回true, 然后这个executor准备关闭
 * 自己。和shutdown()不同, 优雅关闭保证在关闭之前,在静默时间(the quiet per
 * iod, 通常是几秒钟)内没有任务提交。如果在静默时间内有任务提交, 这个任务将被
 * 接受, 而静默时间将重头开始。
 *
 * @param quietPeriod 上面文档中描述的静默时间
 * @param timeout 等待的最大超时时间, 直到executor被关闭, 无
 * 论在静默时间内是否有任务被提交
 * @param unit 静默时间和超时时间的单位
 *
 * @return the {@link #terminationFuture()}
 */
Future<?> shutdownGracefully(long quietPeriod, long timeout,
    TimeUnit unit);

Future<?> terminationFuture();

@Deprecated //禁用,用shutdownGracefully()代替
void shutdown();

@Deprecated //禁用,用shutdownGracefully()代替
List<Runnable> shutdownNow();

```

这里还有增加了一个terminationFuture()方法, 获取一个Future, 当这个EventExecutorGroup管理的所有的EventExecutors都被终止时可以得到通知。

管理EventExecutor

然后是最重要的两个方法next()和children():

```
/**
 * 返回这个EventExecutorGroup管理的一个EventExecutor
 */
EventExecutor next();

/**
 * 返回这个EventExecutorGroup管理的所有EventExecutor的不可变集合
 */
<E extends EventExecutor> Set<E> children();
```

这里可以看到EventExecutorGroup和EventExecutor的关系是:

- **EventExecutorGroup**管理了(从对象关系上说是"聚合"了)多个**EventExecutor**
- next()方法返回其中的一个EventExecutor
- children()返回所有的EventExecutor

覆盖submit()和schedule()方法

EventExecutorGroup中的submit()方法, 对照了一下, 和ExecutorService接口中完全相同的.

```
@Override
Future<?> submit(Runnable task);

@Override
<T> Future<T> submit(Runnable task, T result);

@Override
<T> Future<T> submit(Callable<T> task);
```

但是仔细看,会有个非常不起眼的小地方,submit()方法方法的返回对象不同,虽然都是Future:

- EventExecutorGroup中submit()方法返回的 Future 是 "io.netty.util.concurrent.Future"
- ExecutorService中submit()方法返回的 Future 是 "java.util.concurrent.Future"

下面是io.netty.util.concurrent.Future的代码, 继承自java.util.concurrent.Future, 然后增加了一些特有方法:

```
public interface Future<V> extends java.util.concurrent.Future<V> {  
    .....  
}
```

类似的, schedule()方法也是同样覆盖了接口ScheduledExecutorService中的对应方法, 依然是修改了返回的对象类型, 用io.netty.util.concurrent.ScheduledFuture替代了java.util.concurrent.ScheduledFuture:

```
public interface ScheduledFuture<V> extends Future<V>, java.util.concurrent.ScheduledFuture<V> {  
    .....  
}
```

EventExecutor

接口定义

EventExecutor的设计比较费解, 居然是从EventExecutorGroup下继承:

```
public interface EventExecutor extends EventExecutorGroup {  
}
```

考虑到EventExecutorGroup的设计中, EventExecutorGroup内部是聚合/管理了多个EventExecutor. 然后现在EventExecutor再继承EventExecutorGroup一把, 我有些凌乱了.....

和EventExecutorGroup的关系

在EventExecutor中覆盖了EventExecutorGroup的next()/children(), 在我看来这是netty在努力的收拾凌乱的局面:

```
@Override
EventExecutor next();

@Override
<E extends EventExecutor> Set<E> children();

EventExecutorGroup parent();
```

- next()方法: EventExecutorGroup中next()方法用来返回管理的EventExecutor中的其中一个, 到了EventExecutor中, 这里应该不会继续再管理其他EventExecutor了, 所以next()方法被覆盖为仅仅返回当前EventExecutor本身的引用
- children()方法: EventExecutorGroup中children()方法用来返回管理的所有的EventExecutor, 到了EventExecutor中, 返回的集合就只能包含自身一个引用.
- parent()方法: 新增的方法, 返回当前EventExecutor所属的EventExecutorGroup

inEventLoop()方法

inEventLoop()方法用于查询某个线程是否在EventExecutor所管理的线程中.

```
/**
 * 等同于调用inEventLoop(Thread.currentThread())
 */
boolean inEventLoop();

/**
 * 当给定的线程在event loop中返回true, 否则返回false
 */
boolean inEventLoop(Thread thread);
```


unwrap()方法

unwrap()方法用于返回一个非WrappedEventExecutor的EventExecutor.

- 对于WrappedEventExecutor的实现类,需要返回底层包裹的EventExecutor, 而且要保证返回的EventExecutor也不是WrappedEventExecutor (这意味着可能需要调用多次unwrap())
- 对于非WrappedEventExecutor的实现类, 只要简单返回自身实例就好了
- 强调, 一定不能返回null

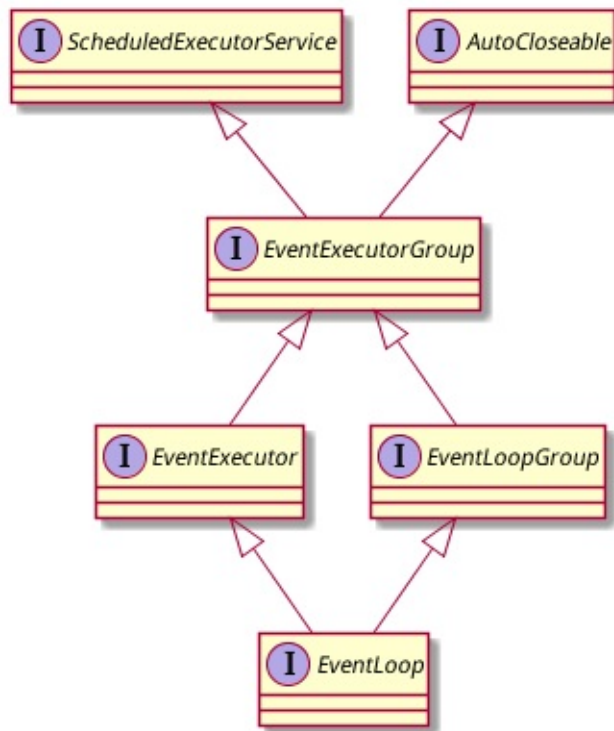
```
EventExecutor unwrap();
```

创建Promise和Future

```
<V> Promise<V> newPromise();  
  
<V> ProgressivePromise<V> newProgressivePromise();  
  
<V> Future<V> newSucceededFuture(V result);  
  
<V> Future<V> newFailedFuture(Throwable cause);
```

这个细节后面研究Promise和Future时再先看, 先跳过.

EventLoopGroup/EventLoop的关系和EventExecutorGroup/EventExecutor的关系非常类似, 都是费解的聚合+继承, 另外EventLoopGroup/EventLoop分别继承自EventExecutorGroup/EventExecutor.



EventLoopGroup

EventLoopGroup是一个特殊的容许注册Channel的EventExecutorGroup.

接口定义

注意package 移到io.netty.channel了:

```
package io.netty.channel;

public interface EventLoopGroup extends EventExecutorGroup {
}
```

覆盖EventExecutorGroup的方法

```
@Override
EventLoop next();    // 返回类型从EventExecutor变成了EventLoop
```

register() 方法

这个是EventLoopGroup的核心, 用来注册Channel.

```
/**
 * 将Channel注册到EventLoopGroup中的一个EventLoop.
 * 当注册完成时, 返回的ChannelFuture会得到通知.
 */
ChannelFuture register(Channel channel);

/**
 * 将Channel注册到EventLoopGroup中的一个EventLoop,
 * 当注册完成时ChannelPromise会得到通知, 而返回的ChannelFuture就是
 传入的ChannelPromise.
 */
ChannelFuture register(Channel channel, ChannelPromise promise);
```

javadoc中有一个注意的提醒: 只有当ChannelPromise成功后, 从ChannelHandler提交新任务到EventLoop中才是安全的, 否则这个任务可能被拒绝. 这个好理解, 只有Channel注册成功之后, 才能开始提交任务.

EventLoop

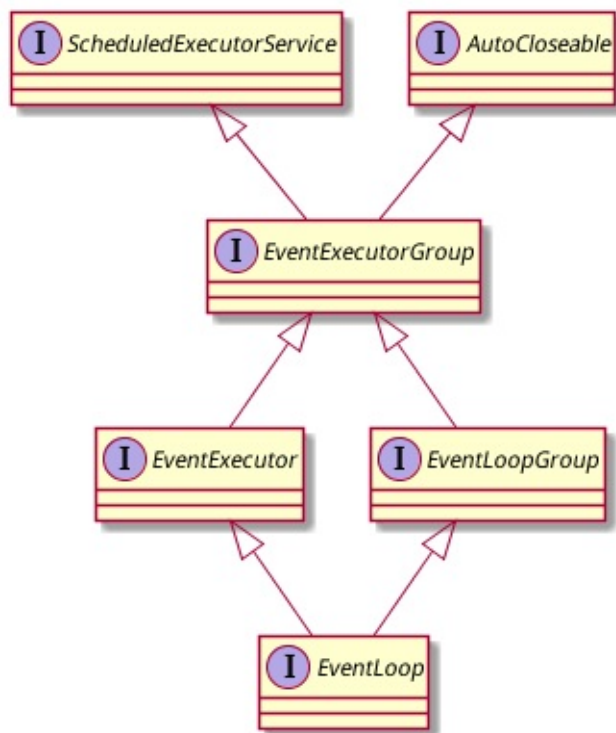
接口定义

EventLoop的接口定义如下, 继承自EventExecutor/EventLoopGroup:

```
public interface EventLoop extends EventExecutor, EventLoopGroup
{
}
```

坦白说第一次看EventLoop的接口定义时,看懵了,完全不能理解netty为什么要这么设计:)

现在分析完成EventExecutor/EventExecutorGroup的关系,再对照下面这个图,总算能理解了.



覆盖基类方法

parent()方法和unwrap()被覆盖,依然开始返回类型被修订为符合EventLoop的具体类型:

```
@Override
EventLoopGroup parent();           // 返回所属的EventLoopGroup

@Override
EventLoop unwrap();               // unwrap出来的当然时EventLoop
```

asInvoker()方法

不是太理解这个方法用来干嘛的,后续继续看.

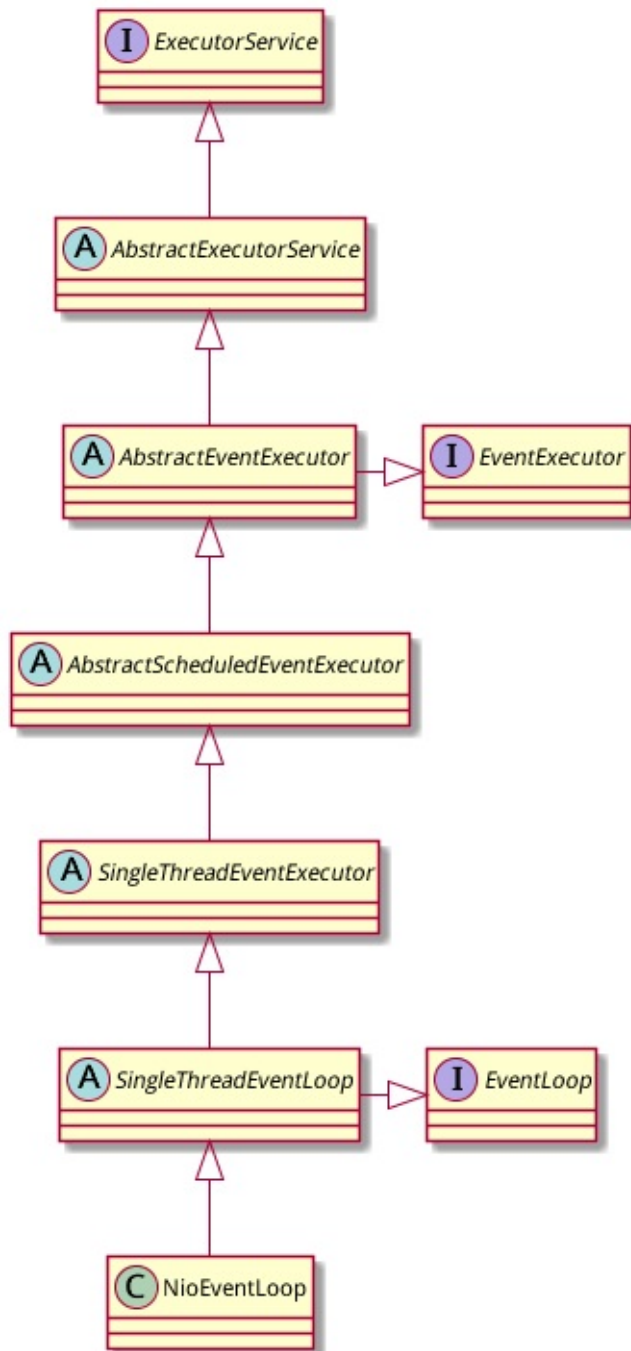
```
/**
 * 创建一个新的默认 ChannelHandlerInvoker 实现，使用当前EventLoop来调用事件处理方法。
 */
ChannelHandlerInvoker asInvoker();
```

额外说明

EventLoop最重要的方法是register()方法, 但是注意这个方法是在EventLoopGroup中定义.

NioEventLoop是一个SingleThreadEventLoop的实现, 将每个Channel注册到NIO Selector并执行multiplexing.

NioEventLoop的类继承结构



NioEventLoop的类继承结构比较深, 继承结构中比较关键的是:

- AbstractScheduledEventExecutor: netty的EventExecutor, 提供对定时任务的支持
- SingleThreadEventExecutor: 在单个线程中执行所有提交的任务

- SingleThreadEventLoop: 单线程版本的EventLoop

这里可以得到一个重要的信息: **NioEventLoop**时用单线程来处理**NIO**任务的!

在这个多层的继承结构中:

1. ExecutorService/AbstractExecutorService 是属于JDK的代码, 著名的JDK Executor, package是"java.util.concurrent"
2. AbstractEventExecutor/AbstractScheduledEventExecutor/SingleThreadEventExecutor 是属于netty util的通用类, package 是 "io.netty.util.concurrent"
3. SingleThreadEventLoop 是属于netty的通用的event loop实现基类, package 是"io.netty.channel"
4. NioEventLoop是属于netty的基于NIO的event loop实现类, package 是"io.netty.channel.nio"

后面我们从上到下来研究具体的代码实现.

在NioEventLoop的类继承机构中,位于最上端的是ExecutorService/AbstractExecutorService,这是属于JDK的代码,著名的JDK Executor框架.

在研究netty的NioEventLoop之前,先重温一下JDK Executor的基础代码.

Executor

大名鼎鼎的JDK Executor, 它的接口定义其实非常简单,就一个execute()方法:

```
package java.util.concurrent;

public interface Executor {
    void execute(Runnable command);
}
```

传入的Runnable对象代表需要执行的命令,但是注意Executor接口并没有定义这个命令的执行方式,因此这个命令有可能被多种方式执行:

- 最简单的同步方式,直接被调用这个execute()方法的线程执行
- 异步方式,启动一个新的线程来执行这个命令
- 带线程池的异步方式,从线程池中取出一个空闲线程来执行这个命令,执行完毕之后归还线程到线程池
- 线程池的实现可能有多种,比如只有单个工作线程和有多个工作线程
- 支持定时任务的实现,可以在内部保存要求执行的命令,等到任务执行条件满足后再执行命令

而这些具体执行方式是交给Executor的实现类来实现,对于调用者只需要选择调用不同的实现类即可轻松实现在多种方式中间选择和切换,甚至可以不关心具体到底是用什么实现类,直接针对Executor接口编程.

这样就轻松的将"任务的执行内容"(比如删除一条记录)和"任务的执行方式"(同步/异步/用线程池/5分钟后再执行)在代码上实现隔离和解耦.

ExecutorService

Executor接口只定义了一个简单的execute()方法,而ExecutorService在Executor的基础上做了扩展:

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit) throws
        InterruptedException;

    <T> Future<T> submit(Callable<T> task);
    <T> Future<T> submit(Runnable task, T result);
    Future<?> submit(Runnable task);

    <T> List<Future<T>> invokeAll(Collection<? extends Callable<
T>> tasks) throws InterruptedException;
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<
T>> tasks, long timeout, TimeUnit unit) throws InterruptedExcept
ion;
    <T> T invokeAny(Collection<? extends Callable<T>> tasks) thr
ows InterruptedException, ExecutionException;
    <T> T invokeAny(Collection<? extends Callable<T>> tasks, long
    timeout, TimeUnit unit) throws InterruptedException, ExecutionE
xception, TimeoutException;
}
```

ExecutorService定义的方法包括:

1. 和关闭相关的方法 包括 shutdown()/shutdownNow()/isShutdown()/isTerminated()/awaitTermination().
2. 提交任务的方法 包括3个submit()方法, 提供对Future的支持 (注意execute()方法返回void)
3. 执行任务的方法 包括invokeAll()和invokeAny()

在这里我们跳过shutdown和invoke的方法, 重点看任务执行的方法 submit()/execute().

注意这里出现了两个接口用来表示任务或者命令,除了之前Executor中用到的Runnable之外,ExecutorService还支持Callable接口:

- Runnable接口

Runnable接口只定义了一个run()方法,注意它的返回值是void.

```
public interface Runnable {  
    public abstract void run();  
}
```

- Callable接口

Callable接口和Runnable接口最大的不同在于Callable的call()可以返回一个结果,另外容许定义抛出受查异常:

```
public interface Callable<V> {  
    V call() throws Exception;  
}
```

ExecutorService中定义的三个submit()方法,都支持返回结果,而且是通过Future可以实现异步不阻塞.

1. Future submit(Callable task) 用Callable接口来提交任务,返回的类型是Future,支持泛型和Future
2. Future submit(Runnable task, T result) 用Runnable接口来提交任务,由于Runnable接口无法表示返回类型,因此多增加一个result参数.
3. Future< ? > submit(Runnable task) 用Runnable接口来提交任务,无返回值.

继续看AbstractExecutorService的代码,看这三个submit()方法具体是如何实现的.

AbstractExecutorService

AbstractExecutorService的类定义,申明实现ExecutorService接口:

```
public abstract class AbstractExecutorService implements ExecutorService {}
```

submit(Callable)代码实现

先看submit(Callable)方法的实现:

```
public <T> Future<T> submit(Callable<T> task) {  
    if (task == null) throw new NullPointerException();  
    RunnableFuture<T> ftask = newTaskFor(task);  
    execute(ftask);  
    return ftask;  
}
```

这里去掉检查task为null的fail fast代码之外, 剩下的三行代码做了三件事情:

1. newTaskFor()方法将Callable类型的task包装为RunnableFuture
2. 调用execute(ftask)方法执行任务
3. 返回任务执行的结果

newTaskFor()方法

newTaskFor()方法的代码如下, 里面出现了FutureTask类和RunnableFuture接口:

```

protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable)
{
    // 将Callable类型的task包装为FutureTask, 然后向上溯型为接口RunnableFuture
    return new FutureTask<T>(callable);
}

public class FutureTask<V> implements RunnableFuture<V> {
    // 其他细节处理代码去掉, run()方法的核心代码就是调用Callable.call()
    // 然后将得到的result保存起来
    public void run() {
        .....
        Callable<V> c = callable;
        V result;
        result = c.call();
        set(result)
        .....
    }
}
// RunnableFuture 是实现了Runnable的Future (废话?!)
public interface RunnableFuture<V> extends Runnable, Future<V> {
    void run();
}

```

总结,这行代码:

```
RunnableFuture<T> ftask = newTaskFor(task);
```

做的事情就是将task包装为RunnableFuture

RunnableFuture的实际实现类是默认是FutureTask, 但是注意newTaskFor()方法是protected, 因此具体的Executor实现类是可以覆盖这个方法来换成其他RunnableFuture实现.

execute(task)方法

execute(ftask)调用回Executor接口的execute()方法, 注意RunnableFuture的作用.

```
execute(ftask);

public interface Executor {
    void execute(Runnable command);
}
```

submit(Callable)方法总结

这就是ExecutorService中定义的submit(Callable)方法的实现方式: 通过将Callable包装为RunnableFuture, 成功的将submit(Callable)方法代理给Executor接口标准的execute(Runnable)方法, 同时提供了Future的支持.

另外两个submit()方法(submit(Runnable task, T result) 方法和submit(Runnable task)方法) 的实现类似, 只是newTaskFor()方法的处理细节小有区别, 不细看了.

AbstractEventExecutor类的类定义, 继承自AbstractExecutorService, 另外申明实现EventExecutor接口.

```
public abstract class AbstractEventExecutor extends AbstractExecutorService implements EventExecutor {  
  
}
```

我们将AbstractEventExecutor类的代码分成两部分来看.

EventExecutor接口实现的代码

下面这些代码是实现EventExecutor接口的基本代码, 没有特别之处.

```
private final EventExecutorGroup parent;

protected AbstractEventExecutor() {
    this(null);          // 为什么要容许parent为null?
}

protected AbstractEventExecutor(EventExecutorGroup parent) {
    this.parent = parent;
}

@Override
public EventExecutorGroup parent() {
    return parent;
}

@Override
public EventExecutor next() {
    // EventExecutor的next()方法需要返回自身引用
    return this;
}

@Override
@SuppressWarnings("unchecked")
public <E extends EventExecutor> Set<E> children() {
    // EventExecutor的children()方法需要的集合只能包含自身的引用
    return Collections.singleton((E) this);
}

..... // 其他代码忽略
```

ExecutorService接口实现的代码

覆盖submit()方法

对ExecutorService接口的实现才是关键代码, 先看三个submit()方法:

```
@Override
public Future<?> submit(Runnable task) {
    return (Future<?>) super.submit(task);
}

@Override
public <T> Future<T> submit(Runnable task, T result) {
    return (Future<T>) super.submit(task, result);
}

@Override
public <T> Future<T> submit(Callable<T> task) {
    return (Future<T>) super.submit(task);
}
```

这里只做了一件事情,将返回的Future类型从java.util.concurrent.Future覆盖成了netty自己的io.netty.util.concurrent.Future. (注意:io.netty.util.concurrent.Future是extends java.util.concurrent.Future的)

覆盖newTaskFor()方法

前面在分析AbstractExecutorService的代码时就说newTaskFor()设计成protected就是为了让子类覆盖的,果然在AbstractEventExecutor中被覆盖了:

```
@Override
protected final <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {
    return new PromiseTask<T>(this, runnable, value);
}

@Override
protected final <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
    return new PromiseTask<T>(this, callable);
}
```

RunnableFuture的实现类从JDK的FutureTask换成了netty自己的PromiseTask.

PromiseTask我们现在不展开,只是简单看一下类的定义.

```
class PromiseTask<V> extends DefaultPromise<V> implements RunnableFuture<V> {}
```

schedule()方法不被支持

四个schedule()方法都直接抛出UnsupportedOperationException.

注: 在稍后的AbstractScheduledEventExecutor中, 这几个schedule()方法都将被覆盖为可工作的版本.

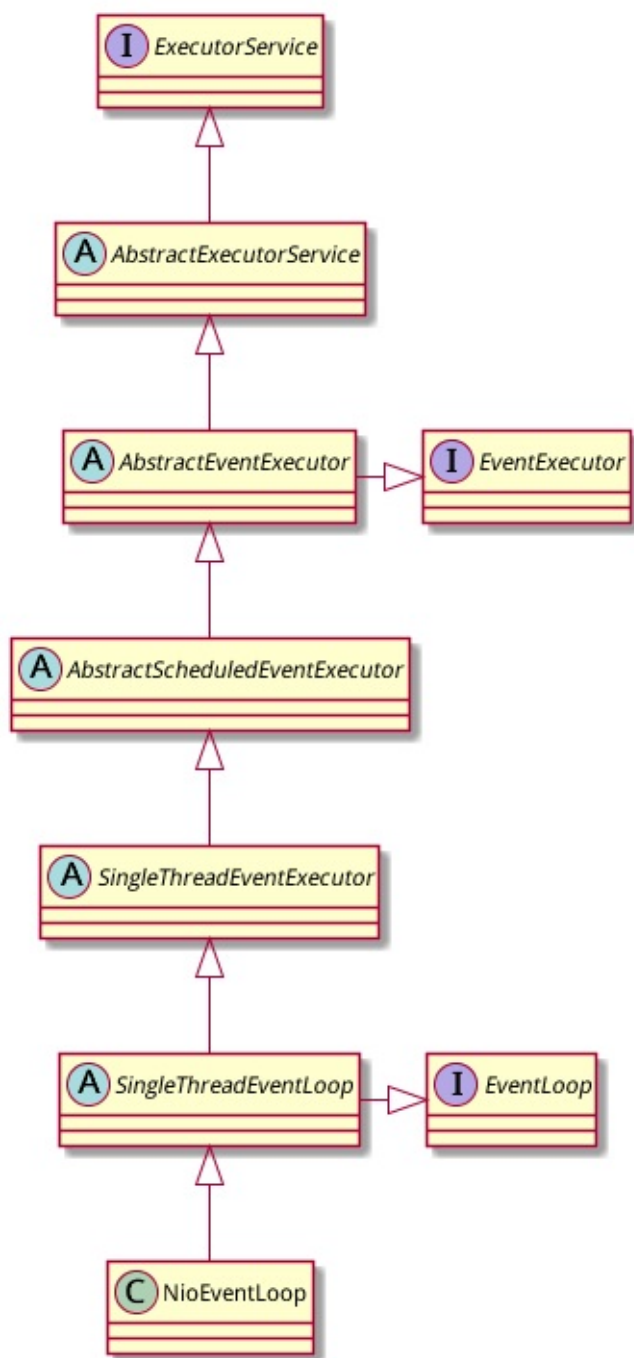
```
@Override
public ScheduledFuture<?> schedule(Runnable command, long delay,
                                   TimeUnit unit) {
    throw new UnsupportedOperationException();
}

@Override
public <V> ScheduledFuture<V> schedule(Callable<V> callable, long
    delay, TimeUnit unit) {
    throw new UnsupportedOperationException();
}

@Override
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
    long initialDelay, long period, TimeUnit unit) {
    throw new UnsupportedOperationException();
}

@Override
public ScheduledFuture< ? > scheduleWithFixedDelay(Runnable comm
and, long initialDelay, long delay, TimeUnit unit) {
    throw new UnsupportedOperationException();
}
```

总结



其实从类继承结构上可以很清楚的看到AbstractEventExecutor的特别之处:

- 从jdk通用的Executor框架变成了netty的EventExecutor: 这里开始引入netty的东西比如Future/PromiseTask就理所当然了
- 增加了EventExecutor的支持, 自然EventExecutor里面定义的一些基本方法就可以在这里实现

AbstractScheduledEventExecutor在AbstractEventExecutor的基础上提供了对定时任务的支持, 在内部有一个queue用于保存定时任务.

```
public abstract class AbstractScheduledEventExecutor extends AbstractEventExecutor {

    Queue<ScheduledFutureTask<?>> scheduledTaskQueue;

    Queue<ScheduledFutureTask<?>> scheduledTaskQueue() {
        if (scheduledTaskQueue == null) {
            scheduledTaskQueue = new PriorityQueue<ScheduledFutureTask<?>>();
        }
        return scheduledTaskQueue;
    }
}
```

在queue中存放的是ScheduledFutureTask的实例. 在深入研究AbstractScheduledEventExecutor的代码前, 我们先看一下ScheduledFutureTask.

ScheduledFutureTask

ScheduledFutureTask继承自PromiseTask, 并实现了ScheduledFuture接口. 而ScheduledFuture接口是从java.util.concurrent.ScheduledFuture继承, 再往上是Delayed接口, 这里有对于定时任务而言最重要的一个方法getDelay().

```
final class ScheduledFutureTask<V> extends PromiseTask<V> implements ScheduledFuture<V> {
    @Override
    public long getDelay(TimeUnit unit) {
        return unit.convert(delayNanos(), TimeUnit.NANOSECONDS);
    }
}

public interface ScheduledFuture<V> extends Future<V>, java.util.concurrent.ScheduledFuture<V> {
}

public interface ScheduledFuture<V> extends Delayed, Future<V> {
}

public interface Delayed extends Comparable<Delayed> {
    long getDelay(TimeUnit unit);
}
```

getDelay()方法用来返回当前这个对象离执行的延迟值,即还要过多长时间就要执行了(所谓定时任务).

ScheduledFutureTask.getDelay()方法的实现

ScheduledFutureTask的getDelay()方法中是通过调用delayNanos()方法得到时间结果,然后转换为指定的时间单位.而delayNanos()方法是通过计算deadlineNanos()和nanoTime()的差值来计算delay时间的.

- deadlineNanos()表示任务计划要执行的时间点
- nanoTime()是取当前时间点

两个时间的差值自然是所谓的delay时间.

```
public long delayNanos() {
    return Math.max(0, deadlineNanos() - nanoTime());
}
```

1. deadlineNanos()方法

`deadlineNanos()`方法返回的`deadlineNanos`属性在构造函数中被指定,即在任务构造的时候指定任务计划执行的时间点,这对于定时任务自然是合理的:

```
private long deadlineNanos;
ScheduledFutureTask(EventExecutor executor, Callable<V> callable, long nanoTime, long period) {
    .....
    deadlineNanos = nanoTime;
}
public long deadlineNanos() {
    return deadlineNanos;
}
```

2. `nanoTime()`方法

`nanoTime()`方法并没有直接调用`System.nanoTime()`,而是先通过常量`START_TIME`中保存着类装载时的时间值,然后`nanoTime()`方法中用每次取当前时间减去这个`START_TIME`,返回其差值.

```
private static final long START_TIME = System.nanoTime();
static long nanoTime() {
    return System.nanoTime() - START_TIME;
}
```

`ScheduledFutureTask.run()`方法的实现

再看`ScheduledFutureTask`的`run()`方法,去掉其他细节处理之后的简化代码如下:

```
public void run() {
    .....
    if (setUncancellableInternal()) {
        V result = task.call();
        setSuccessInternal(result);
    }
    .....
}
```

调用`task.call()`方法来执行`task`, 在执行前注明不能`cancel`, 执行完成后设置执行成功.

看完`ScheduledFutureTask`之后, 我们回到类`AbstractScheduledEventExecutor`.

添加定时任务

前面看到`AbstractScheduledEventExecutor`中有一个`scheduledTaskQueue`用来保存定时任务`ScheduledFutureTask`, 我们来看这些定时任务时如何添加的.

先看`AbstractScheduledEventExecutor`的`schedule()`方法, 在`AbstractEventExecutor`中这几个`schedule()`方法都是简单的抛出`UnsupportedOperationException`, 现在都被实现:

```
@Override
public ScheduledFuture< ? > schedule(Runnable command, long
delay, TimeUnit unit) {
    .....
    return schedule(new ScheduledFutureTask<Void>(
        this, toCallable(command), ScheduledFutureTask.d
eadlineNanos(unit.toNanos(delay))));
}
```

这几个`schedule()`方法都是类似的, 利用输入的参数先生成`ScheduledFutureTask`对象的实例, 然后调用下面这个`schedule(ScheduledFutureTask)`方法:

```
<V> ScheduledFuture<V> schedule(final ScheduledFutureTask<V> task) {
    if (inEventLoop()) {
        scheduledTaskQueue().add(task);
    } else {
        execute(new OneTimeTask() {
            @Override
            public void run() {
                scheduledTaskQueue().add(task);
            }
        });
    }

    return task;
}
```

先通过`inEventLoop()`判断, 如果当前调用的线程是外部线程, 则生成一个`OneTimeTask`, 然后调用`execute()`方法, 之后`event loop`的工作线程会执行这个`OneTimeTask`将`task`加入到`scheduledTaskQueue`; 如果调用的线程就是`event loop`的工作线程, 则直接`enqueue`.

注: `scheduleAtFixedRate()`方法我们先掉过.

获取定时任务

`pollScheduledTask()`方法用来获取到期的定时任务:


```

protected final Runnable pollScheduledTask() {
    return pollScheduledTask(nanoTime());
}

protected final Runnable pollScheduledTask(long nanoTime) {
    assert inEventLoop();

    Queue < ScheduledFutureTask < ? > > scheduledTaskQueue =
this.scheduledTaskQueue;
    ScheduledFutureTask< ? > scheduledTask = scheduledTaskQueue == null ? null : scheduledTaskQueue.peek();
    if (scheduledTask == null) {
        // 任务队列中没有任务
        return null;
    }

    if (scheduledTask.deadlineNanos() <= nanoTime) {
        // 取到的任务满足deadline条件, 可以返回
        // 在返回前将任务从queue中删掉, 注意前面取任务用的时peek()
        scheduledTaskQueue.remove();
        return scheduledTask;
    }

    // 取到的任务不满足deadline条件, 不能返回, 只能返回null表示没有满足条件的任务
    return null;
}

```

方法

hasScheduledTasks()用法用来判断是否有满足条件的定时任务:

```
protected final boolean hasScheduledTasks() {
    Queue<ScheduledFutureTask<?>> scheduledTaskQueue = this.
scheduledTaskQueue;
    ScheduledFutureTask<?> scheduledTask = scheduledTaskQueue
e == null ? null : scheduledTaskQueue.peek();
    return scheduledTask != null && scheduledTask.deadlineNanos() <= nanoTime();
}
```

另外当没有定时任务时, 还有一个nextScheduledTaskNano()方法用来查询下一个可以满足条件的任务的定时时间:

```
protected final long nextScheduledTaskNano() {
    Queue<ScheduledFutureTask< ? >> scheduledTaskQueue = this
.scheduledTaskQueue;
    ScheduledFutureTask< ? > scheduledTask = scheduledTaskQueue
e == null ? null : scheduledTaskQueue.peek();
    if (scheduledTask == null) {
        // 没有任务返回-1
        return -1;
    }
    return Math.max(0, scheduledTask.deadlineNanos() - nanoTime());
}
```

这几个方法明显是留着给基类调用的了.

SingleThreadEventExecutor的类定义:

```
public abstract class SingleThreadEventExecutor extends AbstractScheduledEventExecutor {  
}
```

在SingleThreadEventExecutor中,存在两个queue:

1. taskQueue: 这个是可以执行的任务队列, SingleThreadEventExecutor自己实现
2. scheduledTaskQueue: 定时任务队列, 从AbstractScheduledEventExecutor中继承

taskQueue的实现

taskQueue的定义和初始化

taskQueue构造函数中通过调用newTaskQueue()方法来初始化, 默认是用LinkedBlockingQueue:

```
private final Queue<Runnable> taskQueue;  
  
protected SingleThreadEventExecutor(EventExecutorGroup parent, Executor executor, boolean addTaskWakesUp) {  
    .....  
    taskQueue = newTaskQueue();  
}  
protected Queue<Runnable> newTaskQueue() {  
    return new LinkedBlockingQueue<Runnable>();  
}
```

注意newTaskQueue()方法是protected, 依然给子类做覆盖预留了空间.

pollTask()方法

pollTask()方法用于从taskQueue中获取任务:

```
protected Runnable pollTask() {
    assert inEventLoop();
    for (;;) {
        Runnable task = taskQueue.poll();
        if (task == WAKEUP_TASK) {
            continue;
        }
        return task;
    }
}
```

WAKEUP_TASK先忽略, 后面再细看.

takeTask()方法

takeTask()方法用来获取需要任务, 注意这个时候任务有两个来源: taskQueue和scheduledTask. 因此取任务时需要考虑很多情况:

```
protected Runnable takeTask() {
    // 断言当前调用线程是event loop的工作线程, 自我保护之一
    assert inEventLoop();
    // 检查当前taskQueue, 如果不是BlockingQueue抛异常退出
    if (!(taskQueue instanceof BlockingQueue)) {
        throw new UnsupportedOperationException();
    }

    BlockingQueue<Runnable> taskQueue = (BlockingQueue<Runnable>) this.taskQueue;
    for (;;) {
        // 从scheduledTaskQueue取一个scheduledTask, 注意方法是peek,
        // 另外这个task有可能还没有到执行时间
        ScheduledFutureTask<?> scheduledTask = peekScheduledTask();
        if (scheduledTask == null) {
            // scheduledTask为null, 说明当前scheduledTaskQueue中没有定时任务

            // 这个时候不用考虑定时任务, 直接从taskQueue中拿任务即可
            Runnable task = null;
        }
    }
}
```

```

        try {
            // 从taskQueue取任务, 如果没有任务会被阻塞, 直到取到任
            务, 或者被InterruptedException
            task = taskQueue.take();
            // 检查如果是WAKEUP_TASK, 则需要跳过
            if (task == WAKEUP_TASK) {
                task = null;
            }
        } catch (InterruptedException e) {
            // Ignore
        }

        // 如果从taskQueue中拿任务成功, 则返回, 如果没有任务或者没有
        成功, 则返回的是null
        return task;
    } else {
        // scheduledTask不为null, 说明当前scheduledTaskQueue中
        有定时任务
        // 但是这个定时任务可能还没有到执行时间, 因此需要检查delayNa
        nos

        long delayNanos = scheduledTask.delayNanos();
        Runnable task = null;
        if (delayNanos > 0) {
            // delayNanos > 0 说明这个定时任务没有到执行时间, 所以
            这个定时任务是不能用的了
            // 那就需要从taskQueue里面取任务了
            try {
                // 从taskQueue取任务, 如果没有任务则最多等待delay
                Nanos时间, 注意这里线程会被阻塞
                // 如果delayNanos时间内有任务加入到taskQueue, 则
                可以实时取到这个任务
                // 线程结束阻塞继续执行, 这个task就可以返回了
                // 如果delayNanos时间内一直没有任务, 则timeout后
                线程结束阻塞, poll()方法返回null
                task = taskQueue.poll(delayNanos, TimeUnit.N
                ANOSECONDS);
            } catch (InterruptedException e) {
                // 还有一种可能就是delayNanos时间内被waken up
                // 比如有新的scheduledTask加入, delay时间小于前
                面的delayNanos

```

```
        // 因此不能等待delayNanos timeout,需要提前结束阻塞

        // Waken up.
        return null;
    }
}
if (task == null) {
    // We need to fetch the scheduled tasks now as o
    therwise there may be a chance that
    // scheduled tasks are never executed if there i
    s always one task in the taskQueue.
    // This is for example true for the read task of
    OIO Transport
    // See https://github.com/netty/netty/issues/1614

    // 继续尝试从scheduledTaskQueue取满足条件的task到tas
    kQueue中

    fetchFromScheduledTaskQueue();
    // 然后再从taskQueue获取试试
    task = taskQueue.poll();
}

if (task != null) {
    // 只有task不为空时才返回并退出, 否则前面的for循环就一直

    return task;
}
}
}
```

任务的执行

execute()方法被覆盖为:

```
@Override
public void execute(Runnable task) {
    if (task == null) {
        throw new NullPointerException("task");
    }

    boolean inEventLoop = inEventLoop();
    if (inEventLoop) {
        // 如果当前线程是event loop线程, 则将task加入到taskQueue中
        addTask(task);
    } else {
        // 如果是外部线程
        // 调用startExecution()方法先
        startExecution();
        // 再将task加入到taskQueue中
        addTask(task);
        if (isShutdown() && removeTask(task)) {
            reject();
        }
    }

    if (!addTaskWakesUp && wakesUpForTask(task)) {
        wakeup(inEventLoop);
    }
}
```

实际上execute()方法只是将任务加入到taskQueue中, 任务真正的执行还不在这里.

wake up机制

在SingleThreadEventExecutor中, 设计了一套wake up机制, 包含以下内容:

WAKEUP_TASK任务

定了一个什么都不做的task, 这是一个特殊任务, 用来做wake up的标记

```
private static final Runnable WAKEUP_TASK = new Runnable() {  
    @Override  
    public void run() {  
        // Do nothing.  
    }  
};
```

比如在pollTask()方法中,遇到WAKEUP_TASK就会自动跳过

```
protected Runnable pollTask() {  
    assert inEventLoop();  
    for (;;) {  
        Runnable task = taskQueue.poll();  
        if (task == WAKEUP_TASK) {  
            continue;  
        }  
        return task;  
    }  
}
```

在takeTask()方法中,也会检查taskQueue取出的task会不会是WAKEUP_TASK:


```

protected Runnable takeTask() {
    .....
    for (;;) {
        ScheduledFutureTask< ? > scheduledTask = peekScheduledTask();
        if (scheduledTask == null) {
            Runnable task = null;
            try {
                task = taskQueue.take();
                if (task == WAKEUP_TASK) {
                    // 如果是WAKEUP_TASK, 则设置task为null
                    task = null;
                }
            } catch (InterruptedException e) {
                // Ignore
            }
            // 如果是WAKEUP_TASK, 则返回的是null
            return task;
        } else {
            .....
        }
    }
}

```

添加WAKEUP_TASK任务

WAKEUP_TASK这个特殊任务是在wakeup()方法中被添加, wakeup()方法中还需要检查其他条件:

1. inEventLoop==false, 即调用wakeup()的方法应该是外部线程
2. 如果inEventLoop==true, 则调用wakeup()的方法的时event loop线程本身, 则只有当状态为ST_SHUTTING_DOWN即当前Executor正在关闭中时

```
protected void wakeup(boolean inEventLoop) {  
    if (!inEventLoop || STATE_UPDATER.get(this) == ST_SHUTTING_D  
OWN) {  
        taskQueue.add(WAKEUP_TASK);  
    }  
}
```

wakeup()方法在几个shutdown相关的方法中被调用,比如 shutdown()/shutdownGracefully()/confirmShutdown()方法中调用. 但是这里我们不是太关心, 我们看最重要的调用的地方, 在execute()方法中:

```
public void execute(Runnable task) {  
    .....  
    addTask(task);  
    .....  
  
    if (!addTaskWakesUp && wakesUpForTask(task)) {  
        wakeup(inEventLoop);  
    }  
}  
  
protected boolean wakesUpForTask(Runnable task) {  
    return true;  
}
```

在将task添加到taskQueue中之后, 会做一个检查: 如果addTaskWakesUp是false 并且wakesUpForTask(task)也返回true, 则调用wakeup(inEventLoop), 里面还有一个检测是inEventLoop是false, 这样才真正的会向taskQueue中添加一个WAKEUP_TASK.

wakesUpForTask(task)方法默认简单返回true, 这是个protected方法, 子类可以按照自己的需要覆盖.

实际在SingleThreadEventLoop中, 增加了一个标志接口NonWakeupRunnable, 然后wakesUpForTask()被覆盖为检测是否task是否是NonWakeupRunnable.

```
public abstract class SingleThreadEventLoop extends SingleThread
EventExecutor implements EventLoop {
    protected boolean wakesUpForTask(Runnable task) {
        return !(task instanceof NonWakeupRunnable);
    }

    interface NonWakeupRunnable extends Runnable {}
}
```

总结说, 在execute(task)方法中, 要最终向taskQueue中添加一个WAKEUP_TASK做wake up, 需要满足以下三个条件:

1. addTaskWakesUp==false: 证实在NioEventLoop的实现中, addTaskWakesUp是设置为false的, 因此这个条件在NioEventLoop总是被满足
2. wakesUpForTask(task)返回true: 只要task不要被标记为NonWakeupRunnable就可以
3. inEventLoop==false: 即只能是外部线程调用execute(task)方法

addTaskWakesUp属性

addTaskWakesUp属性在构造函数中设置, 定义为final设值后不可修改:

```
private final boolean addTaskWakesUp;
protected SingleThreadEventExecutor(EventExecutorGroup parent, E
xecutor executor, boolean addTaskWakesUp) {
    this.addTaskWakesUp = addTaskWakesUp;
}
```

经检查,NioEventLoop的实现中, addTaskWakesUp是设置为false的.

wake up总结

- 当外部线程调用schedule()方法添加定时任务时, 定时任务会存放在scheduledTaskQueue中
- 用于保存任务的taskQueue是BlockingQueue, 所以当taskQueue中没有任务时, event loop线程会被阻塞

TBD: 看糊涂了.....

SingleThreadEventLoop的类定义

```
public abstract class SingleThreadEventLoop extends SingleThread
EventExecutor implements EventLoop {}
```

register()方法

register()方法最终调用到unsafe做register.

```
@Override
public ChannelFuture register(Channel channel) {
    return register(channel, new DefaultChannelPromise(channel,
this));
}

@Override
public ChannelFuture register(final Channel channel, final Chan
elPromise promise) {
    if (channel == null) {
        throw new NullPointerException("channel");
    }
    if (promise == null) {
        throw new NullPointerException("promise");
    }

    channel.unsafe().register(this, promise);
    return promise;
}
```

标志接口NonWakeupRunnable

在SingleThreadEventLoop中,增加了一个标志接口NonWakeupRunnable, 然后wakesUpForTask()被覆盖为检测是否task是否是NonWakeupRunnable.

```
public abstract class SingleThreadEventLoop extends SingleThread
EventExecutor implements EventLoop {
    protected boolean wakesUpForTask(Runnable task) {
        return !(task instanceof NonWakeupRunnable);
    }

    interface NonWakeupRunnable extends Runnable {}
}
```

```
public final class NioEventLoop extends SingleThreadEventLoop {  
  
}
```

```
@Override  
protected Queue<Runnable> newTaskQueue() {  
    // This event loop never calls takeTask()  
    return PlatformDependent.newMpscQueue();  
}
```

Channel 是netty中提供的对网络操作的抽象类,类似于NIO中的Channel (java.nio.SocketChannel/java.nio.ServerSocketChannel). Netty的javadoc中是这样描述的:

A nexus to a network socket or a component which is capable of I/O operations such as read, write, connect, and bind. 对网络套接字或者有I/O操作能力(例如读,写,连接和绑定)的模块的关联(抽象?)

channel为使用者提供的功能:

- channel当前的状态(例如是不是打开? 是不是连接上了?)
- channel的配置参数(例如接收缓冲区大小)
- 提供I/O操作(例如读,写,连接和绑定)
- 提供ChannelPipeline,用于处理所有的I/O事件和关联到channel的请求

注意事项

netty channel的一些注意事项:

1. 所有的I/O操作都是异步的

netty中所有的I/O操作都是异步的. 这意味着任何I/O调用都将立即返回而不保证请求的I/O操作在调用结束时已经完成. 替代的是,你将得到一个返回的 **ChannelFuture** 实例,当请求的I/O操作成功,失败或者取消时将通知你.

2. channel是有等级的(hierarchical)

channel可以有一个parent, 取决于它是如何创建的. 例如, SocketChannel 是 ServerSocketChannel 在接受连接时创建的. SocketChannel 的parent()方法就会返回这个ServerSocketChannel.

等级结构的语义取决于channel所属的 **transport** 的实现. 例如, 可以写一个新的Channel实现, 创建子channel用于分享一个socket连接, 像BEEP 和 SSH 那样.

3. 向下溯型来访问传输特有(transport-specific)操作

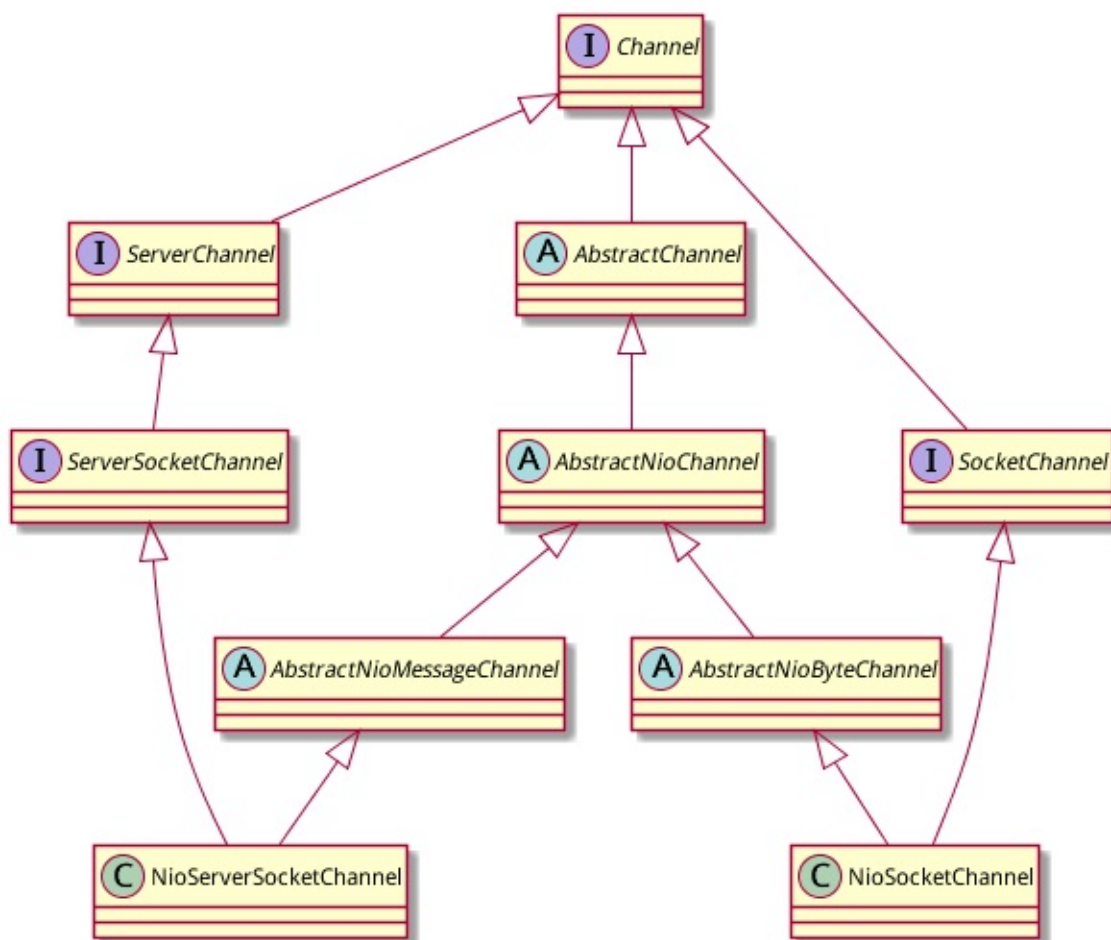
某些transport 暴露transport特有的一些额外的操作. 将Channel向下溯型为子类型来调用这些操作. 例如, 对于老的I/O datagram transport, DatagramChannel 提供了 multicast join / leave 操作.

4. 释放资源

有一个非常重要的事情, 当你完成channel的处理之后, 需要调用`close()` 或 `close(ChannelPromise)` 方法来释放所有的资源. 这样可以确保所有的资源以正确的方式被释放, 例如文件操作等.

Channel 继承结构

Channel的子类非常多, 仅以最常用的nio为例, `NioServerSocketChannel` 和 `NioSocketChannel` 的继承结构如下:



在这个继承结构中, `ServerChannel/ServerSocketChannel/SocketChannel` 三个 interface 的内容都非常简单(`ServerChannel`甚至是空的), 主要内容还在几个abstract channel实现类.

Channel接口的定义如下:

```
package io.netty.channel

public interface Channel extends AttributeMap, Comparable<Channel
> {
}
```

继承了AttributeMap接口用于提供Attribute的访问, 继承了Comparable接口用于???

TBD: 继承Comparable接口是为什么?

Channel接口方法定义

获取基础属性

- ChannelId id()
- Channel parent()
- ChannelConfig config()
- ChannelMetadata metadata(): 获取TCP参数配置

获取运行时属性

- EventLoop eventLoop()
- Channel.Unsafe unsafe()
- ChannelPipeline pipeline()
- ByteBufAllocator alloc()

获取状态

- boolean isOpen()
- boolean isRegistered()
- boolean isActive()
- boolean isWritable()

获取和网络相关属性

- SocketAddress localAddress()
- SocketAddress remoteAddress()

获取各种**Future**和**Promise**

- ChannelFuture closeFuture()
- ChannelPromise newPromise()
- ChannelProgressivePromise newProgressivePromise()
- ChannelFuture newSucceededFuture()
- ChannelFuture newFailedFuture(Throwable cause)
- ChannelPromise voidPromise()

IO操作

- ChannelFuture bind(SocketAddress localAddress)
- ChannelFuture bind(SocketAddress localAddress, ChannelPromise promise)
- ChannelFuture connect(SocketAddress remoteAddress)
- ChannelFuture connect(SocketAddress remoteAddress, SocketAddress localAddress)
- ChannelFuture connect(SocketAddress remoteAddress, ChannelPromise promise)
- ChannelFuture connect(SocketAddress remoteAddress, SocketAddress localAddress, ChannelPromise promise)
- ChannelFuture disconnect()
- ChannelFuture disconnect(ChannelPromise promise)
- ChannelFuture close()
- ChannelFuture close(ChannelPromise promise)
- ChannelFuture deregister()
- ChannelFuture deregister(ChannelPromise promise)
- Channel read()
- ChannelFuture write(Object msg)
- ChannelFuture write(Object msg, ChannelPromise promise)
- Channel flush()
- ChannelFuture writeAndFlush(Object msg)

- ChannelFuture writeAndFlush(Object msg, ChannelPromise promise)

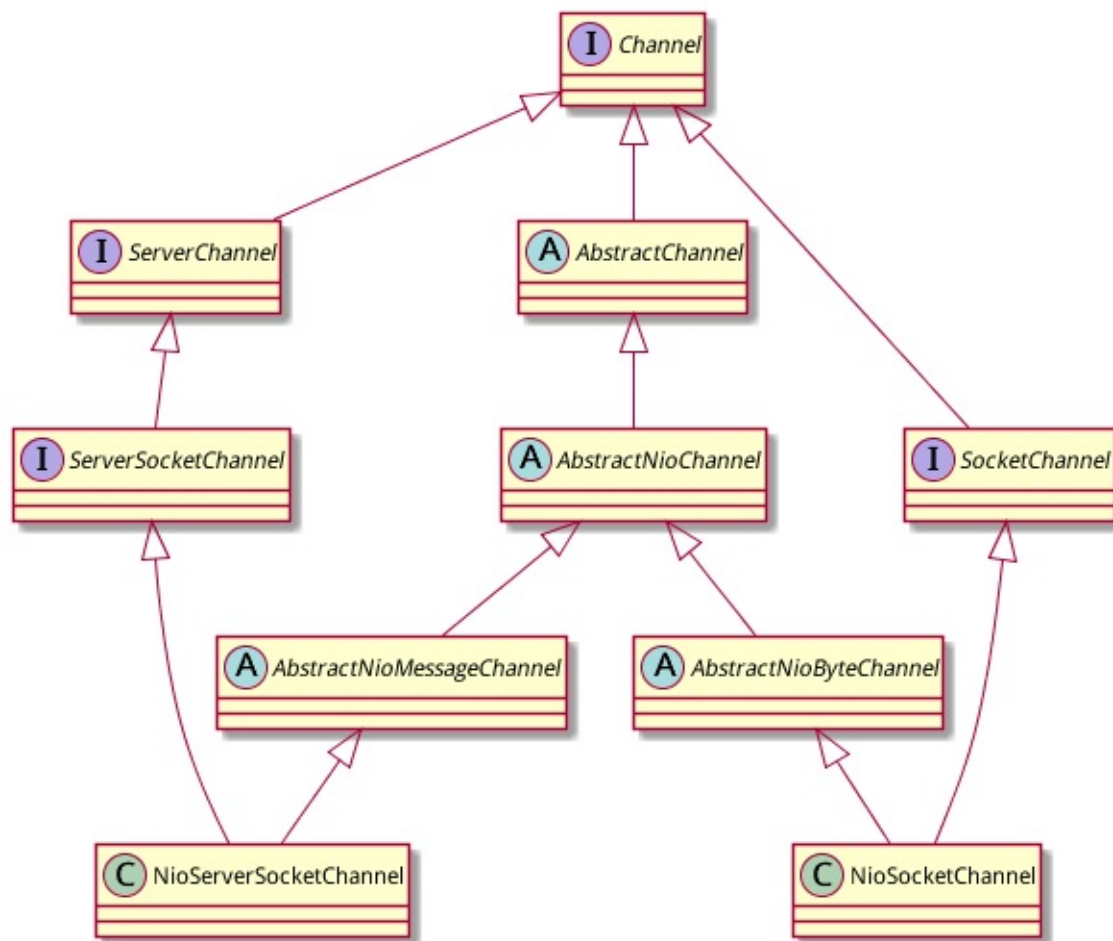
Unsafe

在Channel接口内部定义了一个Unsafe接口, 然后提供了一个unsafe()用于或者这个Channel实例关联的Unsafe对象实例:

```
public interface Channel extends AttributeMap, Comparable<Channel> {  
    .....  
    Unsafe unsafe();  
    .....  
    interface Unsafe {.....}  
}
```

这个Unsafe接口相关的内容后面再详细讲.

其他Channel接口



在Channel继承结构中, ServerChannel/ServerSocketChannel/SocketChannel 三个 interface 继承自 Channel.

ServerChannel

ServerChannel 仅仅是一个标识接口, 没有任何实质内容. 代码也只是简单的有个申明:

```
public interface ServerChannel extends Channel { }
```

ServerSocketChannel

ServerSocketChannel从ServerChannel继承, 内容也很少, 只是覆盖了几个定义在Channel的方法, 修改了函数的返回类型, 用更具体的子类型替代了Channel接口中定义的基础类型:

```
public interface ServerSocketChannel extends ServerChannel {
    @Override
    ServerSocketChannelConfig config();           // return ChannelConfig in Channel
    @Override
    InetSocketAddress localAddress();           // return SocketAddress in Channel
    @Override
    InetSocketAddress remoteAddress();          // return SocketAddress in Channel
}
```

SocketChannel

SocketChannel内容也不多:

1. 和ServerSocketChannel一样,覆盖了几个定义在Channel的方法,修改了函数的返回类型,用更具体的子类型替代了Channel接口中定义的基础类型:

```
public interface SocketChannel extends Channel {
    @Override
    ServerSocketChannel parent();           // return Channel in Channel

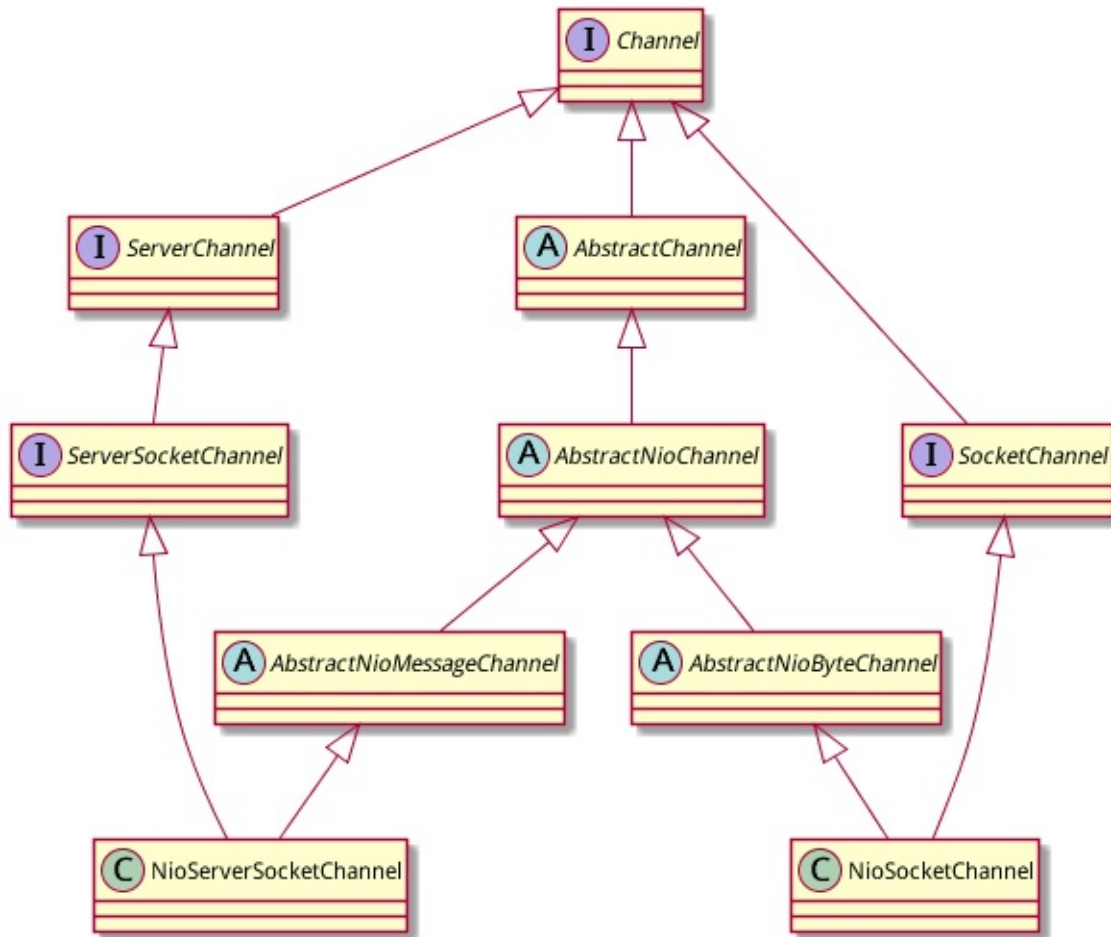
    @Override
    SocketChannelConfig config();           // return ChannelConfig in Channel
    @Override
    InetSocketAddress localAddress();       // return SocketAddress in Channel
    @Override
    InetSocketAddress remoteAddress();     // return SocketAddress in Channel
    .....
}
```

注意 parent() 的修改, SocketChannel应该都是从ServerSocketChannel中 accept得到,所以它的parent总应该是一个 ServerSocketChannel.

2. 定义了几个和socket相关的方法:

```
public interface SocketChannel extends Channel {  
    .....  
    boolean isInputShutdown();  
  
    boolean isOutputShutdown();  
  
    ChannelFuture shutdownOutput();  
  
    ChannelFuture shutdownOutput(ChannelPromise future);  
}
```

AbstractChannel顾名思义, 从Channel的继承结构上也可以看到, 在排除ServerChannel/ServerSocketChannel/SocketChannel三个接口定义之后, Channel的类层次就是从Channel -> AbstractChannel -> AbstractNioChannel 一路继承下去.



下面详细看代码实现.

类定义和外围实现

类定义

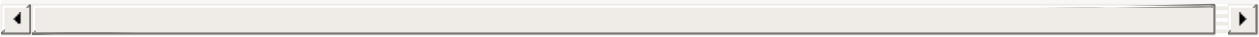
```
package io.netty.channel
```

```
public abstract class AbstractChannel extends DefaultAttributeMap
    implements Channel {}
```


外围接口实现

在Channel接口定义中声明的需要继承的两个接口AttributeMap/Comparable

```
public interface Channel extends AttributeMap, Comparable<Channel> {
```



通过继承DefaultAttributeMap实现了对接口AttributeMap的实现. Comparable接口的实现则是通过实现compareTo()方法来实现, compareTo()方法的代码:

```
@Override
public final int compareTo(Channel o) {
    if (this == o) {
        return 0;
    }

    return id().compareTo(o.id());
}
```

Channel接口实现

AbstractChannel实现了Channel接口定义的部分方法,作为所有Channel子类的基本实现.

下面是AbstractChannel中定义的几个基本属性和构造函数,注意这几个基本属性都是final:

```
private final Channel parent;
private final ChannelId id;
private final Unsafe unsafe;
private final DefaultChannelPipeline pipeline;

protected AbstractChannel(Channel parent) {
    this.parent = parent;
    id = DefaultChannelId.newInstance();
    unsafe = newUnsafe();
    pipeline = new DefaultChannelPipeline(this);
}

protected AbstractChannel(Channel parent, ChannelId id) {
    this.parent = parent;
    this.id = id;
    unsafe = newUnsafe();
    pipeline = new DefaultChannelPipeline(this);
}
```

id

每个Channel的实例都应该有一个唯一的id,这个id一旦设置就不能被修改.默认通过DefaultChannelId.newInstance()方法来获取:

```
private final ChannelId id;

protected AbstractChannel(Channel parent) {
    id = DefaultChannelId.newInstance();
    .....
}

protected AbstractChannel(Channel parent, ChannelId id) {
    this.id = id;
    .....
}

@Override
public final ChannelId id() {
    return id;
}
```

parent

每个Channel的parent在构造函数中指定:

```
private final Channel parent;

protected AbstractChannel(Channel parent) {
    this.parent = parent;
    .....
}

@Override
public Channel parent() {
    return parent;
}
```

pipeline

pipeline的初始化在构造函数中，目前看时直接固定为使用DefaultChannelPipeline的实现：

```
private final DefaultChannelPipeline pipeline;

protected AbstractChannel(Channel parent, ChannelId id) {
    .....
    pipeline = new DefaultChannelPipeline(this);
}

public ChannelPipeline pipeline() {
    return pipeline;
}
```

比较奇怪的是这里pipeline的类型定义是"DefaultChannelPipeline"，而不是"ChannelPipeline"。这个方式按说不正确，呵呵，验证过将类型修改为ChannelPipeline代码编译和测试都OK。

已经提交了一个pull request给netty，看看他们能否接受更新: netty接受了这个commit, 现在pipeline的类型已经修改为ChannelPipeline了。

在AbstractChannel的实现中，有很多和IO相关的方法是直接delegate给pipeline的，例如bind()方法:

```
public ChannelFuture bind(SocketAddress localAddress) {
    return pipeline.bind(localAddress);
}
```

类似delegate给pipeline的方法包括:

- bind()
- connect()
- disconnect()
- close()
- deregister()
- flush()
- read()
- write()
- writeAndFlush()

unsafe

unsafe的初始化也是在构造函数中，通过定义模板方法newUnsafe()让子类来给出具体的Unsafe的实现：

```
private final Unsafe unsafe;

protected AbstractChannel(Channel parent, ChannelId id) {
    .....
    unsafe = newUnsafe();
}

protected abstract AbstractUnsafe newUnsafe();

public Unsafe unsafe() {
    return unsafe;
}
```

在AbstractChannel的实现中，有部分方法是直接delegate给unsafe的，例如isWritable()方法：

```
public boolean isWritable() {
    ChannelOutboundBuffer buf = unsafe.outboundBuffer();
    return buf != null && buf.isWritable();
}
```

类似delegate给unsafe的方法包括：

- isWritable()
- bytesBeforeUnwritable()
- bytesBeforeWritable()
- localAddress()
- remoteAddress()

Channel Factory

由于历史原因，现在netty内有两个ChannelFactory类：

1. io.netty.bootstrap.ChannelFactory
2. io.netty.channel.ChannelFactory

bootstrap.ChannelFactory

这个类在package `io.netty.bootstrap` 中，明显这个package名不适合，所以这个 `io.netty.bootstrap.ChannelFactory` 已经被标记为"@Deprecated".

```
package io.netty.bootstrap;

@Deprecated
public interface ChannelFactory<T extends Channel> {
    /**
     * Creates a new channel.
     */
    T newChannel();
}
```

channel.ChannelFactory

现在要求使用package `io.netty.channel.ChannelFactory` 类，这个新类的定义有点奇怪，是从旧的bootstrap.ChannelFactory下继承：

```
package io.netty.channel;

public interface ChannelFactory<T extends Channel> extends io.netty.bootstrap.ChannelFactory<T> {
    /**
     * Creates a new channel.
     */
    @Override
    T newChannel();
}
```

ReflectiveChannelFactory

这是一个帮助创建Channel的工具类，给定要创建的Channel的类型，通过反射方式实现Channel的创建。

```
package io.netty.channel;

public class ReflectiveChannelFactory<T extends Channel> implements ChannelFactory<T> {
    // 内部保存一个Class
    private final Class<? extends T> clazz;

    // 构造函数传入
    public ReflectiveChannelFactory(Class<? extends T> clazz) {
        if (clazz == null) {
            throw new NullPointerException("clazz");
        }
        this.clazz = clazz;
    }

    @Override
    public T newChannel() {
        try {
            // 简单通过Class的newInstance()方法创建实例
            // 这也要求传入的Channel Class必须有默认构造函数
            return clazz.newInstance();
        } catch (Throwable t) {
            throw new ChannelException("Unable to create Channel from class " + clazz, t);
        }
        .....
    }
}
```

使用中这个类反而是最常用的，典型使用例子如下：

```
ServerBootstrap b = new ServerBootstrap();
b.channel(NioServerSocketChannel.class);
```


前面谈到在Channel接口内部定义了一个Unsafe接口, 然后有部分方法是直接delegate给unsafe的:

```
public boolean isWritable() {  
    ChannelOutboundBuffer buf = unsafe.outboundBuffer();  
    return buf != null && buf.isWritable();  
}
```

Unsafe的概念

Unsafe 是内部接口, 从设计上说这个接口的所有实现都是netty的内部代码, 只能被netty自己使用. netty的使用者时不应该 直接调用这些内部实现的.

Unsafe的javadoc是这样要求的:

Unsafe operations that should **never** be called from user-code. These methods are only provided to implement the actual transport, and must be invoked from an I/O thread except for the following methods:

Unsafe 操作不能被用户代码调用. 这些方法仅仅用于实现具体的**transport**, 而且必须被**I/O**线程调用.

以下方法例外(可以在I/O线程之外被调用):

- invoker()
- localAddress()
- remoteAddress()
- closeForcibly()
- register(EventLoop, ChannelPromise)
- deregister(ChannelPromise)
- voidPromise()

Unsafe 接口定义

Unsafe定义的方法

Unsafe接口中定义了以下方法:

获取特殊属性

- `RecvByteBufAllocator.Handle recvBufAllocHandle()`
- `ChannelHandlerInvoker invoker()`
- `ChannelOutboundBuffer outboundBuffer()`
- `ChannelPromise voidPromise()`

获取网络参数

- `SocketAddress localAddress()`
- `SocketAddress remoteAddress()`

注册到**EventLoop**

- `void register(EventLoop eventLoop, ChannelPromise promise)`
- `void deregister(ChannelPromise promise)`

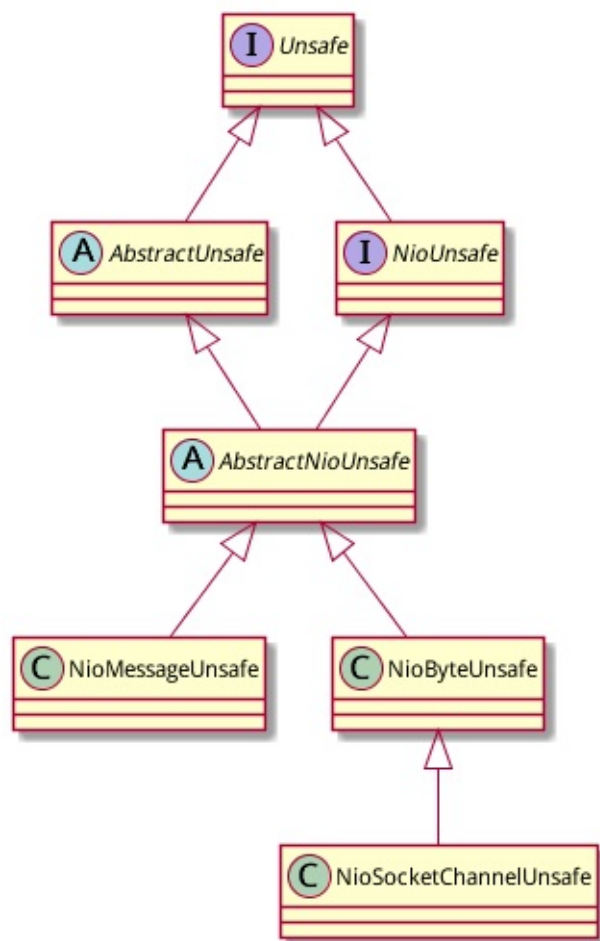
IO操作

- `void bind(SocketAddress localAddress, ChannelPromise promise)`
- `void connect(SocketAddress remoteAddress, SocketAddress localAddress, ChannelPromise promise)`
- `void disconnect(ChannelPromise promise)`
- `void close(ChannelPromise promise)`
- `void closeForcibly()`
- `void beginRead()`
- `void write(Object msg, ChannelPromise promise)`
- `void flush()`

Unsafe的实现

Unsafe的继承结构

以nio unsafe为例:



AbstractUnsafe

在AbstractChannel中定义有内部类AbstractUnsafe, 按照javadoc的说明, 所有的Unsafe的实现都必须从AbstractUnsafe继承:

```
public abstract class AbstractChannel extends DefaultAttributeMap
    implements Channel {
    /**
     * {@link Unsafe} implementation which sub-classes must extend and use.
     */
    protected abstract class AbstractUnsafe implements Unsafe {
    }
}
```


Channel Handler

继承结构

ChannelHandler的类继承结构如下:

```
@startuml

ChannelHandler <|-- ChannelInboundHandler
ChannelHandler <|-- ChannelOutboundHandler

ChannelHandler <|-- ChannelHandlerAdapter

ChannelInboundHandler <|-- ChannelInboundHandlerAdapter
ChannelHandlerAdapter <|-- ChannelInboundHandlerAdapter

ChannelOutboundHandler <|-- ChannelOutboundHandlerAdapter
ChannelHandlerAdapter <|-- ChannelOutboundHandlerAdapter

ChannelInboundHandlerAdapter <|-- ChannelDuplexHandler
ChannelOutboundHandler <|-- ChannelDuplexHandler

ChannelInboundHandlerAdapter <|-- ChannelInitializer

interface ChannelHandler {
+ handlerAdded(ctx)
+ handlerRemoved(ctx)
+ @interface Sharable
}

interface ChannelInboundHandler {
+ void channelRegistered(ctx)
+ void channelUnregistered(ctx)
+ void channelActive(ctx)
+ void channelInactive(ctx)
+ void channelRead(ctx, Object msg)
+ void channelReadComplete(ctx)
```

```
+ void userEventTriggered(ctx, evt)
+ void channelWritabilityChanged(ctx)
+ void exceptionCaught(ctx, cause)
}

interface ChannelOutboundHandler {
+ void bind(ctx, localAddress, promise)
+ void connect(ctx, remoteAddress, localAddress, promise)
+ void disconnect(ctx, promise)
+ void close(ctx, promise)
+ void deregister(ctx, promise)
+ void read(ctx)
+ void write(ctx, msg, promise)
+ void flush(ctx)
}

abstract class ChannelHandlerAdapter {
~ boolean added
}

class ChannelInboundHandlerAdapter {
}

class ChannelOutboundHandlerAdapter {
}

class ChannelDuplexHandler {
}

abstract ChannelInitializer {
# {abstract} void initChannel(ch)
}

@enduml
```

功能

Channel Handler的功能, 如在javadoc中所说:

处理I/O事件或者拦截I/O操作, 并转发给它所在ChannelPipeline中的下一个handler.

子类型

ChannelHandler本身并不提供很多方法, 但是通常需要实现它的子类型之一:

- ChannelInboundHandler 用于处理inbound I/O事件
- ChannelOutboundHandler 用于处理outbound I/O事件

或者, 为了方便使用, 提供了下面的adapter 类:

- ChannelInboundHandlerAdapter 用于处理inbound I/O事件
- ChannelOutboundHandlerAdapter 用于处理outbound I/O事件
- ChannelDuplexHandler 用于同时处理 inbound 和 outbound事件

上下文对象

ChannelHandler伴随有一个ChannelHandlerContext对象. ChannelHandler被设定为通过一个context对象来和它所属的ChannelPipeline交互.

通过使用这个context对象, ChannelHandler可以在上游和下游中传递事件, 动态修改pipeline, 或者存储handler特定的信息(使用AttributeKeys).

状态管理

ChannelHandler通常需要存储一些状态化的信息. 最简单的而且推荐的方式是使用成员变量:

使用成员变量

```
public interface Message {
    // your methods here
}

public class DataServerHandler extends SimpleChannelInboundHandler<Message> {

    private boolean loggedIn;

    @Override
    public void channelRead0(ChannelHandlerContext ctx, Message message) {
        Channel ch = e.getChannel();
        if (message instanceof LoginMessage) {
            authenticate((LoginMessage) message);
            loggedIn = true;
        } else (message instanceof GetDataMessage) {
            if (loggedIn) {
                ch.write(fetchSecret((GetDataMessage) message));
            } else {
                fail();
            }
        }
        ...
    }
}
```

因为这个handler实例有一个归属于某一个特定连接的状态变量, 我们不得不为每个新channel创建一个新的handler实例来.


```
// Create a new handler instance per channel.  
// See ChannelInitializer.initChannel(Channel).  
public class DataServerInitializer extends ChannelInitializer<Channel> {  
    @Override  
    public void initChannel(Channel channel) {  
        // 注意这里的DataServerHandler的实例是每次initChannel时都全新new  
        // 出来一个  
        // 这保证了DataServerHandler实例和channel的严格一对一关系  
        channel.pipeline().addLast("handler", new DataServerHandler(  
    ));  
    }  
}
```

使用AttributeKeys

Although it's recommended to use member variables to store the state of a handler, for some reason you might not want to create many handler instances. In such a case, you can use AttributeKeys which is provided by ChannelHandlerContext:

虽然推荐使用成员变量来存储handler的状态,但是因为某些原因,可能不想创建太多的handler实例. 在这种情况下,可以使用ChannelHandlerContext提供的AttributeKeys:

```
public interface Message {
    // your methods here
}

@Sharable
public class DataServerHandler extends SimpleChannelInboundHandler<Message> {
    // 申明一个AttributeKey
    private final AttributeKey<Boolean> auth = AttributeKey.valueOf(
        "auth");

    @Override
    public void channelRead(ChannelHandlerContext ctx, Message message) {
        // 从context对象中获取AttributeKey对应的Attribute
        Attribute<Boolean> attr = ctx.attr(auth);
        Channel ch = ctx.channel();
        if (message instanceof LoginMessage) {
            authenticate((LoginMessage) o);
            // 保存结果到Attribute
            attr.set(true);
        } else if (message instanceof GetDataMessage) {
            // 检查Attribute保存的值
            if (Boolean.TRUE.equals(attr.get())) {
                ch.write(fetchSecret((GetDataMessage) o));
            } else {
                fail();
            }
        }
    }
    ...
}
```

现在handler的状态被附加在ChannelHandlerContext, 你可以将同一个handler实例添加到不同的pipeline(对应不同的channel):

```
public class DataServerInitializer extends ChannelInitializer<Channel> {

    // 注意这里的DataServerHandler的实例是共享的，只有一个
    private static final DataServerHandler SHARED = new DataServerHandler();

    @Override
    public void initChannel(Channel channel) {
        // 所有的channel和pipeline，都使用这同一个handler的实例
        channel.pipeline().addLast("handler", SHARED);
    }
}
```

@Sharable 标签

在上面的使用AttributeKey的例子中，你可能已经发现@Sharable标签。

如果一个ChannelHandler被标注为@Sharable标签，这意味着你可以仅仅创建这个handler的一个实例然后将它多次添加到一个或者多个ChannelPipelines中，不会导致冲突或竞争。

如果没有特别标明这个标签，你将不得不在每次你将它添加到pipeline时创建一个新的handler实例，因为它有不共享的状态例如成员变量。

这个标签仅用于文档目的。

ChannelHandler相关接口

接口 ChannelHandler

ChannelHandler接口只定义了两个方法:

1. handlerAdded: 在ChannelHandler被添加到实际的context并准备处理事件后被调用
2. handlerRemoved: 在ChannelHandler被从实际的context中删除并不再处理事件后被调用

```
void handlerAdded(ChannelHandlerContext ctx) throws Exception;  
void handlerRemoved(ChannelHandlerContext ctx) throws Exception;
```

补充: 还有一个exceptionCaught()方法, 但是这个方法已经被标注为@Deprecated, javaoc上说"part of ChannelInboundHandler", 应该被转移到ChannelInboundHandler中了.

此外, 还有个@Sharable 标签的定义:

```
@Inherited  
@Documented  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@interface Sharable {  
    // no value  
}
```

前面说过, 这个@Sharable 标签只是用于文档目的.

接口 ChannelInboundHandler

接口ChannelInboundHandler的定义, 继承自接口ChannelHandler:

```
public interface ChannelInboundHandler extends ChannelHandler {}
```

然后添加了一系列的方法定义, 这些方法和inbound操作相关:

- `channelRegistered()`: ChannelHandlerContext的Channel被注册到EventLoop
- `channelUnregistered()`: ChannelHandlerContext的Channel被从EventLoop中注销
- `channelActive()`: ChannelHandlerContext的Channel被激活
- `channelInactive()`: 被注册的ChannelHandlerContext的Channel现在被取消并到达了它生命周期的终点
- `channelRead()`: 当Channel读取到一个消息时被调用
- `channelReadComplete()`: 当前读操作读取的最后一个消息被`channelRead()`方法消费时调用. 如果`ChannelOption.AUTO_READ` 属性被设置为off, 不会再尝试从当前channel中读取inbound数据, 直到`ChannelHandlerContext.read()`方法被调用.
- `userEventTriggered()`: 当用户事件被触发时调用
- `channelWritabilityChanged()`: 当channel的可写状态变化时被调用, 可以通过`Channel.isWritable()`来检查状态.
- `exceptionCaught()`: 当Throwable被抛出时被调用

```
void channelRegistered(ChannelHandlerContext ctx) throws Exception;
void channelUnregistered(ChannelHandlerContext ctx) throws Exception;
void channelActive(ChannelHandlerContext ctx) throws Exception;
void channelInactive(ChannelHandlerContext ctx) throws Exception;
void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception;
void channelReadComplete(ChannelHandlerContext ctx) throws Exception;
void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception;
void channelWritabilityChanged(ChannelHandlerContext ctx) throws Exception;
void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception;
```

接口 ChannelOutboundHandler

接口 ChannelOutboundHandler 的定义, 继承自接口 ChannelHandler:

```
public interface ChannelOutboundHandler extends ChannelHandler {
}
```

然后添加了一系列的方法定义, 这些方法和 outbound 操作相关:

- bind(): 当 bind 操作发生时被调用
- connect(): 当 connect 操作发生时被调用
- disconnect(): 当 disconnect 操作发生时被调用
- close(): 当 close 操作发生时被调用
- deregister(): 当在当前已经注册的 EventLoop 中发生 deregister 操作时被调用
- read(): 拦截 ChannelHandlerContext.read() 方法
- write(): 当 write 操作发生时被调用. 写操作将通过 ChannelPipeline 写信息. 这些信息做好了被 flush 到实际 channel 中的准备, 一旦 Channel.flush() 被调用.
- flush(): 当 flush 操作发生时被调用. flush 操作将试图将所有之前写入的被暂缓的

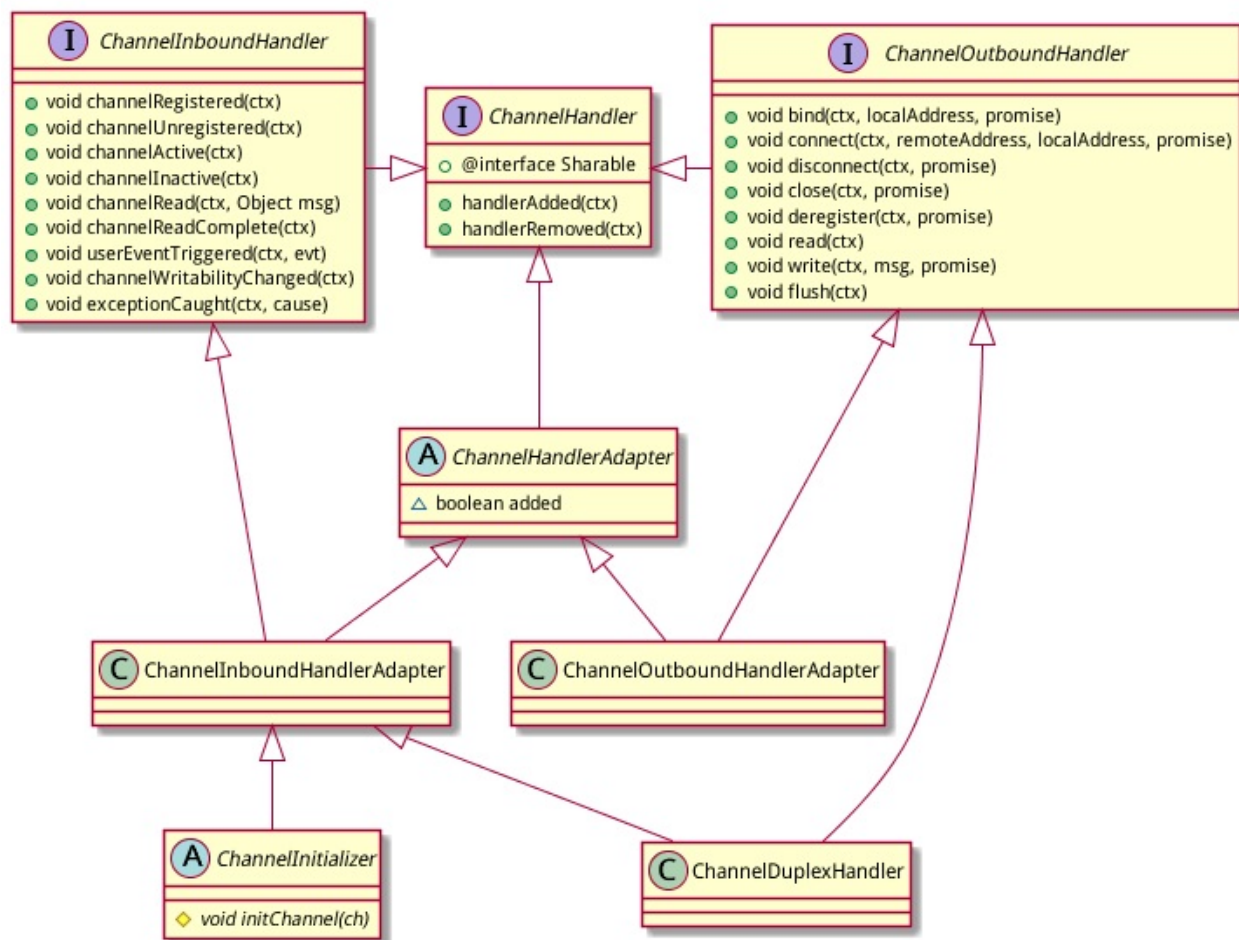
信息flush出去.

```
void bind(ChannelHandlerContext ctx, SocketAddress localAddress,
    ChannelPromise promise) throws Exception;
void connect(ChannelHandlerContext ctx, SocketAddress remoteAddress,
    SocketAddress localAddress, ChannelPromise promise) throws
    Exception;
void disconnect(ChannelHandlerContext ctx, ChannelPromise promise)
    throws Exception;
void close(ChannelHandlerContext ctx, ChannelPromise promise) throws
    Exception;
void deregister(ChannelHandlerContext ctx, ChannelPromise promise)
    throws Exception;
void read(ChannelHandlerContext ctx) throws Exception;
void write(ChannelHandlerContext ctx, Object msg, ChannelPromise
    promise) throws Exception;
void flush(ChannelHandlerContext ctx) throws Exception;
```

ChannelHandlerAdapter 相关类

为了方便使用, netty 提供了几个 adapter 类, 这些 adapter 类实现 ChannelHandler 接口并提供了一些平庸(空)实现, 这样子类可以方便的只覆盖自己关心的方法。

类继承结构如下:



类 ChannelHandlerAdapter

类定义, ChannelHandlerAdapter 实现了 ChannelHandler 接口:

```
public abstract class ChannelHandlerAdapter implements ChannelHandler {}
```

`handlerAdded()`和`handlerRemoved()`方法都给了一个空实现:


```
public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
    // NOOP
}
public void handlerRemoved(ChannelHandlerContext ctx) throws Exception {
    // NOOP
}
```

exceptionCaught()方法被delegate给ChannelHandlerContext了:

```
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
    ctx.fireExceptionCaught(cause);
}
```

此外提供了isSharable()方法来判断当前handler是否是可共享(在多个pipeline中,见前文):

```
public boolean isSharable() {
    Class<?> clazz = getClass();
    Map<Class<?>, Boolean> cache = InternalThreadLocalMap.get().
handlerSharableCache();
    Boolean sharable = cache.get(clazz);
    if (sharable == null) {
        sharable = clazz.isAnnotationPresent(Sharable.class);
        cache.put(clazz, sharable);
    }
    return sharable;
}
```

实现时有一个cache用来保存结果.

另外定义了一个added属性:

```
boolean added;
```

注: 类ChannelHandlerAdapter真没有什么内容, 主要是ChannelHandler接口也够简单. 继续看.

类ChannelInboundHandlerAdapter

类ChannelInboundHandlerAdapter的定义如下, 继承自ChannelHandlerAdapter, 另外声明实现ChannelInboundHandler接口:

```
public class ChannelInboundHandlerAdapter extends ChannelHandlerAdapter implements ChannelInboundHandler {}
```

类ChannelInboundHandlerAdapter实现了ChannelInboundHandler接口定义的各个inbound操作方法, 实现方式简单就一个话: 统统**delegate**给**ChannelHandlerContext!** 以channelRegistered()方法为例:

```
@Override
public void channelRegistered(ChannelHandlerContext ctx) throws Exception {
    ctx.fireChannelRegistered();
}
```

类ChannelOutboundHandlerAdapter

类ChannelOutboundHandlerAdapter也是类似, 类定义如下:

```
public class ChannelOutboundHandlerAdapter extends ChannelHandlerAdapter implements ChannelOutboundHandler {}
```

所有outbound方法也都是同样的delegate给ChannelHandlerContext, 以bind()方法为例:

```
public void bind(ChannelHandlerContext ctx, SocketAddress localAddress,
    ChannelPromise promise) throws Exception {
    ctx.bind(localAddress, promise);
}
```

类ChannelDuplexHandler

类ChannelDuplexHandler同时实现了ChannelInboundHandler和ChannelOutboundHandler:

```
public class ChannelDuplexHandler extends ChannelInboundHandlerAdapter implements ChannelOutboundHandler {}
```

方式方式也是一样的delegate给ChannelHandlerContext.

总结

这三个Adapter类的实现非常简单, 所有方法要不就是空实现, 要不就delegate给ChannelHandlerContext.

类ChannelInitializer

介绍

类ChannelInitializer是一个特殊的ChannelInboundHandler, 当一个Channel被注册到它的EventLoop时, 提供一个简单的方式用于初始化.

它的实现类通常在

Bootstrap.handler(ChannelHandler)/ServerBootstrap.handler(ChannelHandler)和ServerBootstrap.childHandler(ChannelHandler)的上下文中使用, 用于构建channel的ChannelPipeline.

```
public class MyChannelInitializer extends ChannelInitializer {
    public void initChannel(Channel channel) {
        channel.pipeline().addLast("myHandler", new MyHandler());
    }
}

ServerBootstrap bootstrap = ...;
...
bootstrap.childHandler(new MyChannelInitializer());
...
```

注意这个类被标记为ChannelHandler.Sharable, 因此它的实现类必须可以安全的被重用.

注: 平时实现自己的ChannelInitializer时需要注意这点.

实现

类定义

ChannelInitializer类定义如下, 继承自ChannelInboundHandlerAdapter:

```
@Sharable
public abstract class ChannelInitializer<C extends Channel> extends ChannelInboundHandlerAdapter {}
```

初始化逻辑实现

ChannelInitializer覆盖了方法channelRegistered(), 在这里实现了自己的初始化逻辑:

```
@Override
@SuppressWarnings("unchecked")
public final void channelRegistered(ChannelHandlerContext ctx) throws Exception { // step2
    initChannel((C) ctx.channel()); //step3
    ctx.pipeline().remove(this); //step5
    ctx.fireChannelRegistered(); // step6
}
```

步骤细节

我们结合前面的实现例子看:

```
public class MyChannelInitializer extends ChannelInitializer {
    public void initChannel(Channel channel) { // step3
        channel.pipeline().addLast("myHandler", new MyHandler()); //
    // step4
    }
}

bootstrap.childHandler(new MyChannelInitializer()); //step1
```

整个初始化的步骤和顺序如下:

- step1: "bootstrap.childHandler(new MyChannelInitializer());"

将MyChannelInitializer作为childHandler加入到bootstrap, 而且只加了这一个Handler.

- step2: bootstrap启动时回调channelRegistered()

bootstrap在启动后注册所有的handler, 此时会调用它所有的handler的channelRegistered()方法.

在前面我们只给bootstrap添加了一个handler(即MyChannelInitializer), 因此此时只有这个handler的channelRegistered()方法被调用.

- step3: 调用"initChannel((C) ctx.channel())"

ChannelInitializer的channelRegistered()方法实现中, 第一步就是调用抽象方法(模板方法)initChannel(), 实际的实现是MyChannelInitializer的initChannel()方法.

- step4: add MyHandler()到pipeline

这里是MyChannelInitializer真正做事情的地方, 添加它需要的handler到pipeline.

例子中只添加了一个, 实际这里通常是添加多个.

- step5: 将MyChannelInitializer删除

"ctx.pipeline().remove(this)" 将当前MyChannelInitializer从pipeline中删除, 以后的handler就只剩下MyChannelInitializer在step4中添加的handler了.

- step6:

"ctx.fireChannelRegistered()" 这里重新激活ChannelRegistered事件, 以便可以通知到step4中添加的各个handler.

下面是整个步骤的流程图:

```

@startuml

bootstrap -> bootstrap: add MyChannelInitializer to handler
note left: step1

-> bootstrap: start

bootstrap -> bootstrap : create Channel
bootstrap -> bootstrap : register MyChannelInitializer to pipeline
note left: step2
ChannelInitializer -> ChannelInitializer : channelRegistered() triggered

group channelRegistered()
ChannelInitializer -> MyChannelInitializer : initChannel()
note left: step3
group initChannel()
MyChannelInitializer -> MyChannelInitializer : add MyHandler to pipeline
note left: step4
end
MyChannelInitializer --> ChannelInitializer

ChannelInitializer -> ChannelInitializer : remove MyChannelInitializer from pipeline
note left: step5
ChannelInitializer -> ChannelInitializer : fire channelRegistered()
note left: step6
end

MyHandler -> MyHandler: channelRegistered() triggered
@enduml

```

异常处理

类ChannelInitializer对异常处理的方式就是做两件事情:

1. 打印日志
2. 关闭channel

```
@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable
    cause) throws Exception {
    //1. 打印日志
    logger.warn("Failed to initialize a channel. Closing: " + ctx.channel(), cause);
    try {
        ChannelPipeline pipeline = ctx.pipeline();
        // 在关闭channel之前, 检查channel的pipeline中是否还包含当前ChannelInitializer的handler
        if (pipeline.context(this) != null) {
            // 如果有, 去除它
            pipeline.remove(this);
        }
    } finally {
        // 2. 关闭channel
        ctx.close();
    }
}
```

当然子类可以继续覆盖这个实现. 不过一般都够用了.

类ChannelHandlerAppender

类的功能

类ChannelHandlerAppender是一个特殊的ChannelHandler, 如名字中的Appender所示, 这是一个包含多个特定 ChannelHandler 的Appender. 而且这些ChannelHandler会在ChannelHandlerAppender的右边(在pipeline中的位置).

默认情况下, 一旦这些特定的ChannelHandler被添加, 它会从ChannelPipeline中将自己删除. 当然, 也可以通过在构造函数中设置参数selfRemoval为false来保留它.

类的定义

类ChannelHandlerAppender的定义, 简单继承自ChannelInboundHandlerAdapter:

```
public class ChannelHandlerAppender extends ChannelInboundHandlerAdapter {}
```

ChannelHandlerAppender的子类实现有一下, 基本都是给codec用的, 具体继承结构如图:

```
@startuml

ChannelInboundHandlerAdapter <|-- ChannelHandlerAppender

ChannelHandlerAppender <|-- HttpServerCodec
ChannelHandlerAppender <|-- HttpClientCodec
ChannelHandlerAppender <|-- SpdyHttpClientCodec
ChannelHandlerAppender <|-- BinaryMemcacheServerCodec
ChannelHandlerAppender <|-- BinaryMemcacheClientCodec

class ChannelHandlerAppender {
- boolean selfRemoval
- List<Entry> handlers
- boolean added

+ add(handler)
}

@enduml
```

类的实现

属性

ChannelHandlerAppender有三个属性:

```
private final boolean selfRemoval;
private final List<Entry> handlers = new ArrayList<Entry>();
private boolean added;
```

1. **selfRemoval**: 标记在将append的各个handler添加到pipeline之后, 是否要将自身删除, 默认行为是true/删除, 可以在构造函数中设置
2. **handlers**: 这个就是内部保存的要append的各个handler了, 可以通过add()方法添加
3. **added**: 标记append的各个handler是否已经被添加到pipeline中了

上面的add()方法中, 会检查added属性, 如果已经添加了则抛异常.

相关的代码示意:

```
public ChannelHandlerAppender(boolean selfRemoval, ChannelHandle
r... handlers) {
    this.selfRemoval = selfRemoval;    // 构造函数中指定
    add(handlers);
}
protected final ChannelHandlerAppender add(String name, ChannelH
andler handler) {
    if (added) {    // 检查added属性
        throw new IllegalStateException("added to the pipeline a
lready");
    }

    handlers.add(new Entry(name, handler));
    return this;
}

@Override
public void handlerAdded(ChannelHandlerContext ctx) throws Excep
tion {
    added = true;    //设置added为true

    DefaultChannelPipeline pipeline = (DefaultChannelPipeline) d
ctx.pipeline();
    try {

    } finally {
        if (selfRemoval) {    // 检查selfRemoval属性看是否要删除自身
            pipeline.remove(this);
        }
    }
}
```

addpend的代码实现

handlerAdded()方法中是这个类真正的逻辑实现:

```
@Override
public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
    added = true;

    AbstractChannelHandlerContext dctx = (AbstractChannelHandlerContext) ctx;
    DefaultChannelPipeline pipeline = (DefaultChannelPipeline) dctx.pipeline();
    String name = dctx.name();
    try {
        for (Entry e: handlers) { // 循环处理每个append的handler
            String oldName = name; // 第一个oldname是dctx.name(),
            // 也就是ChannelHandlerAppender的name, 以后oldname是上一次的name
            if (e.name == null) {
                name = pipeline.generateName(e.handler);
            } else {
                name = e.name;
            }

            // 添加这个handler到上一个hanler(由oldName指定)之后
            pipeline.addAfter(dctx.invoker, oldName, name, e.handler);
        }
    } finally {
        if (selfRemoval) { // 检查是否要删除自身
            pipeline.remove(this);
        }
    }
}
```

备注: 这个类的具体功能, 在后面细读codec实现时再结合使用场景一起看.

Channel Handler Context

功能

参考[接口ChannelHandlerContext的javadoc](#)说明, 接口ChannelHandlerContext的作用:

使得ChannelHandler可以和它的ChannelPipeline和其他handlers交互. 除此之外, handler可以通知ChannelPipeline中的下一个ChannelHandler, 此外还可以动态的修改它所属的ChannelPipeline

具体功能如下(下面翻译自javadoc):

1. 通知

可以通过调用这里提供的多个方法中的某一个方法来通知在同一个ChannelPipeline的最近的handler. 参考ChannelPipeline来理解事件是如何流动的.

2. 修改pipeline

可以通过调用pipeline()方法来得到handler所属的ChannelPipeline. 有一定分量的(non-trivial)应用能在pipeline中动态的插入, 删除, 或者替换handler.

3. 保存以便后续可以取回使用

可以保存ChannelHandlerContext以便后续使用, 例如在handler方法外触发一个事件, 甚至可以从不同的线程发起.

```
public class MyHandler extends ChannelHandlerAdapter {  
  
    private ChannelHandlerContext ctx;  
  
    public void beforeAdd(ChannelHandlerContext ctx) {  
        this.ctx = ctx;  
    }  
  
    public void login(String username, password) {  
        ctx.write(new LoginMessage(username, password));  
    }  
    ...  
}
```

4. 存储状态化信息

`AttributeMap.attr(AttributeKey)` 容许你保存和访问和handler以及它的上下文相关的状态化信息. 参考ChannelHandler来学习多种推荐用于管理状态化信息的方法.

有一个非常重要的信息, 有些出乎意外, 需要特别注意和留神:

一个handler可以多个context

请注意, 一个ChannelHandler实例可以被添加到多个ChannelPipeline. 这意味着一个单独的ChannelHandler实例可以有多个ChannelHandlerContext, 因此一个单独的ChannelHandler实例可以在被调用时传入不同的ChannelHandlerContext, 如果它被添加到多个ChannelPipeline.

例如, 下面的handler被加入到pipelines多少次, 就会有多少个相互独立的AttributeKey. 不管它是多次添加到同一个pipeline还是不同的pipeline.

```

public class FactorialHandler extends ChannelHandlerAdapter {

    private final AttributeKey<Integer> counter = AttributeKey.v
alueOf("counter");

    // This handler will receive a sequence of increasing intege
rs starting
    // from 1.
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object ms
g) {
        Attribute<Integer> attr = ctx.getAttr(counter);
        Integer a = ctx.getAttr(counter).get();

        if (a == null) {
            a = 1;
        }

        attr.set(a * (Integer) msg);
    }
}

// 不同的context对象被赋予"f1", "f2", "f3", 和 "f4", 虽然他们引用到同
一个handler实例.
// 由于FactorialHandler在context对象中保存它的状态(使用AttributeKey),
// 一旦这两个pipelines (p1 和 p2)被激活, factorial会计算4次.
FactorialHandler fh = new FactorialHandler();

ChannelPipeline p1 = Channels.pipeline();
p1.addLast("f1", fh);
p1.addLast("f2", fh);

ChannelPipeline p2 = Channels.pipeline();
p2.addLast("f3", fh);
p2.addLast("f4", fh);

```

备注: 这个的代码和例子来自javadoc,但是有点问题, ctx.getAttr()应该是ctx.attr(). 已经pull了一个request给netty. 更新: 修改已经被netty接受, netty4.1版本会包含这个修订.

总结: 也就是说, 在handler被添加到pipeline时, 会创建一个新的context, 无视handler是否是一个实例添加多次.

类继承结构

ChannelHandlerContext继承关系简单, 就一个ChannelHandlerContext接口, 一个AbstractChannelHandlerContext抽象类, 和一个默认实现DefaultChannelHandlerContext.

另外在DefaultChannelPipeline中内嵌有两个特殊的实现类TailContext和HeadContext.

```
@startuml

AttributeMap <|-- ChannelHandlerContext
ChannelHandlerContext <|-- AbstractChannelHandlerContext
AbstractChannelHandlerContext <|-- DefaultChannelHandlerContext
AbstractChannelHandlerContext <|-- TailContext
AbstractChannelHandlerContext <|-- HeadContext

interface AttributeMap {
}

interface ChannelHandlerContext {
}

abstract class AbstractChannelHandlerContext {
    - boolean inbound;
    - boolean outbound;
    - ChannelPipeline pipeline;
    - String name;
    - boolean removed;

    - AbstractChannelHandlerContext next;
    - AbstractChannelHandlerContext prev;
}
```



```
class DefaultChannelHandlerContext {  
  - ChannelHandler handler  
}  
  
class TailContext {  
  - ChannelHandler handler  
}  
  
class HeadContext {  
  - ChannelHandler handler  
}  
  
@enduml
```

接口 ChannelHandlerContext

接口属性

name属性

```
String name();
```

ChannelHandlerContext的名字, unique, 不能重复.

这个名字在ChannelHandler被添加到ChannelPipeline时使用. 例如:

```
p1.addLast("f1", handler);
```

也可以用来通过名字在ChannelPipeline访问已经注册的ChannelHandler.

```
ChannelHandler get(String name);
```

实际上就是在ChannelPipeline的各个方法中出现的参数 name 或者 baseName.

接口方法

获取属性

removed

removed属性如果是true, 表明属于当前context的ChannelHandler已经从ChannelPipeline中移除.

注意这个方法通常只在EventLoop中调用.

```
boolean isRemoved();
```

获取context保存相关对象

以下方法用于获取context相关的几个重要对象, context中保存有他们的引用:

- channel: 当前的channel, 代表当前的连接
- handler: 当前的ChannelHandler, 记住context是一个handler实例添加到一个pipeline时产生的一个"契约"(subscription).
- pipeline: 当前所在的ChannelPipeline, 解释同上
- executor: EventExecutor, 用于执行任意任务
- alloc: 当前被分配的ByteBufAllocator, 用于ByteBuf的内存分配
- invoker: netty5.0中多暴露了一个invoker, netty4.1中没有

```
Channel channel();
ChannelHandler handler();
ChannelPipeline pipeline();
EventExecutor executor();
ByteBufAllocator alloc();
```

Inbound的I/O事件方法

这些方法会导致当前Channel的ChannelPipeline中包含的下一个ChannelInboundHandler的相应的方法被调用:

```
ChannelHandlerContext fireChannelRegistered();
ChannelHandlerContext fireChannelUnregistered();
ChannelHandlerContext fireChannelActive();
ChannelHandlerContext fireChannelInactive();
ChannelHandlerContext fireExceptionCaught(Throwable cause);
ChannelHandlerContext fireUserEventTriggered(Object event);
ChannelHandlerContext fireChannelRead(Object msg);
ChannelHandlerContext fireChannelReadComplete();
ChannelHandlerContext fireChannelWritabilityChanged();
```

outbound的I/O方法

这些方法会导致当前Channel的ChannelPipeline中包含的下一个ChannelOutboundHandler的相应的方法被调用:

```
ChannelFuture bind(SocketAddress localAddress);
ChannelFuture connect(SocketAddress remoteAddress);
ChannelFuture connect(SocketAddress remoteAddress, SocketAddress
    localAddress);
ChannelFuture disconnect();
ChannelFuture close();
ChannelFuture deregister();
```

以及以上方法的带promise参数的重载方法:

```
ChannelFuture bind(SocketAddress localAddress, ChannelPromise promise);
ChannelFuture connect(SocketAddress remoteAddress, ChannelPromise promise);
ChannelFuture connect(SocketAddress remoteAddress, SocketAddress
    localAddress, ChannelPromise promise);
ChannelFuture disconnect(ChannelPromise promise);
ChannelFuture close(ChannelPromise promise);
ChannelFuture deregister(ChannelPromise promise);
```

I/O操作方法

```
// 请求从Channel中读取数据到第一个inbound buffer, 如果数据被读取则触发ChannelInboundHandler.channelRead(ChannelHandlerContext, Object)事件, 并触发channelReadComplete事件以便handler可以决定继续读取.  
// 将会导致当前Channel的ChannelPipeline中包含的下一个ChannelOutboundHandler的ChannelOutboundHandler.read(ChannelHandlerContext)方法被调用
```

```
ChannelHandlerContext read();
```

```
// 请求通过当前ChannelHandlerContext将消息写到ChannelPipeline.  
// 这个方法不会要求做实际的flush, 因此一旦你想将所有待处理的数据flush到实际的transport请务必调用flush()方法.
```

```
ChannelFuture write(Object msg);
```

```
ChannelFuture write(Object msg, ChannelPromise promise);
```

```
// 请求通过ChannelOutboundInvoker flush所有待处理的消息
```

```
ChannelHandlerContext flush();
```

```
// write()和flush()方法的快捷调用版本
```

```
ChannelFuture writeAndFlush(Object msg, ChannelPromise promise);
```

```
ChannelFuture writeAndFlush(Object msg);
```

TODO: 不理解为什么read()会导致

ChannelOutboundHandler.read(ChannelHandlerContext)方法被调用, 为什么是Outbound? 后面注意看实现代码.

创建Future/Promise的方法

```
ChannelProgressivePromise newProgressivePromise();
```

```
ChannelFuture newSucceededFuture();
```

```
ChannelFuture newFailedFuture(Throwable cause);
```

```
ChannelPromise voidPromise();
```

类AbstractChannelHandlerContext

类定义

```
abstract class AbstractChannelHandlerContext implements ChannelH  
andlerContext, ResourceLeakHint {}
```

类属性

只有一个构造函数, name/pipeline/invoke/inbound/outbound都在这里初始化:

```
private final boolean inbound;
private final boolean outbound;
private final ChannelPipeline pipeline;
private final String name;
private boolean removed;

final ChannelHandlerInvoker invoker;

AbstractChannelHandlerContext(
    DefaultChannelPipeline pipeline, ChannelHandlerInvoker i
nvoker,
    String name, boolean inbound, boolean outbound) {

    if (name == null) {
        throw new NullPointerException("name");
    }

    this.pipeline = pipeline;          // 为什么pipeline不做null检测
    ? 从代码实现上看肯定不能为null的, 既然检查了name就应该检查pipeline
    this.name = name;
    this.invoker = invoker;            // 这个invoker是容许设置为nu
    ll的!! 看invoker()函数

    this.inbound = inbound;
    this.outbound = outbound;
}
```

TODO: 准备增加pipeline的null check, 然后pull给netty. 等上一个request被接受再试.

有几个属性的getter方法比较特殊:

```
@Override
public ByteBufAllocator alloc() {
    // alloc从channel的config中获取
    return channel().config().getAllocator();
}

@Override
public EventExecutor executor() {
    // EventExecutor来自invoker
    return invoker().executor();
}

public ChannelHandlerInvoker invoker() {
    // 检查invoker是否有设置为null, 见构造函数
    if (invoker == null) {
        // 如果构造函数中设置为null了, 则通过unsafe来获取
        return channel().unsafe().invoker();
    } else {
        return invoker;
    }
}
```

invoker属性

invoker容许为null比较奇怪, 特意查了一下, 有两个地方是可以设置invoker为null的:

```
HeadContext(DefaultChannelPipeline pipeline) {
    super(pipeline, null, HEAD_NAME, false, true);
    unsafe = pipeline.channel().unsafe();
}
TailContext(DefaultChannelPipeline pipeline) {
    super(pipeline, null, TAIL_NAME, true, false);
}
```

TODO: 后面看HeadContext/TailContext的代码实现时再仔细看.

removed属性

这是一个非常特别的属性,只用于非常极端的情况,getter/setter方法如下:

```
void setRemoved() {
    removed = true;
}

@Override
public boolean isRemoved() {
    return removed;
}
```

其中setRemoved()还不是public的,调用的地方只有一处,在类DefaultChannelPipeline:

```
private void callHandlerRemoved0(final AbstractChannelHandlerContext ctx) {
    // Notify the complete removal.
    try {
        ctx.handler().handlerRemoved(ctx);
        ctx.setRemoved();
    } catch (Throwable t) {
        fireExceptionCaught(new ChannelPipelineException(
            ctx.handler().getClass().getName() + ".handlerRemoved() has thrown an exception.", t));
    }
}
```

只有当一个handler被从pipeline上remove或者replace时,才会调用setRemoved()方法.

而这个属性使用的地方也非常少,只找到三处调用,从注释上看都和issues 1664相关:

```
// Check if this handler was removed before continuing the loop.
// If it was removed, it is not safe to continue to operate on the buffer.
//
// See https://github.com/netty/netty/issues/1664
if (ctx.isRemoved()) {
    break;
}
```

next 和 prev

每个AbstractChannelHandlerContext都保存有两个属性, next和prev, 明显是做链表.

```
volatile AbstractChannelHandlerContext next;
volatile AbstractChannelHandlerContext prev;
```

在DefaultChannelPipeline中, 则保存了名为head和tail的两个引用:

```
final class DefaultChannelPipeline implements ChannelPipeline {
    final AbstractChannelHandlerContext head;
    final AbstractChannelHandlerContext tail;
}
```

这样在DefaultChannelPipeline中就实现了一个从head一路next/next到tail, 再从tail一路prev/prev到head的一个双向链表.

inbound 和 outbound

这两个属性用来表明当前handler是inbound还是outbound, 通常和ChannelInboundHandler/ChannelOutboundHandler联系起来,.

比如HeadContext实现了ChannelOutboundHandler, 而TailContext实现了ChannelInboundHandler, 他们在调用super构造函数时就写死了inbound和outbound属性:

```
static final class HeadContext extends AbstractChannelHandlerContext implements ChannelOutboundHandler {
    HeadContext(DefaultChannelPipeline pipeline) {
        super(pipeline, null, HEAD_NAME, false, true); // inbound=false, outbound=true
    }
}
static final class TailContext extends AbstractChannelHandlerContext implements ChannelInboundHandler {
    TailContext(DefaultChannelPipeline pipeline) {
        super(pipeline, null, TAIL_NAME, true, false); // inbound=true, outbound=false
    }
}
```

注: netty设计上是使用两个boolean来记录inbound/outbound, 不知道是否存在即是inbound又是outbound 的情况?

在DefaultChannelHandlerContext的实现中, 通过检查传入的handler来判断inbound/outbound, 判断的方法非常直接, instanceof ChannelInboundHandler/ChannelOutboundHandler:

```
DefaultChannelHandlerContext(
    ChannelPipeline pipeline, ChannelHandlerInvoker invoker,
    String name, ChannelHandler handler) {
    super(pipeline, invoker, name, isInbound(handler), isOutbound(handler));
    if (handler == null) {
        throw new NullPointerException("handler");
    }
    this.handler = handler;
}
private static boolean isInbound(ChannelHandler handler) {
    return handler instanceof ChannelInboundHandler;
}
private static boolean isOutbound(ChannelHandler handler) {
    return handler instanceof ChannelOutboundHandler;
}
```

注: 如果一个handler同时实现ChannelInboundHandler/ChannelOutboundHandler两个接口, 是有可能的. 但具体是否有这种情况还不清楚, 继续看代码.

类方法

来自AttributeMap的方法

ChannelHandlerContext申明继承AttributeMap, AbstractChannelHandlerContext中有实现AttributeMap要求的两个方法:

```
@Override
public <T> Attribute<T> attr(AttributeKey<T> key) {
    return channel().attr(key);
}

@Override
public <T> boolean hasAttr(AttributeKey<T> key) {
    return channel().hasAttr(key);
}
```

最终还是delegate给channel的对应方法了.

Inbound的I/O事件方法

这些方法最终都是delegate给invoker的对应方法, 以fireChannelRegistered()为例:

```
@Override
public ChannelHandlerContext fireChannelRegistered() {
    // 找到下一个inbound的context
    AbstractChannelHandlerContext next = findContextInbound();
    // 调用invoker的invokeChannelRegistered()方法
    next.invoker().invokeChannelRegistered(next);
    return this;
}

private AbstractChannelHandlerContext findContextInbound() {
    // ctx初始化为指向当前context, 也就是this
    AbstractChannelHandlerContext ctx = this;
    do {
        // 然后指向ctx的next指针
        ctx = ctx.next;
    } while (!ctx.inbound); // 检查是否是inbound, 如果不是则继续next, 直到找到下一个
    return ctx;
}
```

这和javadoc中对这些方法的说明一致: "这些方法会导致当前Channel的ChannelPipeline中包含的下一个ChannelInboundHandler的相应的方法被调用"

类似的方法有以下:

```
ChannelHandlerContext fireChannelRegistered();
ChannelHandlerContext fireChannelUnregistered();
ChannelHandlerContext fireChannelActive();
ChannelHandlerContext fireChannelInactive();
ChannelHandlerContext fireExceptionCaught(Throwable cause);
    // 这个例外!
ChannelHandlerContext fireUserEventTriggered(Object event);
ChannelHandlerContext fireChannelRead(Object msg);
ChannelHandlerContext fireChannelReadComplete();
ChannelHandlerContext fireChannelWritabilityChanged();
```

但是fireExceptionCaught方法非常的特殊, 与众不同的是, 这个方法中的next是不区分inbound和outbound的, 直接取this.next:

```
@Override
public ChannelHandlerContext fireExceptionCaught(Throwable cause)
{
    AbstractChannelHandlerContext next = this.next;
    next.invoke().invokeExceptionCaught(next, cause);
    return this;
}
```

TODO: 暂时还没有理解为什么fireExceptionCaught()方法会有这个特例?

outbound的I/O方法

首先, 下面这些方法在实现时都会转变为他们对应的带promise的版本:

```
ChannelFuture bind(SocketAddress localAddress);
ChannelFuture connect(SocketAddress remoteAddress);
ChannelFuture connect(SocketAddress remoteAddress, SocketAddress
    localAddress);
ChannelFuture disconnect();
ChannelFuture close();
ChannelFuture deregister();
```

例如bind()方法, 通过newPromise()方法创建一个promise之后, 就调用带promise的版本了:

```
@Override
public ChannelFuture bind(SocketAddress localAddress) {
    return bind(localAddress, newPromise());
}

@Override
public ChannelPromise newPromise() {
    return new DefaultChannelPromise(channel(), executor());
}
```

继续看bind()方法的实现, 发现机制和前面的雷同:

```
@Override
public ChannelFuture bind(final SocketAddress localAddress, final
    ChannelPromise promise) {
    // 找到下一个outbound的context
    AbstractChannelHandlerContext next = findContextOutbound();
    // 调用invokeBind()方法
    next.invoker().invokeBind(next, localAddress, promise);
    return promise;
}
```



I/O操作方法

实现方式类似, 不细看了.

类DefaultChannelHandlerContext

ChannelHandlerContext的实现类有三个:

1. DefaultChannelHandlerContext
2. HeadContext (内嵌在DefaultChannelPipeline中)
3. TailContext (内嵌在DefaultChannelPipeline中)

类DefaultChannelHandlerContext

DefaultChannelHandlerContext的代码特别的少,基本上除了保存一个handler属性和getter方法,就没有其他内容了:

```
final class DefaultChannelHandlerContext extends AbstractChannelHandlerContext {
    private final ChannelHandler handler;

    DefaultChannelHandlerContext(ChannelPipeline pipeline, ChannelHandlerInvoker invoker, String name, ChannelHandler handler) {
        super(pipeline, invoker, name, isInbound(handler), isOutbound(handler));
        if (handler == null) {
            throw new NullPointerException("handler");
        }
        this.handler = handler;
    }

    @Override
    public ChannelHandler handler() {
        return handler;
    }
}
```

比较奇怪的是,为什么handler要放在DefaultChannelHandlerContext中,而不是AbstractChannelHandlerContext?

TailContext

TailContext和HeadContext有些奇怪, 他们既是ChannelHandlerContext, 又实现了ChannelHandler接口.

TailContext是ChannelInboundHandler:

```
// A special catch-all handler that handles both bytes and messages.
static final class TailContext extends AbstractChannelHandlerContext implements ChannelInboundHandler {
    private static final String TAIL_NAME = generateName0(TailContext.class);

    TailContext(DefaultChannelPipeline pipeline) {
        // invoker特意设置为null
        // inbound=true, outbound=false
        super(pipeline, null, TAIL_NAME, true, false);
    }

    @Override
    public ChannelHandler handler() {
        return this;
    }
}
```

HeadContext

HeadContext是ChannelOutboundHandler:

```
static final class HeadContext extends AbstractChannelHandlerContext implements ChannelOutboundHandler {
    private static final String HEAD_NAME = generateName0(HeadContext.class);

    private final Unsafe unsafe;

    HeadContext(DefaultChannelPipeline pipeline) {
        // invoker特意设置为null
        // inbound=false, outbound=true
        super(pipeline, null, HEAD_NAME, false, true);
        unsafe = pipeline.channel().unsafe();
    }
}
```

注: 从代码实现上看, TailContext和HeadContext在DefaultChannelPipeline中扮演特殊角色, 还是和DefaultChannelPipeline的代码一起看比较好.

Channel Pipeline

netty中和Servlet Filter机制类似, 实现了的Channel的过滤器, 具体就是ChannelPipeline和ChannelHandler.

- ChannelHandler: 负责对I/O事件进行拦截和处理
- ChannelPipeline: ChannelHandler的容器, 管理ChannelHandler

Pipeline的构建

通常不需要用户自己构建Pipeline, 在使用ServerBootstrap或者Bootstrap时, ServerBootstrap或者Bootstrap会为每个Channel创建一个独立的Pipeline.

代码分析

ChannelPipeline的代码不多, 继承关系简单, 就一个ChannelPipeline接口和一个默认实现DefaultChannelPipeline.

```
@startuml  
  
ChannelPipeline <|-- DefaultChannelPipeline  
  
@enduml
```

接口ChannelPipeline

接口定义

ChannelPipeline继承了Iterable, Entry表明ChannelPipeline内部是保存有一个以String为key, ChannelHandler为value的类似map的结构:

```
public interface ChannelPipeline extends Iterable<Entry<String,  
ChannelHandler>> {  
}
```

接口方法

管理ChannelHandler

增加ChannelHandler

1. addFirst()方法: 将handler添加到pipeline的第一个位置

```
ChannelPipeline addFirst(String name, ChannelHandler handler);  
ChannelPipeline addFirst(EventExecutorGroup group, String name, ChannelHandler handler);  
ChannelPipeline addFirst(ChannelHandlerInvoker invoker, String name, ChannelHandler handler);
```

参数中, name是当前要插入的handler的名字, 如果设置为null则自动生成.

注意: **name**不容许重复, 如果添加的时候发现已经存在同样name的handler, 则会抛出IllegalArgumentException.

2. addLast()方法

和addFirst()方法完全对应

1. addBefore()方法

```
ChannelPipeline addBefore(String baseName, String name, ChannelHandler handler);
ChannelPipeline addBefore(EventExecutorGroup group, String baseName, String name, ChannelHandler handler);
ChannelPipeline addBefore(ChannelHandlerInvoker invoker, String baseName, String name, ChannelHandler handler);
```

参数中, name和addFirst()中一致, 但是多了一个baseName, 这个是插入的基准handler的名字, 即要插在这个名字的handle前面.

2. addAfter()方法

和addBefore()方法完全对应.

另外以上方法还有可以参数的方法重载, 无非就是将参数中的一个handler变成ChannelHandler... handlers.

删除ChannelHandler

remove()方法查找给定参数的handler, 并从ChannelPipeline中删除它, 然后将被删除的handle返回:

```
ChannelPipeline remove(ChannelHandler handler);
ChannelHandler remove(String name);
<T extends ChannelHandler> T remove(Class<T> handlerType);
ChannelHandler removeLast();
ChannelHandler removeFirst();
```

注意: 如果删除时发现找不到要删除的目标, 这些remove()方法会抛出NoSuchElementException, 这个要小心处理.

替换ChannelHandler

replace()方法用于替换原来的handler为新的handler,

```
ChannelPipeline replace(ChannelHandler oldHandler, String newName,  
    ChannelHandler newHandler);  
ChannelHandler replace(String oldName, String newName, ChannelHandler newHandler);  
<T extends ChannelHandler> T replace(Class<T> oldHandlerType, String newName, ChannelHandler newHandler);
```

注意: 如果替换时发现找不到要替换的目标, `replace()`方法会抛出 `NoSuchElementException`.

获取ChannelHandler

```
ChannelHandler first();  
ChannelHandler last();  
ChannelHandler get(String name);  
<T extends ChannelHandler> T get(Class<T> handlerType);
```

注意: 如果pipeline为空, 则上面的方法返回null值.

获取ChannelHandlerContext

```
ChannelHandlerContext firstContext();  
ChannelHandlerContext lastContext();  
ChannelHandlerContext context(ChannelHandler handler);  
ChannelHandlerContext context(String name);  
ChannelHandlerContext context(Class< ? extends ChannelHandler> handlerType);
```

注意: 如果pipeline为空或者pipeline中找不到要求的handler, 则上面的方法返回null值.

其他管理handler的方法

```
List<String> names();  
Map<String, ChannelHandler> toMap();
```

获取Channel

```
Channel channel();
```

注意: 如果当前pipeline 还没有绑定到channel, 则你这里返回null.

fire方法

fire开头的这些方法是给事件通知用的:

```
// channel被注册到eventloop  
ChannelPipeline fireChannelRegistered();  
  
// channel被从eventloop注销  
ChannelPipeline fireChannelUnregistered();  
  
// channel被激活, 通常是连接上了  
ChannelPipeline fireChannelActive();  
  
// channel被闲置, 通常是被关闭  
ChannelPipeline fireChannelInactive();  
  
// channel收到一个Throwable, 比较有意思的是javadoc中明确指出发生的地点  
// 是"in one of its inbound operations"  
ChannelPipeline fireExceptionCaught(Throwable cause);  
  
// channel收到一个用户自定义事件  
ChannelPipeline fireUserEventTriggered(Object event);
```

还有几个方法是给channel读写的:

```
// channel收到信息
ChannelPipeline fireChannelRead(Object msg);

ChannelPipeline fireChannelReadComplete();
ChannelPipeline fireChannelWritabilityChanged();
```

I/O操作方法

```
ChannelFuture bind(SocketAddress localAddress);
ChannelFuture connect(SocketAddress remoteAddress);
ChannelFuture connect(SocketAddress remoteAddress, SocketAddress
    localAddress);
ChannelFuture disconnect();
ChannelFuture close();
ChannelFuture deregister();
```

以及带ChannelPromise的变体版本:

```
ChannelFuture bind(SocketAddress localAddress, ChannelPromise pr
omise);
ChannelFuture connect(SocketAddress remoteAddress, ChannelPromis
e promise);
ChannelFuture connect(SocketAddress remoteAddress, SocketAddress
    localAddress, ChannelPromise promise);
ChannelFuture disconnect(ChannelPromise promise);
ChannelFuture close(ChannelPromise promise);
ChannelFuture deregister(ChannelPromise promise);
```

I/O读写方法


```
ChannelPipeline read();  
ChannelFuture write(Object msg);  
ChannelFuture write(Object msg, ChannelPromise promise);  
ChannelPipeline flush();  
ChannelFuture writeAndFlush(Object msg, ChannelPromise promise);  
ChannelFuture writeAndFlush(Object msg);
```

类DefaultChannelPipeline

类定义

DefaultChannelPipeline实现了ChannelPipeline方法, 注意这个类是包级私有, 而且这个是final类不能继承.

```
final class DefaultChannelPipeline implements ChannelPipeline {  
}
```

继续看具体的代码实现, 代码量有点大.

类属性和初始化

类属性

类DefaultChannelPipeline有以下属性:

```
final AbstractChannel channel;  
  
final AbstractChannelHandlerContext head;  
final AbstractChannelHandlerContext tail;  
  
private final boolean touch = ResourceLeakDetector.getLevel() !=  
    ResourceLeakDetector.Level.DISABLED;  
  
private Map<EventExecutorGroup, ChannelHandlerInvoker> childInvokers;
```

除了childInvokers外其他属性都是final.

构造函数

构造函数只有一个:

```
DefaultChannelPipeline(AbstractChannel channel) {  
    if (channel == null) {  
        throw new NullPointerException("channel");  
    }  
    this.channel = channel;  
  
    tail = new TailContext(this);  
    head = new HeadContext(this);  
  
    head.next = tail;  
    tail.prev = head;  
}
```

这里channel/tail/head都被初始化:

- channel 由参数指定, 不容许为空, final属性表明设置之后不能更改. 这个和ChannelPipeline的设计是一致的: ChannelPipeline 和 channel之间是一一对应的
- tail/head 分别new出了具体的对象, 后面具体看实现代码
- 初始化时head的next指向了tail, tail的prev指向了head, 完成链表初始化

HandlerContext

先看看HandlerContext的代码,

netty javadoc对bytebuffer的定义:

Abstraction of a byte buffer - the fundamental data structure to represent a low-level binary and text message.

byte buffer的抽象 - 用于表述底层二进制和文本信息的基本数据结构.

Netty uses its own buffer API instead of NIO ByteBuffer to represent a sequence of bytes. This approach has significant advantage over using ByteBuffer. Netty's new buffer type, ByteBuf, has been designed from ground up to address the problems of ByteBuffer and to meet the daily needs of network application developers.

Netty 使用自己的 buffer API 替代java NIO的ByteBuffer来表示字节系列.理由自然是觉得JAVA NIO那套ByteBuffer设计的不好, 所以自己整一套更好用的出来.而实际上netty设计的这套ByteBuf的确要强大和实用.

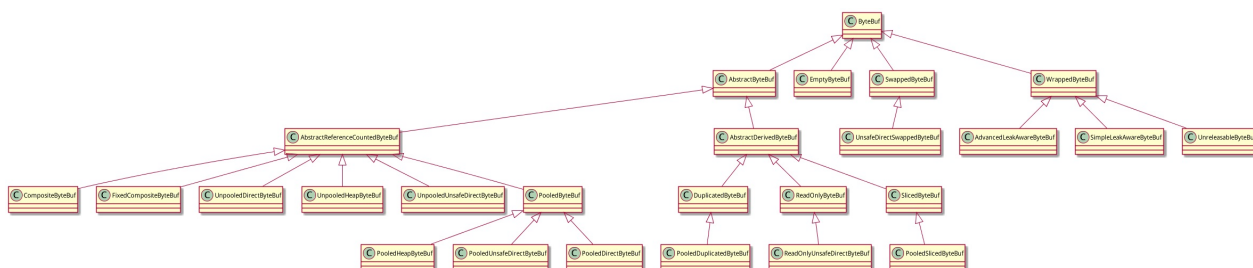
netty列出了认为酷的特性:

- You can define your buffer type if necessary.
- Transparent zero copy is achieved by built-in composite buffer type.
- A dynamic buffer type is provided out-of-the-box, whose capacity is expanded on - demand, just like StringBuffer.
- There's no need to call the flip() method anymore.
- It is often faster than ByteBuffer.

继承结构

ByteBuf全景图

ByteBuf实现类众多,继承层级复杂:



第一眼看上去比较乱,拆分为三个大块后就容易理解:

- ByteBuf基础实现类: ByteBuf的基本实现
- ByteBuf衍生类: 基于Basic ByteBuf衍生的ByteBuf
- 特殊的ByteBuf类: 一些比较特殊的ByteBuf实现

ByteBuf基础实现类

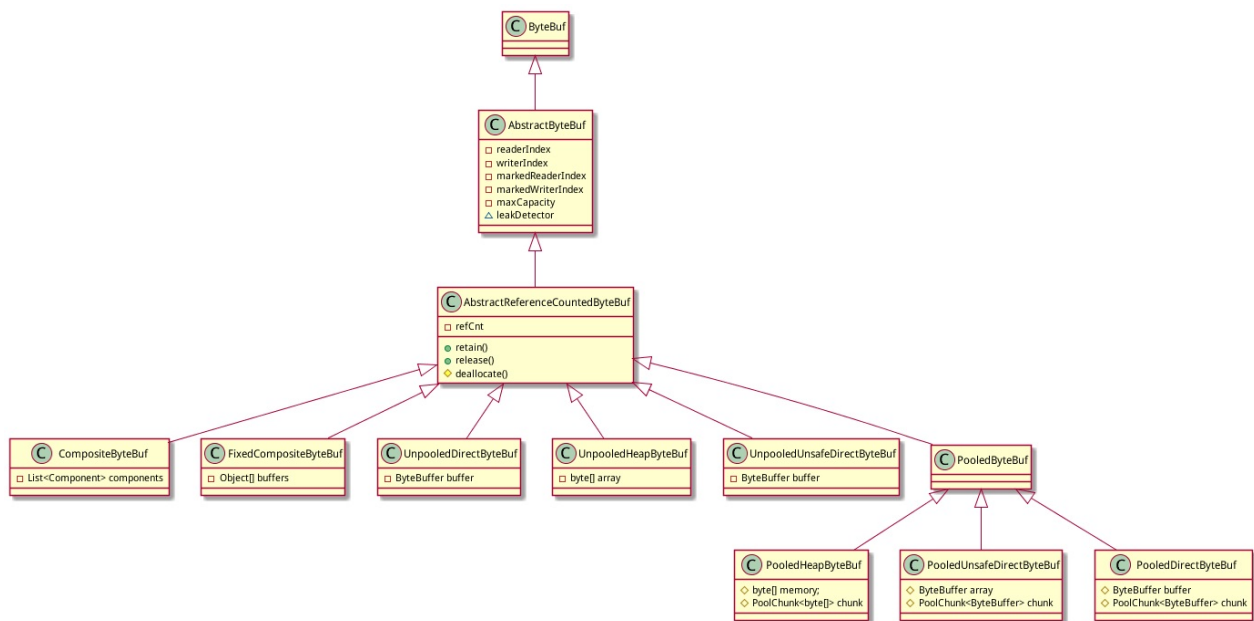
ByteBuf的基础实现类,按照内存分配方式不同分为:

- HeapByteBuf: 直接使用java堆内存
- DirectByteBuf: 使用java nio提供的直接内存,体现为使用nio ByteBuffer
- UnsafeDirectByteBuf: DirectByteBuf的特殊情况,如果当前系统提供了sun.misc.Unsafe

然后按照是否支持池共享又分为:

- PooledByteBuf
- UnpooledByteBuf

注意: PooledByteBuf是4.x之后的新特性, netty3.*和之前版本不支持的。



去掉干扰之后,再看ByteBuf的基本实现就很清晰:

1. 首先是ByteBuf/AbstractByteBuf/AbstractReferenceCountedByteBuf搭建起基础

- ByteBuf: ByteBuf的接口定义 比较奇怪的是netty的作者选择了不定义为interface,而是abstract class然后没有任何接口定义之外的代码
- AbstractByteBuf: 提供了readerIndex/writerIndex/maxCapacity等相关的基础功能,实现了一些通用的方法方便基类使用,定义了一些模板方法. 另外定义了leakDetector.
- AbstractReferenceCountedByteBuf: 为Reference Counted提供了基础实现如refCnt/retain()方法/release()方法,还定义了非常重要的模板方法deallocate()

2. 根据是否支持Pool,区分为Unpooled和Pooled两大阵营

Unpooled阵营:

- UnpooledHeapByteBuf: 提供非池化的堆内存支持, 基于byte[]
- UnpooledDirectByteBuf: 提供非池化的直接内存支持, 基于 NIO Byte Buffer
- UnpooledUnsafeDirectByteBuf: 提供非池化的直接内存支持, 基于 NIO ByteBuffer, **要求sun.misc.Unsafe可用**

Pooled阵营:

- PooledByteBuf: Pooled的基类, 提供Pool的基本实现
- PooledHeapByteBuf: 提供池化的堆内存支持, 基于byte[]
- PooledUnsafeDirectByteBuf: 提供池化的直接内存支持, 基于 NIO ByteBuffer
- PooledDirectByteBuf: 提供池化的直接内存支持, 基于 NIO ByteBuffer, **要求sun.misc.Unsafe可用**

设计上,比较奇怪的是没有定义一个UnpooledByteBuf基类,造成两大阵营的结构不对称,看上去不够美观:)

3. CompositeByteBuf

在Unpooled和Pooled两大阵营之外, CompositeByteBuf/FixedCompositeByteBuf是一个超然的存在.

CompositeByteBuf是一个虚拟的buffer,将多个buffer展现为一个简单合并的buffer,以便实现netty最重要的特性之一: zero copy.后面有详细分析.

FixedCompositeByteBuf功能类似CompositeByteBuf, 以只读的方式包装一个ByteBuf数组,通常用于写入一组ByteBuf的内容.

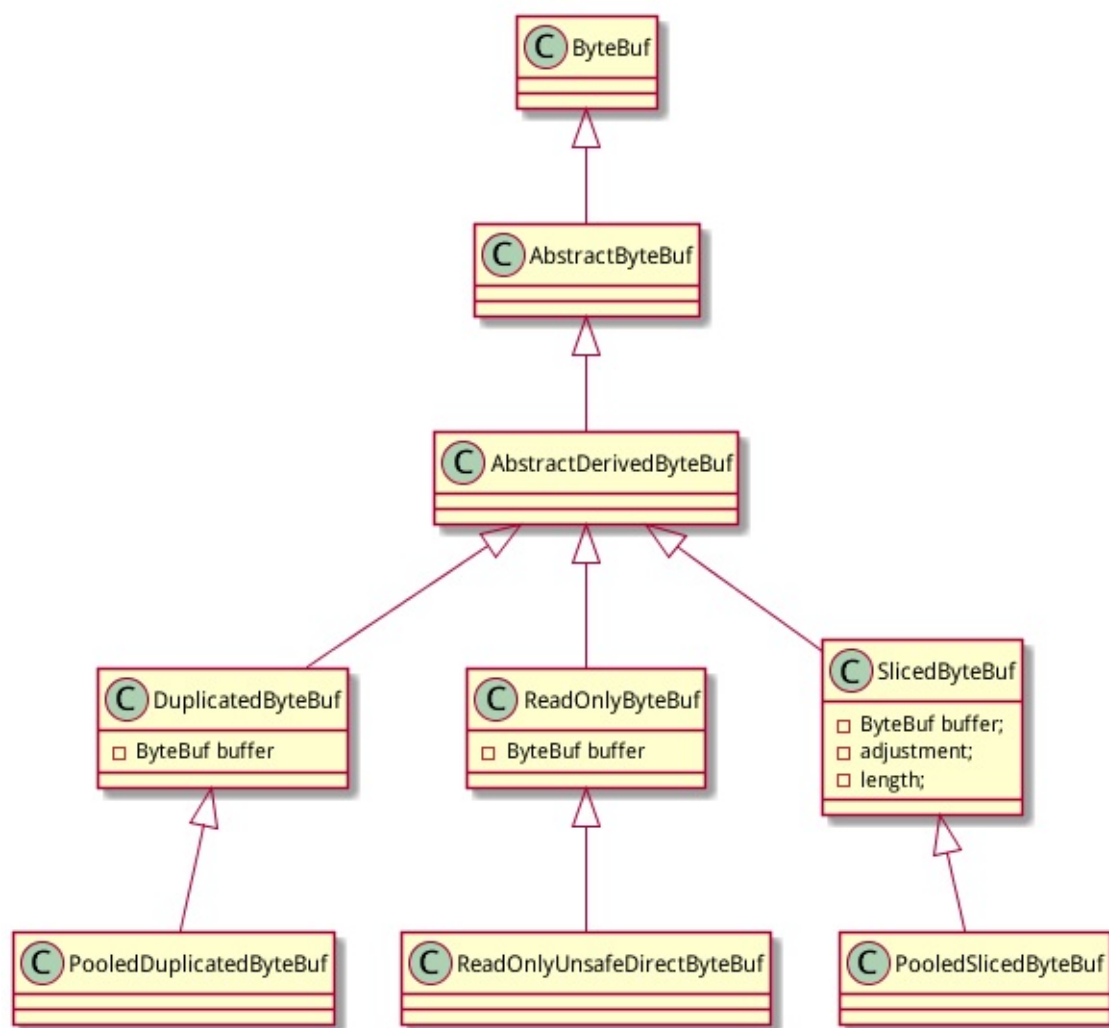
注: 从类的功能和实现上看, 感觉CompositeByteBuf更应该归入后面的ByteBuf衍生类.

ByteBuf衍生类

Derived ByteBuf是在ByteBuf的基本实现基础上衍生出来的: 包装其他ByteBuf,然后增加一些特别的功能.

Derived ByteBuf有抽象类AbstractDerivedByteBuf作为基类. AbstractDerivedByteBuf的javadoc如此描述:

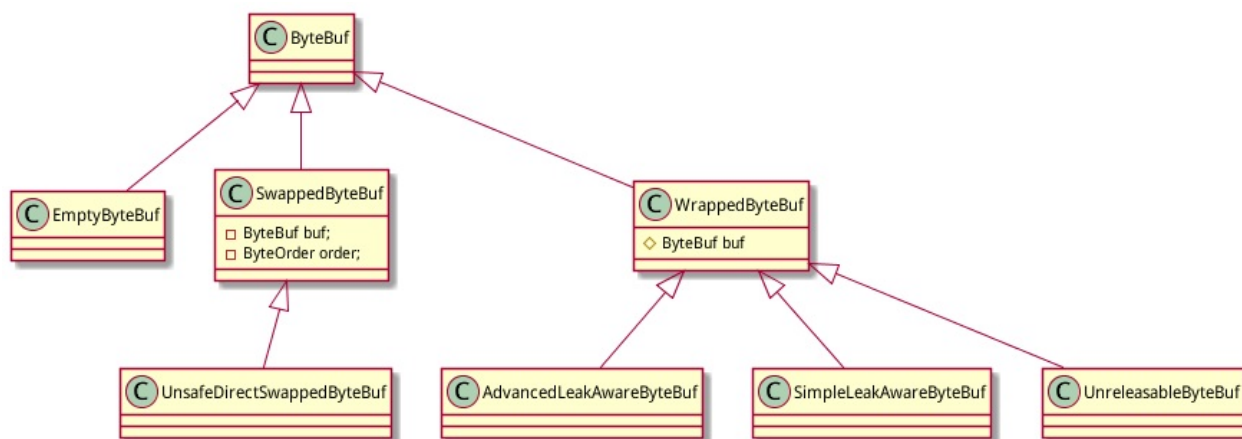
```
Abstract base class for ByteBuf implementations that wrap another ByteBuf.
```



类继承结构如图,主要有:

- DuplicatedByteBuf: 简单的将所有的请求都委托给包装的ByteBuf
- PooledDuplicatedByteBuf: 在DuplicatedByteBuf的基础上提供池的支持
- ReadOnlyByteBuf: 将原有ByteBuf包装为只读的ByteBuf, 所有的写方法都被禁止(抛ReadOnlyBufferException)
- ReadOnlyUnsafeDirectByteBuf: 在ReadOnlyByteBuf的基础上提供对direct ByteBuf的支持,为了得到最佳性能使用unsafe
- SlicedByteBuf: 将原有ByteBuf的部分内容拆分出来的ByteBuf,主要是为了实现zero-copy特性,后面有详细介绍
- PooledSlicedByteBuf: 在SlicedByteBuf的基础上提供池的支持

特殊的ByteBuf类



除了上面介绍的ByteBuf的底层实现类和衍生类之外,netty还提供了3种特殊的ByteBuf:

- EmptyByteBuf: 空的ByteBuf, capacity 和 maximum capacity 都被设置为0.
- SwappedByteBuf: Swap/交换指的是LITTLE_ENDIAN/BIG_ENDIAN之间的交换, SwappedByteBuf用于包装一个ByteBuf,内容相同但是ByteOrder相反.它的子类UnsafeDirectSwappedByteBuf支持memoryAddress.
- WrappedByteBuf: WrappedByteBuf顾名思义是用来包装其他ByteBuf的,代码实现也简单: 包装一个ByteBuf,然后所有方法调用委托给被包装的ByteBuf

WrappedByteBuf的主要作用是作为基类派生出下面几个子类:

- AdvancedLeakAwareByteBuf
- SimpleLeakAwareByteBuf
- UnleasableByteBuf

AdvancedLeakAwareByteBuf和SimpleLeakAwareByteBuf用于resource leak detect. 具体介绍见类[LeakAwareByteBuf](#).

UnleasableByteBuf用于阻止其他对ByteBuf的销毁. 具体介绍见类[UnleasableByteBuf](#).

总结

在所有的ByteBuf类划分为上述3个大块之后, ByteBuf的设计就很清晰了.

后面将通过重点介绍netty提供的几个重要特性来详细展开相关ByteBuf类的具体实现.

ByteBuf 提供了几个非常重要的特性:

- Pooled
- Reference Count
- Zero Copy

后面详细展开.

Jetty 4.x 增加了Pooled Buffer，实现了高性能的buffer池，分配策略则是结合了buddy allocation和slab allocation的jemalloc变种(其实我也不懂...)，代码在io.netty.buffer.PoolArena。

官方说提供了以下优势：

- 频繁分配、释放buffer时减少了GC压力；
- 在初始化新buffer时减少内存带宽消耗（初始化时不可避免的要给buffer数组赋初始值）；
- 及时的释放direct buffer

这块内容太细,代码量太大,暂时还没有深入研读.

自从Netty 4开始，对象的生命周期由它们的引用计数（reference counts）管理，而不是由垃圾收集器（garbage collector）管理。

netty的官方wiki给出了Reference counted的一份介绍资料: [Reference counted objects](#)(也可以看它的[中文翻译版本](#)).

Netty 中通过接口 `ReferenceCounted` 定义了引用计数的基本功能，然后 `ByteBuf` 申明实现了`ReferenceCounted`接口：

```
public abstract class ByteBuf implements ReferenceCounted, Comparable<ByteBuf>
```

先看一下`ReferenceCounted`的定义，和`ByteBuf`是如何实现`ReferenceCounted`的，后面再来看如何使用`Reference Counted`特性。

接口 `ReferenceCounted` 用于实现引用计数,以便显式回收.

定义在 `package io.netty.util` 中:

```
package io.netty.util;

public interface ReferenceCounted {}
```

原理

`ReferenceCounted` 的工作原理: 当引用计数对象被初始化时, 引用计数器从1开始计数. `retain()` 方法增加引用计数, 而 `release()` 方法减少引用计数. 如果引用计数被减到0, 则这个对象就将被显式回收, 而访问被回收的对象将通常会导致访问违规.

如果一个实现引用计数的对象是一个包含其他实现了引用计数的容器, 那么当容器的引用计数变成0时, 被包含的对象也应该通过 `release()` 方法释放一个引用.

方法定义

接口 `ReferenceCounted` 中定义了以下方法:

- `int refCnt()` 返回对象的引用计数. 如果返回 **0**, 意味着对象已经被回收.
- `ReferenceCounted retain()` 将引用计数增加1
- `ReferenceCounted retain(int increment)` 将引用计数增加指定数量
- `boolean release()` 将引用计数减一, 如果引用计数达到0则回收这个对象. 注意: 返回的 **boolean** 值, 当且仅当引用计数变成 **0** 并且这个对象被回收才返回 **true**.
- `boolean release(int decrement)` 同上, 将引用计数减少指定数量
- `ReferenceCounted touch(Object hint)` 出于调试目的, 用一个额外的任意的 (*arbitrary*) 信息记录这个对象的当前访问地址. 如果这个对象被检测到泄露了, 这个操作记录的信息将通过 `ResourceLeakDetector` 提供.
- `ReferenceCounted touch()` 这个方法等价于 `touch(null)`.

注意除了 `refCnt()` 方法之外, 其他的几个方法都是返回 `ReferenceCounted` 对象. 实现中一般时返回当前对象本身, 以便实现链式 (*train*) 调用.

类定义

AbstractReferenceCountedByteBuf的类定义:

```
public abstract class AbstractReferenceCountedByteBuf extends AbstractByteBuf

public abstract class AbstractByteBuf extends ByteBuf

public abstract class ByteBuf implements ReferenceCounted, Comparable<ByteBuf>
```

ByteBuf声明了对接口ReferenceCounted的实现,但是实际的代码支持在AbstractReferenceCountedByteBuf中.

代码分析

类AbstractReferenceCountedByteBuf中主要是实现了ByteBuf对接口ReferenceCounted的支持,没有其他代码.

refCntUpdater

静态变量refCntUpdater和静态初始化代码:


```
private static final AtomicIntegerFieldUpdater<AbstractReference
CountedByteBuf> refCntUpdater;

static {
    AtomicIntegerFieldUpdater<AbstractReferenceCountedByteBuf> u
pdater =
        PlatformDependent.newAtomicIntegerFieldUpdater(Abstr
actReferenceCountedByteBuf.class, "refCnt");
    if (updater == null) {
        updater = AtomicIntegerFieldUpdater.newUpdater(Abstrac
tReferenceCountedByteBuf.class, "refCnt");
    }
    refCntUpdater = updater;
}
```

在类装载时初始化`refCntUpdater`以备用. 使用到`PlatformDependent`类(这块细节不展开).

refCnt

类属性`refCnt`的定义, 注意初始化值为1:

```
private volatile int refCnt = 1;
```

关键字`volatile`使得对`refCnt` 的访问都会基于主内存.`refCnt()`方法就只是简单返回这个`refCnt`属性.

```
public final int refCnt() {
    return refCnt;
}
```

retain() 方法

ReferenceCounted的工作原理: 当引用计数对象被初始化时, 引用计数器从1开始计数. **retain()**方法增加引用计数, 而**release()**方法减少引用计数. 如果引用计数被减到0, 则这个对象就将被显式回收, 而访问被回收的对象将通常会导致访问违规.

前面看到**refCnt**初始化值为1, 和上面的原理一致. 继续看**retain()**方法的实现:

```
@Override
public ByteBuf retain() {
    for (;;) {
        int refCnt = this.refCnt;
        if (refCnt == 0) {
            throw new IllegalReferenceCountException(0, 1);
        }
        if (refCnt == Integer.MAX_VALUE) {
            throw new IllegalReferenceCountException(Integer.MAX
_VALUE, 1);
        }
        if (refCntUpdater.compareAndSet(this, refCnt, refCnt + 1
)) {
            break;
        }
    }
    return this;
}
```

核心代码是for循环加**compareAndSet()**方法, 通过CAS机制在不加锁的情况下实现了对**refCnt**变量的并发更新, "**refCnt + 1**"表明计数器加1:

```
    for (;;) {
        int refCnt = this.refCnt;
        if (refCntUpdater.compareAndSet(this, refCnt, refCnt + 1
)) {
            break;
        }
    }
```

"**refCnt == 0**"的检测则是遵循原理中的要求: 访问被回收的对象将通常会导致访问违规.

`retain(int increment)` 方法的实现和`retain()`几乎完全相同,只是`increment`不是固定为1而已. 有点不太理解的是`netty`为什么不直接通过在`retain()`中调用`retain(1)`的方式来实现,我能想到的理由就是`retain()`的调用非常频繁(远远超过`retain(int increment)`),因此现在的做法可以带来少许的性能提升?

release() 方法

`release()`方法的代码实现:

```
@Override
public final boolean release() {
    for (;;) {
        int refCnt = this.refCnt;
        if (refCnt == 0) {
            throw new IllegalReferenceCountException(0, -1);
        }

        if (refCntUpdater.compareAndSet(this, refCnt, refCnt - 1
    )) {
            if (refCnt == 1) {
                deallocate();
                return true;
            }
            return false;
        }
    }
}
```

代码核心有两个功能:

1. 通过CAS实现`refCnt`减一,方式和`retain()`里面是一样的
2. 判断`refCnt`,如果`==1`,意味着这次`release()`调用之后计数器会被置零.按照原理要求,"如果引用计数被减到0,则这个对象就将被显式回收".因此调用`deallocate()`方法显式回收对象

返回值只有在`deallocate()`时才`return true`,其他情况都是`return false`.

"refCnt == 0"的检测同样时遵循原理的要求: 访问被回收的对象将通常会导致访问违规.

然后看`deallocate()`,只是一个`protected abstract`的模板方法, 子类自行实现.

```
/**
 * Called once {@link #refCnt()} is equals 0.
 */
protected abstract void deallocate();
```

`release(int decrement)`方法实现基本相同.

touch()方法

`AbstractReferenceCountedByteBuf`中提供了`touch()`方法的两个空实现, 基本没有任何实际意义:

```
@Override
public ByteBuf touch() {
    return this;
}

@Override
public ByteBuf touch(Object hint) {
    return this;
}
```

找了一下子类,也没有发现有其他的具体实现,这块有点糊涂(难道netty还没有完成这个touch机制的实现?).

ReferenceCounted的设计和ByteBuf的实现(AbstractReferenceCountedByteBuf)都不复杂。

但是对于如何使用ByteBuf, 在使用时候应该调用remain()/release()依然是个非常重要的话题。

原则

究竟应该谁, 在申明时候来销毁ByteBuf?

[netty的wiki](#)中给出建议:

通常的经验是谁最后访问引用计数对象, 谁就负责销毁它

具体来说有以下两点:

- 如果组件A把一个引用计数对象传给另一个组件B, 那么组件A通常不需要销毁对象, 而是把决定权交给组件B
- 如果一个组件不再访问一个引用计数对象, 那么这个组件负责销毁它

做法

看了一下[netty中的例子](#), 对照上面的原则和做法, 总结如下.

传递buffer对象不返回

对于A调用B的情况, 如果B的函数不会返回Buffer对象, 代码类似如下:

```
void a() {
    ByteBuf buffer = createBuffer();
    b(buffer);
}

void b(ByteBuf buffer) {}
```

会有以下几种情况:

1. 如果A传递buffer对象到B之后,A就放弃对buffer的控制

之后只有B才控制(或者说访问)buffer,那么此时B就负有在使用完成之后销毁buffer的责任.

```
void a() {  
    ByteBuf buffer = createBuffer();  
    b(buffer);  
}  
  
void b(ByteBuf buffer) {  
    try {  
        // handle buffer  
    } finally {  
        buffer.release();  
    }  
}
```

因此b需要遵循的原则: 如果有buffer传入,在使用完成之后需要销毁.

2. 如果A传递buffer对象到B之后,A并没有放弃对buffer的控制

但是b是无法区分,b在被调用时是不知道a是否要放弃控制权,因此b只能继续遵循上述的原则做一次release.

此时a是知道自己的行为,因此a可以在调用b之前,先行调用一次retain()方法. 这样a就可以在调用b之后继续保留对buffer的访问.

```
void a() {
    ByteBuf buffer = createBuffer();
    buffer.remain();
    try {
        b(buffer);
        // a continue to use buffer
    } finally {
        buffer.release();
    }
}

void b(ByteBuf buffer) {
    try {
        // handle buffer
    } finally {
        buffer.release();
    }
}
```

传递**buffer**对象又被返回

对于A调用B的情况,如果B的函数会返回Buffer对象,代码类似如下:

```
void a() {
    ByteBuf buffer = createBuffer();
    buffer = b(buffer);
    // continue to access buffer
}

ByteBuf b(ByteBuf buffer) {}
```

会有以下几种情况:

1. B在处理完成之后又将buffer原样返回,此时B不再控制buffer,而A在获取返回的buffer之后需要继续访问buffer,因此A才是最后的访问者,因此这时是A需要担负销毁buffer的责任

```
void a() {
    ByteBuf buffer = createBuffer();
    try {
        buffer = b(buffer);
        // continue to access buffer
    } finally {
        buffer.release();
    }
}

ByteBuf b(ByteBuf buffer) {
    // handle buffer
    return buffer;
}
```

2. B在处理传入的buffer之后,又重新生成一个新的ByteBuf返回.此时对于A时无法判断返回的buffer对象和上面情况有什么不同,因此A只能继续遵循上述的原则对返回的ByteBuf做release(). 而B应该对输入的ByteBuf做release().

```
void a() {
    ByteBuf buffer = createBuffer();
    try {
        buffer = b(buffer);
        // continue to access buffer
    } finally {
        buffer.release();
    }
}

ByteBuf b(ByteBuf buffer) {
    try {
        // handle buffer
    } finally {
        buffer.release();
    }
    ByteBuf newBuffer = createBuffer();
    return newBuffer;
}
```


特殊场景

阻止销毁

某些情况下,我们会在将ByteBuf传递出去之后阻止其他人对这个ByteBuf做销毁操作.

netty为此提供了一个特别的类UnreleasableByteBuf.

UnreleasableByteBuf 用于阻止他人对目标ByteBuf的销毁.

实现方式

在构造函数中传入需要包裹的ByteBuf:

```
final class UnreleasableByteBuf extends WrappedByteBuf {  
  
    UnreleasableByteBuf(ByteBuf buf) {  
        super(buf);  
    }  
}
```

然后覆盖retain()/release()方法,不做任何操作,只是简单的返回false:

```
@Override  
public ByteBuf retain(int increment) {  
    return this;  
}  
  
@Override  
public ByteBuf retain() {  
    return this;  
}  
  
@Override  
public boolean release() {  
    return false;  
}  
  
@Override  
public boolean release(int decrement) {  
    return false;  
}
```

再覆盖slice()/readSlice()/duplicate()方法,将需要返回的ByteBuf再次包装为UnreleasableByteBuf:

```
@Override
public ByteBuf readSlice(int length) {
    return new UnreleasableByteBuf(buf.readSlice(length));
}

@Override
public ByteBuf slice() {
    return new UnreleasableByteBuf(buf.slice());
}

@Override
public ByteBuf slice(int index, int length) {
    return new UnreleasableByteBuf(buf.slice(index, length));
}

@Override
public ByteBuf duplicate() {
    return new UnreleasableByteBuf(buf.duplicate());
}
```

使用方式

Unpooled中提供unreleasableBuffer()工具方法,代码够简单的:

```
public static ByteBuf unreleasableBuffer(ByteBuf buf) {
    return new UnreleasableByteBuf(buf);
}
```

一般的使用场景就是定义特殊的常量ByteBuf,然后包装成unreleasableBuffer()后就不怕被其他人错误的销毁掉:

```
public abstract class HttpObjectEncoder<H extends HttpMessage> extends MessageToMessageEncoder<Object> {
    private static final ByteBuf CRLF_BUF = unreleasableBuffer(directBuffer(CRLF.length).writeBytes(CRLF));
}
```


考虑到buffer释放的复杂性,理论上总是有可能因为某些失误导致有buffer没有正确释放从而发生内存泄露的可能.

为此netty提供了资源泄露检测机制,在AbstractByteBuf中提供了一个ResourceLeakDetector的实例leakDetector来做资源泄露检测.

```
public abstract class AbstractByteBuf extends ByteBuf {  
    static final ResourceLeakDetector<ByteBuf> leakDetector = new  
        ResourceLeakDetector<ByteBuf>(ByteBuf.class);  
}
```

工作原理

为了检测资源是否泄露,netty中使用了PhantomReference(虚引用)和ReferenceQueue(引用队列).

工作原理如下:

1. 根据检测级别和采样率的设置,在需要时为需要检测的ByteBuf创建PhantomReference
2. 当JVM回收掉ByteBuf对象时,JVM会将PhantomReference放入ReferenceQueue
3. 通过对ReferenceQueue中PhantomReference的检查,判断在GC前是否有释放ByteBuf的资源,就可以知道是否有资源释放

实现机制

理论上,ByteBuf应该做两件事情:

1. 告之leakDetector需要监控的所有对象 在创建时应该调用leakDetector.open()方法,传入自身(也就是this)的引用.这样leakDetector就可以通过PhantomReference + ReferenceQueue 的机制来监控这些对象. 为了后面能调用close方法,这里需要保存返回的ResourceLeak对象.
2. 当被监控的对象正常释放时告之leakDetector 在ByteBuf回收时,应该调用ResourceLeak.close()方法来告之.

以类CompositeByteBuf为例:

```
public class CompositeByteBuf extends AbstractReferenceCountedByteBuf implements Iterable<ByteBuf> {
    private final ResourceLeak leak;

    public CompositeByteBuf(ByteBufAllocator alloc, boolean direct, int maxNumComponents) {
        .....
        leak = leakDetector.open(this);
    }
}

protected void deallocate() {
    .....

    if (leak != null) {
        leak.close();
    }
}
```

代码实现

open()

ResourceLeakDetector的open()方法生成leakDetector对象:

```
public final class ResourceLeakDetector<T> {
    public ResourceLeak open(T obj) {
        .....
        return new DefaultResourceLeak(obj);
    }
}
```

这个leakDetector对象实际是一个PhantomReference, 注意构造函数的实现,refQueue是前面所说的ReferenceQueue:

```
private final class DefaultResourceLeak extends PhantomReference<
Object> implements ResourceLeak {
    DefaultResourceLeak(Object referent) {
        super(referent, referent != null? refQueue : null);
    }
}
```

close()

ResourceLeakDetector的close()方法,根据内部的freed属性来判断是否是第一次调用,然后返回true/false:

```
public boolean close() {
    if (freed.compareAndSet(false, true)) {
        .....
        return true;
    }
    return false;
}
```

判断的基本原则:

1. 如果ByteBuf在GC前被正常释放,那么就会正常调用close()方法,这样下一次再调用close时会返回false
2. 如果ByteBuf被GC前未能正常释放, close()方法没能及时调用,那么再调用close()时会返回true

reportLeak()

检测的方式就是从refQueue中获取已经被GC了的DefaultResourceLeak对象,然后按照上面的逻辑调用close()方法做检测:

```
private void reportLeak(Level level) {
    for (;;) {
        DefaultResourceLeak ref = (DefaultResourceLeak) refQueue
            .poll();
        if (ref == null) {
            break;
        }

        ref.clear();

        if (!ref.close()) {
            // 返回false, 说明之前已经调用郭, 资源被正常释放
            continue;
        }

        // 发现泄露了, 在这里报错!
    }
}
```

详尽的代码分析, 请见下节.

netty在package util中实现了类ResourceLeakDetector：

```
package io.netty.util;  
public final class ResourceLeakDetector<T> {
```

基本概念

Detect Level

ResourceLeakDetector.Level定义了资源泄露检测的等级：

- **DISABLED**：禁用，不做检测
- **SIMPLE**：开启资源泄露检测的简化采样，仅仅报告是否存在泄露，开销小。这个是默认级别。
- **ADVANCED**：开始资源泄露检测的高级采样，报告被泄露的对象最后一次访问的地址，消耗高
- **PARANOID**：开始资源泄露检测的偏执采样，报告被泄露的对象最后一次访问的地址，消耗最高(仅适用于测试)

类ResourceLeakDetector被classloader装载时，会执行下面的静态初始化代码：

```
private static final String PROP_LEVEL = "io.netty.leakDetectionLevel";
private static final Level DEFAULT_LEVEL = Level.SIMPLE;

private static Level level;

static {
    String levelStr = SystemPropertyUtil.get(PROP_LEVEL, DEFAULT_LEVEL.name()).trim().toUpperCase();
    Level level = DEFAULT_LEVEL;
    for (Level l: EnumSet.allOf(Level.class)) {
        if (levelStr.equals(l.name()) || levelStr.equals(String.valueOf(l.ordinal()))) {
            level = l;
        }
    }

    ResourceLeakDetector.level = level;
    if (logger.isDebugEnabled()) {
        logger.debug("-D{: {}}", PROP_LEVEL, level.name().toLowerCase());
    }
}
```

读取系统配置`io.netty.leakDetectionLevel`，如果没有设置或者设置不正确则取默认值 `Level.SIMPLE`。

Sample Interval

采样间隔在初始化时指定，注意`samplingInterval`是`final`，设定后不可修改。

```
private final int samplingInterval;

public ResourceLeakDetector(String resourceType, int samplingInterval, long maxActive) {
    .....
    this.samplingInterval = samplingInterval;
}
```

如果不指定，则取默认值113：

```
private static final int DEFAULT_SAMPLING_INTERVAL = 113;

public ResourceLeakDetector(String resourceType) {
    this(resourceType, DEFAULT_SAMPLING_INTERVAL, Long.MAX_VALUE);
}
```

samplingInterval的使用和leakDetectionLevel直接相关：

- 如果leakDetectionLevel被设置为PARANOID，则每次都采样，samplingInterval设置失效
- 如果leakDetectionLevel被设置为DISABLED，则禁用采样，samplingInterval设置失效
- 当leakDetectionLevel被设置为SIMPLE或者ADVANCED时，samplingInterval设置才生效

看open()方法中的代码实现，先忽略其他细节，只看level和samplingInterval的部分：

```
public ResourceLeak open(T obj) {
    Level level = ResourceLeakDetector.level;
    if (level == Level.DISABLED) {
        return null;
    }

    if (level.ordinal() < Level.PARANOID.ordinal()) {
        if (leakCheckCnt ++ % samplingInterval == 0) {
            reportLeak(level);
            return new DefaultResourceLeak(obj);
        } else {
            return null;
        }
    } else {
        reportLeak(level);
        return new DefaultResourceLeak(obj);
    }
}
```

可以看到这里的处理逻辑是：

1. 当"level == Level.DISABLED" 时，return null表示不采样
2. 当"level.ordinal() < Level.PARANOID"不成立时，实际就是level等于PARANOID时，每次都取样
3. 当"level.ordinal() < Level.PARANOID" 时, 则根据当前计数器leakCheckCnt的值对samplingInterval取余数判断是否为0来决定是否取样

考虑默认情况samplingInterval=113，采样概率为1/113 大概等于9/1000，或者近似于1%.

leakCheckCnt

leakCheckCnt用于计数，配合samplingInterval来判断是否满足取样的条件。

```
private long leakCheckCnt;

public ResourceLeak open(T obj) {
    if (leakCheckCnt ++ % samplingInterval == 0) {}
}
```

resourceType

resourceType在构造函数中指定，依然是final设置后不可修改：

```
private final String resourceType;

public ResourceLeakDetector(String resourceType, int samplingInterval, long maxActive) {
    if (resourceType == null) {
        throw new NullPointerException("resourceType");
    }
    this.resourceType = resourceType;
}
```

resourceType可以通过String类型来指定，也可以通过resource类的Class来设置：

```
public ResourceLeakDetector(Class<?> resourceType) {
    this(simpleClassName(resourceType));
}

public ResourceLeakDetector(String resourceType){}
```

通过Class指定时会通过调用simpleClassName()方法去掉package名而获取简单类名。

这里貌似有个隐患：如果有两个同名而package不同的resource，会被视为同一个。

resourceType没有涉及到任何具体的实现逻辑，仅仅是在打印日志的时候作为一个标识符输出。

active

和**active**概念相关的有三个变量：

```
private final long maxActive;
private long active;
private final AtomicBoolean loggedTooManyActive = new AtomicBoolean();
```

- **maxActive**：容许的**active**最大数量，在初始化时指定，如果不指定则默认为 `Long.MAX_VALUE`
- **active**：当前**active**的数量
- **loggedTooManyActive**：是否已经打印太多**active**实例的标识符

active默认为0，在**DefaultResourceLeak**构造函数中+1，在**close()**中-1（细节先忽略）：

```
DefaultResourceLeak(Object referent) {
    .....
    active ++;
}

public boolean close() {
    .....
    active --;
}
```

然后在**reportLeak()**方法中检测当前**active**：

```
private void reportLeak(Level level) {
    .....
    // Report too many instances.
    int samplingInterval = level == Level.PARANOID? 1 : this.samplingInterval;
    if (active * samplingInterval > maxActive && loggedTooManyActive.compareAndSet(false, true)) {
        logger.error("LEAK: You are creating too many " + resourceType + " instances. " +
            resourceType + " is a shared resource that must be reused across the JVM," +
            "so that only a few instances are created.");
    }
}
```

如果`active * samplingInterval > maxActive`成立，则以CAS的方式设置`loggedTooManyActive`为`true`，如果成功，则打印错误信息，提示当前`resourceType`类型的资源已经创建了太多的实例。

注意这个日志打印的操作只会触发一次，后续即使满足`active * samplingInterval > maxActive`，`loggedTooManyActive`因为已经被设置为`true`了，所以`loggedTooManyActive.compareAndSet(false, true)`必然失败返回`false`从而`if`条件不满足。

内部辅助类

interface ResourceLeak

接口`ResourceLeak`定于在`package io.netty.util`中。

```
package io.netty.util;
public interface ResourceLeak {}
```

方法如下：

- `void record(Object hint)`：记录调用者当前的`stack trace`和指定的额外的任意的(`arbitrary`)信息以便`ResourceLeakDetector`能告诉被泄露的资源最后一次

访问是在哪里。

- `void record()`：等同于`record(null)`
- `boolean close()`：关闭当前leak，这样`ResourceLeakDetector`就不再担心资源泄露

class DefaultResourceLeak

`DefaultResourceLeak`类实现`ResourceLeak`接口，继承自`PhantomReference`：

```
private final class DefaultResourceLeak extends PhantomReference<
Object> implements ResourceLeak {}
```

creationRecord

`creationRecord`属性用来记录`DefaultResourceLeak`对象初始化时的记录，只有当**referent**不等于**null**并且**level**大于或等于**Level.ADVANCED**时才做这个记录。

```
private final String creationRecord;

DefaultResourceLeak(Object referent) {
    .....
    if (referent != null) {
        Level level = getLevel();
        if (level.ordinal() >= Level.ADVANCED.ordinal()) {
            creationRecord = newRecord(null, 3);
        } else {
            creationRecord = null;
        }
        .....
    } else {
        .....
        creationRecord = null;
    }
}
```


看newRecord()函数的实现，会发现为了得到stack trace，采用了一个非常规的动作"new Throwable()":

```
static String newRecord(Object hint, int recordsToSkip) {  
    .....  
    // Append the stack trace.  
    StackTraceElement[] array = new Throwable().getStackTrace();  
    for (StackTraceElement e: array) {  
        .....  
    }  
}
```

在运行时创建Throwable对象是一个开销非常巨大的操作，因此这个操作对性能影响非常巨大。

lastRecords

lastRecords记录最后几次访问记录。初始化为空，之后在record方法被调用时开始记录信息：

```

private final Deque<String> lastRecords = new ArrayDeque<String>
();

private void record0(Object hint, int recordsToSkip) {
    if (creationRecord != null) {
        String value = newRecord(hint, recordsToSkip);

        synchronized (lastRecords) {
            int size = lastRecords.size();
            if (size == 0 || !lastRecords.getLast().equals(value
)) {
                lastRecords.add(value);
            }
            if (size > MAX_RECORDS) {
                lastRecords.removeFirst();
            }
        }
    }
}

```

注意"creationRecord != null"的判断，考虑creationRecord是final，前面看到creationRecord属性只有当referent不等于null并且level大于或等于**Level.ADVANCED**时才做记录。因此这里lastRecords的记录实际上也是遵循上面的判断条件。

记录时同样调用newRecord()，里面依然是一次"new Throwable()"的开销。

MAX_RECORDS限制了最大记录数量，默认时4，看代码中没有能设置的地方，而且也是final，因此次数就被固定为4了：

```

private static final int MAX_RECORDS = 4;

```

freed

freed在初始化时赋值,如果referent为空,则初始值时true.

```
private final AtomicBoolean freed;

DefaultResourceLeak(Object referent) {
    .....
    if (referent != null) {
        .....
        freed = new AtomicBoolean();
    } else {
        .....
        freed = new AtomicBoolean(true);
    }
}
```

唯一使用`freed`属性的地方是`close()`函数:

```
public boolean close() {
    if (freed.compareAndSet(false, true)) {
        .....
        return true;
    }
    return false;
}
```

只有当构造函数中`referent`不为`null`时, `freed`的初始值为`false`,才能让这里的"`freed.compareAndSet(false, true)`"返回`true`.

linked list of active resources

`DefaultResourceLeak`内部有两个引用`prev/next`, 分别指向上一个和下一个. 这样就将所有`DefaultResourceLeak`都链接起来.

```
public final class ResourceLeakDetector<T> {
    /** the linked list of active resources */
    private final DefaultResourceLeak head = new DefaultResourceLeak(null);
    private final DefaultResourceLeak tail = new DefaultResourceLeak(null);

    private final class DefaultResourceLeak {
        private DefaultResourceLeak prev;
        private DefaultResourceLeak next;
    }
}
```

而在类ResourceLeakDetector中,还保存有两个指向active resources 链接列表的头和尾的引用head/tail.在ResourceLeakDetector的构造函数中,将head.next指向tail,将tail.prev指向head:

```
public ResourceLeakDetector(String resourceType, int samplingInterval, long maxActive) {
    .....

    head.next = tail;
    tail.prev = head;
}
```

在DefaultResourceLeak的构造函数中,如果"referent != null",则需要在原来的head和head.next之间插入当前实例:

```
"head -- head.next"    >>> "head    -- this -- head.next"
```

然后需要分别调整head/this/head.next的引用

```

DefaultResourceLeak(Object referent) {
    .....

    if (referent != null) {
        .....
        synchronized (head) {
            prev = head;                //将this.prev指向原来的head

            next = head.next;           //将this.next指向原来的head.next
            head.next.prev = this;      //将head.next对象的prev
            //从指向原来的head变成指向this
            head.next = this;          //将head的next指向this
        }
        .....
    } else {
        .....
    }
}

```

考虑到`head`和`tail`只是两个虚拟的引用,其实上面的操作相当于在linked list的最前面增加了一个元素.

在`close()`方法中,需要将当前对象从链中摘出来:

```
"head -- this -- head.next" >>> "head -- head.next"
```

代码类似:

```
public boolean close() {
    if (freed.compareAndSet(false, true)) {
        synchronized (head) {
            prev.next = next;           //将prev的next从指向this
            修改为指向this.next
            next.prev = prev;           //将next的prev从指向this
            修改为指向this.prev
            prev = null;                 //清理this.prev
            next = null;                 //清理this.next
        }
    }
}
```

核心代码

完成上面的代码分析之后,我们来看最核心的代码逻辑:

```
private final ReferenceQueue<Object> refQueue = new ReferenceQueue<Object>();
```

这里定义了refQueue作为PhantomReference的ReferenceQueue,还记得DefaultResourceLeak类继承PhantomReference吗?

ResourceLeakDetector.open()方法

open()方法用户创建一个新的ResourceLeak对象,当相关的资源被回收时期望这个ResourceLeak对象会通过调用close()方法来关闭.

```

public ResourceLeak open(T obj) {
    Level level = ResourceLeakDetector.level;
    if (level == Level.DISABLED) {
        return null;
    }

    if (level.ordinal() < Level.PARANOID.ordinal()) {
        if (leakCheckCnt ++ % samplingInterval == 0) {
            reportLeak(level);
            return new DefaultResourceLeak(obj);
        } else {
            return null;
        }
    } else {
        reportLeak(level);
        return new DefaultResourceLeak(obj);
    }
}

```

如果满足level和samplingInterval的要求,则开始采样.open()方法就会返回一个DefaultResourceLeak对象,包含传入的需要被监测的资源对象.

注意"new DefaultResourceLeak(obj)"的调用中, DefaultResourceLeak的构造函数会调用super(也就是PhantomReference)的构造函数,传入要引用的对象和ReferenceQueue:

```

DefaultResourceLeak(Object referent) {
    super(referent, referent != null? refQueue : null);
    .....
}

```

这样就完成了对需要检测的对象的监控,之后如果被检测的引用对象被JVM回收,则PhantomReference会被放入ReferenceQueue(也就是refQueue).

DefaultResourceLeak.close() 方法

如果`close()`被正常调用,那么`freed.compareAndSet(false, true)`就会成立,最后返回`true`.如果是第二次调用,则会返回`false`.因此可以通过调用`close()`方法然后判断返回值来看是否有资源泄露.

```
public boolean close() {
    if (freed.compareAndSet(false, true)) {
        .....
        return true;
    }
    return false;
}
```

ResourceLeakDetector.reportLeak() 方法

在`ResourceLeakDetector.open()`方法当中,如果需要采样,则会先调用一次`reportLeak()`方法来判断之前检测的对象是否有出现资源泄露的情况.

```
private void reportLeak(Level level) {
    if (!logger.isErrorEnabled()) {
        for (;;) {
            @SuppressWarnings("unchecked")
            DefaultResourceLeak ref = (DefaultResourceLeak) refQueue.poll();
            if (ref == null) {
                break;
            }
            ref.close();
        }
        return;
    }
    .....
}
```

这段代码先判断是否容许输出错误日志,如果不被容许,那么也就没有机会输出内存泄露的信息了,所有没有必要做后面的真实检测,直接清理`refQueue`中的数据.

如果容许输出错误日志,则继续检测流程. 先判断一下是否当前有太多的实例了, 注意只是实例太多,实例太多不代表一定是内存泄露.但太多超过了`maxActive`的限制,就应该输出对应的错误信息:

```
// Report too many instances.
int samplingInterval = level == Level.PARANOID? 1 : this.samplingInterval;
if (active * samplingInterval > maxActive && loggedTooManyActive
    .compareAndSet(false, true)) {
    logger.error("LEAK: You are creating too many " + resourceType
        + " instances.  " +
            resourceType + " is a shared resource that must be reused across the JVM," +
            "so that only a few instances are created.");
}
```

然后才是最核心的检测代码: 从`refQueue`中循环`poll`数据,然后调用`ref.close()`. 如果返回`false`,说明之前已经被正常调用一次`close()`,资源没有泄露.

```
// Detect and report previous leaks.
for (;;) {
    @SuppressWarnings("unchecked")
    DefaultResourceLeak ref = (DefaultResourceLeak) refQueue.poll();
    if (ref == null) {
        break;
    }

    ref.clear();

    if (!ref.close()) {
        continue;
    }

    .....
}
```

如果`ref.close()`返回`true`,则说明之前`close()`方法没有被调用.这里就需要汇报资源泄露的信息了:

```
String records = ref.toString();
if (reportedLeaks.putIfAbsent(records, Boolean.TRUE) == null) {
    if (records.isEmpty()) {
        logger.error("LEAK: {}.release() was not called before i
t's garbage-collected. " +
            "Enable advanced leak reporting to find out wher
e the leak occurred. " +
            "To enable advanced leak reporting, " +
            "specify the JVM option '-D{}={}' or call {}.set
Level() " +
            "See http://netty.io/wiki/reference-counted-obje
cts.html for more information.",
            resourceType, PROP_LEVEL, Level.ADVANCED.name().
toLowerCase(), simpleClassName(this));
    } else {
        logger.error(
            "LEAK: {}.release() was not called before it's g
arbage-collected. " +
            "See http://netty.io/wiki/reference-counted-obje
cts.html for more information.{}",
            resourceType, records);
    }
}
```

netty中为了方便监控ByteBuf的泄露,实现了两个LeakAwareByteBuf:

- AdvancedLeakAwareByteBuf
- SimpleLeakAwareByteBuf

调用方式

AdvancedLeakAwareByteBuf和SimpleLeakAwareByteBuf和resource leak detect 相关,在方法 AbstractByteBufAllocator.toLeakAwareBuffer()中被调用,分别对应 resource leak detect的不同级别:

```
protected static ByteBuf toLeakAwareBuffer(ByteBuf buf) {
    ResourceLeak leak;
    switch (ResourceLeakDetector.getLevel()) {
        case SIMPLE:
            leak = AbstractByteBuf.leakDetector.open(buf);
            if (leak != null) {
                buf = new SimpleLeakAwareByteBuf(buf, leak);
            }
            break;
        case ADVANCED:
        case PARANOID:
            leak = AbstractByteBuf.leakDetector.open(buf);
            if (leak != null) {
                buf = new AdvancedLeakAwareByteBuf(buf, leak);
            }
            break;
        default:
            break;
    }
    return buf;
}
```

toLeakAwareBuffer()方法在以下几个地方被调用,都是用来将普通ByteBuf包装为LeakAwareBuffer.

- PooledByteBufAllocator.newDirectBuffer()
- PooledByteBufAllocator.newHeapBuffer()

- `UnpooledByteBufAllocator.newDirectBuffer()`

代码类似如下:

```
protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {  
    // 创建目标ByteBuf  
    ByteBuf buf = .....;  
    // 包装为LeakAwareBuffer  
    return toLeakAwareBuffer(buf);  
}
```

总结: 在使用**ByteBufAllocator**分配**ByteBuf**时, **netty**会根据当前的**resource detect level**的设置使用对应的**AdvancedLeakAwareByteBuf**或**SimpleLeakAwareByteBuf**包装真实的**ByteBuf**

代码实现

AdvancedLeakAwareByteBuf

构造函数中传入需要包装的ByteBuf和ResourceLeak对象:

```
final class AdvancedLeakAwareByteBuf extends WrappedByteBuf {  
    private final ResourceLeak leak;  
    AdvancedLeakAwareByteBuf(ByteBuf buf, ResourceLeak leak) {  
        super(buf);  
        this.leak = leak;  
    }  
    .....  
}
```

然后几乎所有的方法,都在**delegate**之前多加一句对**leak.record()**的调用:

```
public double getDouble(int index) {  
    leak.record();  
    return super.getDouble(index);  
}
```

然后所有的slice()/duplicate()/readSlice()/order()方法都在返回新的ByteBuf前再次包装成AdvancedLeakAwareByteBuf:

```
public ByteBuf slice() {  
    leak.record();  
    return new AdvancedLeakAwareByteBuf(super.slice(), leak);  
}
```

SimpleLeakAwareByteBuf

实现方式基本和AdvancedLeakAwareByteBuf相似,但是只有一个方法添加了对leak.record()的调用:

- order(ByteOrder endianness)

netty中的Zero-copy与传统意义的zero-copy不太一样,注意不要混淆:

- 传统的zero-copy是IO传输过程中,数据无需中内核态到用户态、用户态到内核态的数据拷贝,减少拷贝次数。
- Netty的zero-copy则是完全在用户态,或者说传输层的zero-copy机制,可以参考下图。

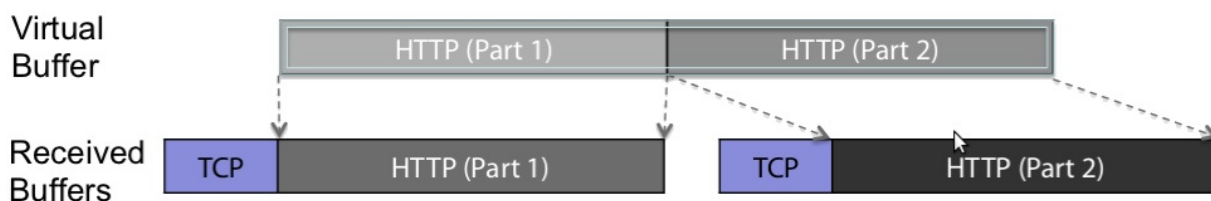
在协议传输过程中,通常需要拆包、合并包,常见的做法就是通过System.arraycopy来复制需要的数据,但这样需要付出内容复制的开销。

Netty通过ByteBuf.slice和Unpooled.wrappedBuffer等方法拆分、合并Buffer,做到无需拷贝数据。

slice

slice是拆包,在原有包的基础上获取其中部分内容,为此在ByteBuf中定义有slice()方法。

```
requestPart1 = buffer1.slice(OFFSET_PAYLOAD,
    buffer1.readableBytes() - OFFSET_PAYLOAD);
requestPart2 = buffer2.slice(OFFSET_PAYLOAD,
    buffer2.readableBytes() - OFFSET_PAYLOAD);
request = ChannelBuffers.wrappedBuffer(requestPart1, requestPart2);
```



wrap

netty提供了一系列的方法,用来将多个数据源包装成一个可以直接使用的ByteBuf对象,以避免内容复制。

这些方法处理的数据类型包括:

- byte[]

- ByteBuf: wraps the specified buffer's readable bytes
- ByteBuffer (java.nio)

```
public static ByteBuf wrappedBuffer(byte[] array) {}
public static ByteBuf wrappedBuffer(byte[] array, int offset, int
length) {}
public static ByteBuf wrappedBuffer(ByteBuffer buffer) {}
public static ByteBuf wrappedBuffer(ByteBuf buffer) {}

public static ByteBuf wrappedBuffer(byte[]... arrays) {}
public static ByteBuf wrappedBuffer(ByteBuf... buffers) {}
public static ByteBuf wrappedBuffer(ByteBuffer... buffers) {}

public static ByteBuf wrappedBuffer(int maxNumComponents, byte[]
... arrays) {}
public static ByteBuf wrappedBuffer(int maxNumComponents, ByteBu
f... buffers) {}
public static ByteBuf wrappedBuffer(int maxNumComponents, ByteBu
ffer... buffers) {}
```

可以包装一个或者多个数据。

继承关系

类SlicedByteBuf的继承关系,继承自AbstractDerivedByteBuf:

```
public class SlicedByteBuf extends AbstractDerivedByteBuf {  
    private ByteBuf buffer;  
    private int adjustment;  
    private int length;  
}
```

内部属性

SlicedByteBuf有三个最重要的内部属性:

1. **buffer**: 引用(或者说映射/指向)的底层ByteBuf对象,真实的数据时存放在这里
2. **adjustment**: 调整量,类似偏移量的概念,指当前SlicedByteBuf映射到底层ByteBuf对象时的偏移量
3. **length**: 当前SlicedByteBuf的长度

可以理解为SlicedByteBuf是对原有ByteBuf对象的部分数据的抽象,比如原来有一个ByteBuf有100个字节,我们将其中第20到49这30个字节的数据映射到一个新的SlicedByteBuf,这个SlicedByteBuf

1. **buffer**: 指向原来的这个100字节的ByteBuf
2. **adjustment**: 设置为20,表示从下标20开始的数据
3. **length**: 设置为30,表示从adjustment开始的30字节的数据范围

代码实现

构造函数

我们来看SlicedByteBuf的构造函数:


```
public SlicedByteBuf(ByteBuf buffer, int index, int length) {
    super(length);
    init(buffer, index, length);
}

final void init(ByteBuf buffer, int index, int length) {
    if (buffer instanceof SlicedByteBuf) {
        this.buffer = ((SlicedByteBuf) buffer).buffer;
        adjustment = ((SlicedByteBuf) buffer).adjustment + index
    ;
    } else if (buffer instanceof DuplicatedByteBuf) {
        this.buffer = buffer.unwrap();
        adjustment = index;
    } else {
        this.buffer = buffer;
        adjustment = index;
    }
    this.length = length;
    maxCapacity(length);
    setIndex(0, length);
    discardMarks();
}
```

可以看到buffer/adjustment/length的设置和前面说的一致.

注意SlicedByteBuf是支持对另外一个SlicedByteBuf做slice的.

方法

部分方法是直接delegate给buffer:

```
public boolean hasArray() {
    return buffer.hasArray();
}
```

部分方法是delegate给buffer,但是需要考虑adjustment因素:

```
protected byte _getBytes(int index) {  
    return buffer.getBytes(index + adjustment);  
}
```

在类Unpooled中提供给了多个名为wrappedBuffer()的帮助方法,用来创建包装给定数据的ByteBuf对象.

包装单个byte数组

代码实现:

```
public static ByteBuf wrappedBuffer(byte[] array) {  
    if (array.length == 0) {  
        return EMPTY_BUFFER;  
    }  
    return new UnpooledHeapByteBuf(ALLOC, array, array.length);  
}
```

结论: byte[]被包装成一个UnpooledHeapByteBuf对象.

包装多个byte数组

代码实现:

```

public static ByteBuf wrappedBuffer(int maxNumComponents, byte[]
... arrays) {
    switch (arrays.length) {
        case 0:
            //如果长度为0, 返回EMPTY_BUFFER
            break;
        case 1:
            if (arrays[0].length != 0) {
                //长度为1, 调用wrappedBuffer(byte[])
                return wrappedBuffer(arrays[0]);
            }
            break;
        default:
            // Get the list of the component, while guessing the byte
            // order.
            final List<ByteBuf> components = new ArrayList<ByteBuf>(
            arrays.length);
            for (byte[] a: arrays) {                // 遍历数组, 如果为null或空
就跳过
                if (a == null) {
                    break;
                }
                if (a.length > 0) {
                    // 有效的数据就通过调用wrappedBuffer(byte[])包装为一个
ByteBuf
                    components.add(wrappedBuffer(a));
                }
            }

            if (!components.isEmpty()) {
                //将多个ByteBuf包装为1个CompositeByteBuf
                return new CompositeByteBuf(ALLOC, false, maxNumComp
onents, components);
            }

            return EMPTY_BUFFER;
    }
}

```

结论: 多个byte[]被包装成多个UnpooledHeapByteBuf对象,最后再包装进一个CompositeByteBuf.

包装单个ByteBuf

代码实现:

```
public static ByteBuf wrappedBuffer(ByteBuf buffer) {  
    if (buffer.isReadable()) {  
        // 有可读数据, 调用buffer.slice()将可读数据包装为一个ByteBuf  
        return buffer.slice();  
    } else {  
        // 如果没有可读数据, 返回EMPTY_BUFFER  
        return EMPTY_BUFFER;  
    }  
}
```

结论: 多个ByteBuf对象(的可读内容)被包装进一个CompositeByteBuf.

包装多个ByteBuf

代码实现:

```
public static ByteBuf wrappedBuffer(int maxNumComponents, ByteBuf... buffers) {
    switch (buffers.length) {
        case 0:
            //如果长度为0, 返回EMPTY_BUFFER
            break;
        case 1:
            if (buffers[0].isReadable()) {
                //长度为1并且有可读数据, 调用wrappedBuffer(ByteBuf)
                // TBD: 不清楚为什么这里需要调用 order(BIG_ENDIAN)?
                return wrappedBuffer(buffers[0].order(BIG_ENDIAN));
            }
            break;
        default:
            // 注意这里的判断: 只要有一个ByteBuf有可读数据, 就包装为CompositeByteBuf
            for (ByteBuf b: buffers) {
                if (b.isReadable()) {
                    //将多个ByteBuf包装为1个CompositeByteBuf
                    return new CompositeByteBuf(ALLOC, false, maxNumComponents, buffers);
                }
            }
            return EMPTY_BUFFER;
    }
}
```

结论: 多个ByteBuf对象(的可读内容)被包装进一个CompositeByteBuf.

在netty的zero-copy的实现中,类CompositeByteBuf非常的重要.

CompositeByteBuf的javadoc这样描述:

```
A virtual buffer which shows multiple buffers as a single merged buffer.
```

虚拟的buffer,将多个buffer展现为一个简单合并的buffer.

使用上,推荐使用以下两类帮助方法来创建CompositeByteBuf对象,尽量不要直接调用构造函数:

- ByteBufAllocator.compositeBuffer()
- Unpooled.wrappedBuffer(ByteBuf...)

继承关系

```
public class CompositeByteBuf extends AbstractReferenceCountedByteBuf implements Iterable<ByteBuf> {}
```

属性

以下几个属性是CompositeByteBuf特有的属性:

```
private final ByteBufAllocator alloc;  
private final boolean direct;  
private final List<Component> components = new ArrayList<Component>();  
private final int maxNumComponents;
```

Component是私有内嵌类,用于保存ByteBuf的引用,长度等信息.

```
private static final class Component {
    final ByteBuf buf;
    final int length;
    int offset;
    int endOffset;
    .....
}
```

另外为了支持ReferenceCounted和resourceleakdetect,还有两个特殊属性:

```
private final ResourceLeak leak;
private boolean freed;
```

构造函数

第一个构造函数比较简单,没有传入ByteBuf,需要在后面通过调用addComponent()方法来增加内容:

```
public CompositeByteBuf(ByteBufAllocator alloc, boolean direct,
int maxNumComponents) {
    super(Integer.MAX_VALUE);    //设置maxCapacity为Integer.MAX_V
    ALUE
    if (alloc == null) {
        throw new NullPointerException("alloc");
    }
    this.alloc = alloc;
    this.direct = direct;
    this.maxNumComponents = maxNumComponents;
    leak = leakDetector.open(this);
}
```

第二个构造函数传入了ByteBuf:


```
public CompositeByteBuf(ByteBufAllocator alloc, boolean direct,
int maxNumComponents, ByteBuf... buffers) {
    super(Integer.MAX_VALUE);
    if (alloc == null) {
        throw new NullPointerException("alloc");
    }
    if (maxNumComponents < 2) {
        // 这里做法比较狠,如果传入的maxNumComponents<2就直接抛异常退出
        throw new IllegalArgumentException(
            "maxNumComponents: " + maxNumComponents + " (exp
ected: >= 2)");
    }

    this.alloc = alloc;
    this.direct = direct;
    this.maxNumComponents = maxNumComponents;

    addComponents0(0, buffers);           //添加传入的buffers,index自
然是从0开始
    consolidateIfNeeded();
    setIndex(0, capacity());
    leak = leakDetector.open(this);
}
```

addComponent0()方法:

```
private int addComponent0(int cIndex, ByteBuf buffer) {
    checkComponentIndex(cIndex);

    if (buffer == null) {
        throw new NullPointerException("buffer");
    }

    int readableBytes = buffer.readableBytes();

    // No need to consolidate - just add a component to the list.

    Component c = new Component(buffer.order(ByteOrder.BIG_ENDIAN)
        .slice());
    if (cIndex == components.size()) {
        // 加在最后
        components.add(c);
        if (cIndex == 0) {
            c.endOffset = readableBytes;
        } else {
            Component prev = components.get(cIndex - 1);
            c.offset = prev.endOffset;
            c.endOffset = c.offset + readableBytes;
        }
    } else {
        // 加在中间指定index上
        components.add(cIndex, c);
        if (readableBytes != 0) {
            updateComponentOffsets(cIndex);
        }
    }
    return cIndex;
}
```

再看consolidateIfNeeded()方法:

```
private void consolidateIfNeeded() {
    // Consolidate if the number of components will exceed the allowed maximum by the current
    // operation.
    final int numComponents = components.size();
    if (numComponents > maxNumComponents) {    //这个if判断至关重要
        !!
        final int capacity = components.get(numComponents - 1).endOffset;

        ByteBuffer consolidated = allocBuffer(capacity);

        // We're not using foreach to avoid creating an iterator.

        for (int i = 0; i < numComponents; i++) {
            Component c = components.get(i);
            ByteBuffer b = c.buf;
            consolidated.writeBytes(b);
            c.freeIfNecessary();
        }
        Component c = new Component(consolidated);
        c.endOffset = c.length;
        components.clear();
        components.add(c);
    }
}
```

可以看到: 当实际**components**的数量大于**maxNumComponents**时, 所有的**components**就会被合并为一个ByteBuffer, "consolidated.writeBytes(b)" 这段代码就是在将原来ByteBuffer的数据写入到新的consolidated这个合并之后的ByteBuffer. 这意味着数据被复制, zeor-copy也就不存在了.

反回来看numComponents的设值, 这是一个final, 通过构造函数一次性赋值.

```
private final int maxNumComponents;
public CompositeByteBuf(ByteBufAllocator alloc, boolean direct,
int maxNumComponents) {
    .....
    this.maxNumComponents = maxNumComponents;
    .....
}
```

而在前面Unpooled.wrappedBuffer()方法中, 如果没有明确设置maxNumComponents, 则默认为16:

```
public static ByteBuf wrappedBuffer(byte[]... arrays) {
    return wrappedBuffer(16, arrays);
}
```

这里是否存在风险: 如果默认maxNumComponents为16, 那么当component添加时, 一旦超过16, 就将触发consolidate的操作, 违背zero-copy.

其他方法

貌似没有什么特殊

codec

Decoder

Decoder Result

DecoderResult

解码的结果只有三种：unfinished，success，failure。

为 UNFINISHED 和 SUCCESS 定义了两个 Signal 常量和 DecoderResult 常量：

```
protected static final Signal SIGNAL_UNFINISHED = Signal.valueOf(
    DecoderResult.class, "UNFINISHED");
protected static final Signal SIGNAL_SUCCESS = Signal.valueOf(De
    coderResult.class, "SUCCESS");

public static final DecoderResult UNFINISHED = new DecoderResult
    (SIGNAL_UNFINISHED);
public static final DecoderResult SUCCESS = new DecoderResult(SI
    GNAL_SUCCESS);
```

如果失败则记录失败原因，这个类只有一个类成员变量 cause，所有的方法都是围绕它，如构造方法：

```
private final Throwable cause;
protected DecoderResult(Throwable cause) {
    if (cause == null) {
        throw new NullPointerException("cause");
    }
    this.cause = cause;
}
```

定义了几个判断方法来获取解码的结果：

```
public boolean isFinished() {
    return cause != SIGNAL_UNFINISHED;
}

public boolean isSuccess() {
    return cause == SIGNAL_SUCCESS;
}

public boolean isFailure() {
    return cause != SIGNAL_SUCCESS && cause != SIGNAL_UNFINISHED
;
}
```

DecoderResultProvider

为被解码的消息的 `DecoderResult` 属性提供访问方法。

```
public interface DecoderResultProvider {
    /**
     * 返回解码这个对象的结果
     */
    DecoderResult decoderResult();

    /**
     * 更新解码这个对象的结果。
     * 这个方法应该由解码器调用。
     * 不要调用这个方法，除非你知道自己在做什么。
     */
    void setDecoderResult(DecoderResult result);
}
```


http

http objects

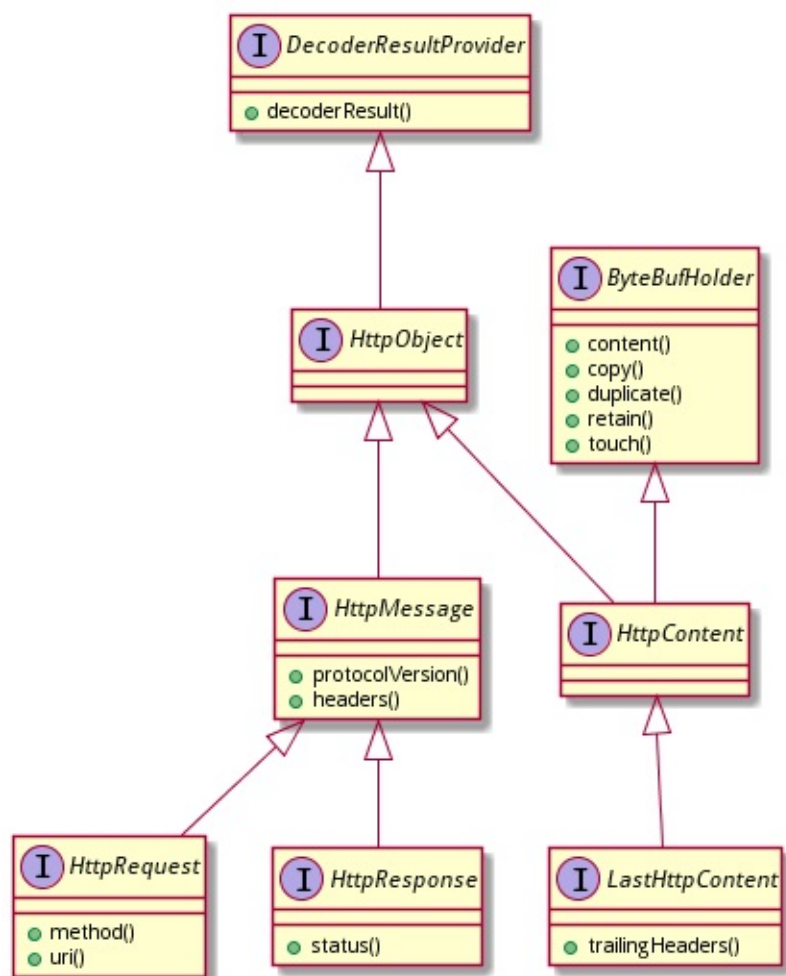
为了表示HTTP中各种消息体，netty设计了一套完整的类型定义，包括接口和实现类。

由于类型比较多，部分实现类继承多个类和接口，关系有些复杂。我们按照接口先梳理一遍各个消息的基本情况，再看具体的类实现方式。

HTTP接口定义

基础接口

我们先来看看HTTP的基本的接口定义。下图是netty中HTTP相关的6个接口的继承结构，还有他们继承的接口：



先简单过一下各个接口的情况：

1. `HttpObject`:

```
public interface HttpObject extends DecoderResultProvider {  
}
```

这个接口是一个标识接口，本身没有定义任何东西。通过继承 `DecoderResultProvider` 提供了获取 `DecoderResult` 的方法。

2. `HttpMessage`: 定义HTTP消息，为 `HttpRequest` 和 `HttpResponse` 提供通用属性：

```
public interface HttpMessage extends HttpObject {  
    HttpVersion protocolVersion();  
    HttpHeaders headers();  
}
```

所谓通用属性其实就只有两个：协议版本和header。

3. `HttpRequest`: 对应HTTP request。

访问查询参数和cookie。和servlet api不同的是，query string通过 `QueryStringEncoder`和 `QueryStringDecoder`来构建和拆解。

```
public interface HttpRequest extends HttpMessage {  
    HttpMethod method();  
    String uri();  
}
```

和 `HttpMessage` 相比，`HttpRequest` 增加了 `http method` 和 `uri` 两个参数。

4. `HttpResponse`：对应HTTP response.

```
public interface HttpResponse extends HttpMessage {  
    HttpResponseStatus status();  
}
```

和`HttpMessage`相比，`HttpResponse`增加了`status`参数。

5. `HttpContent`：是一个HTTP chunk，用于HTTP chunked 传输编码.

当内容很大或者内容编码是chunked时，`HttpObjectDecoder` 会在生成 `HttpMessage` 之后再生成 `HttpContent` 。如果不想在handler中接受到 `HttpContent` ，可以在 `ChannelPipeline` 中的 `HttpObjectDecoder` 之后放一个 `HttpObjectAggregator` 。

`HttpContent` 继承自 `HttpObject` 和 `ByteBufHolder`：

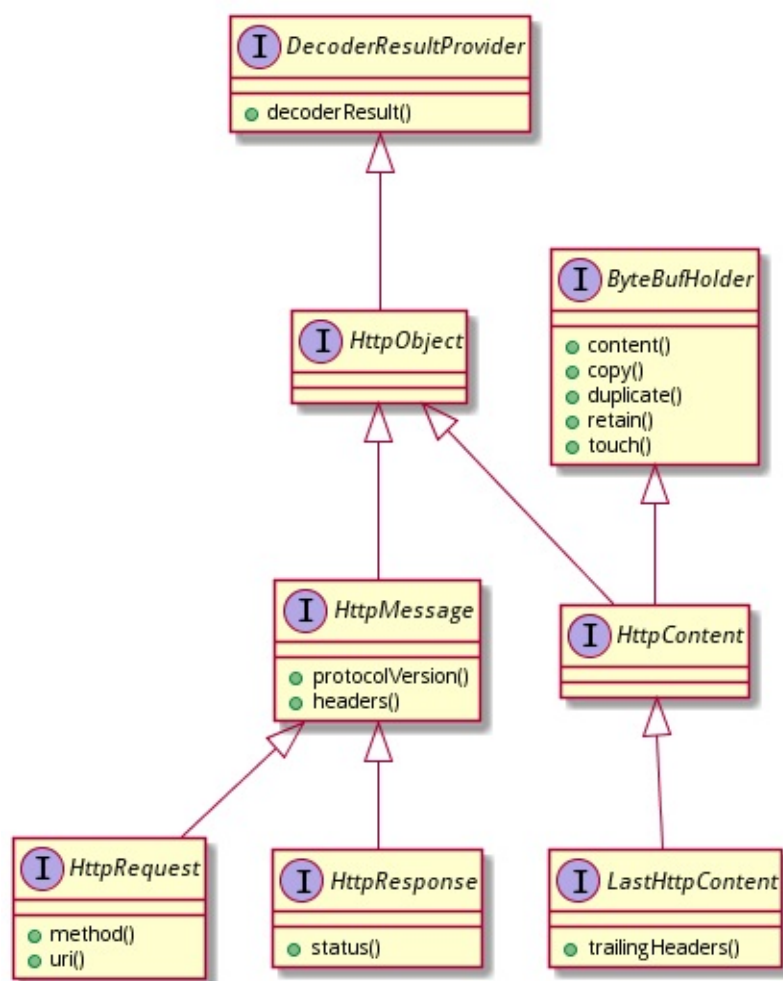
```
public interface HttpContent extends HttpObject, ByteBufHolder {}
```

6. `LastHttpContent`

`LastHttpContent` 是最后一个 `HttpContent` ，带有trailing headers。

```
public interface LastHttpContent extends HttpContent {  
    HttpHeaders trailingHeaders();  
}
```

再对照继承结构图，总结一下：



1. HttpObject接口是整个HTTP结构的继承树顶端。
2. HttpObject下分为HttpMessage和HttpContent两大体系
3. HttpMessage进一步拆分为HttpRequest和HttpResponse
4. HttpContent是chunk专用的内容表示方式
5. LastHttpContent是作为trunk结束的一个特别HttpContent

全数据接口

为了方便使用HttpMessage和HttpContent，以表示一个完整的HTTP消息体，netty定义了三个Full Http接口：

1. FullHttpMessage

FullHttpMessage用来合并HttpMessage和LastHttpContent到一个消息中，因此FullHttpMessage可以用来表示一个完整的http信息：

```
public interface FullHttpMessage extends HttpMessage, LastH
ttpContent {
    // 用替代的内容创建这个FullHttpMessage的副本
    FullHttpMessage copy(ByteBuf newContent);
}
```

2. FullHttpRequest

合并HttpRequest和FullHttpMessage（实际指LastHttpContent）表示一个完整的http request：

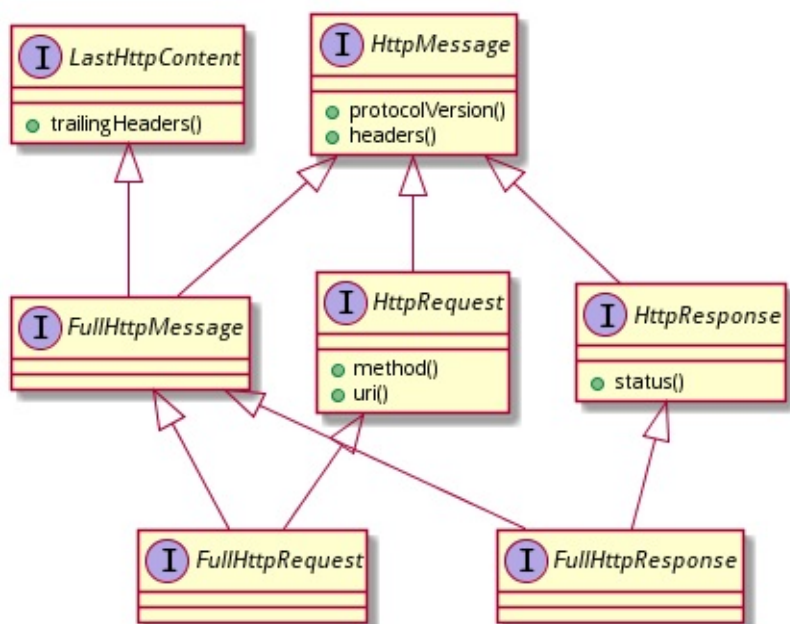
```
public interface FullHttpRequest extends HttpRequest, FullH
ttpMessage {}
```

3. FullHttpResponse

合并HttpResponse和FullHttpMessage（实际指LastHttpContent）表示一个完整的http response：

```
public interface FullHttpResponse extends HttpResponse, Ful
lHttpMessage {}
```

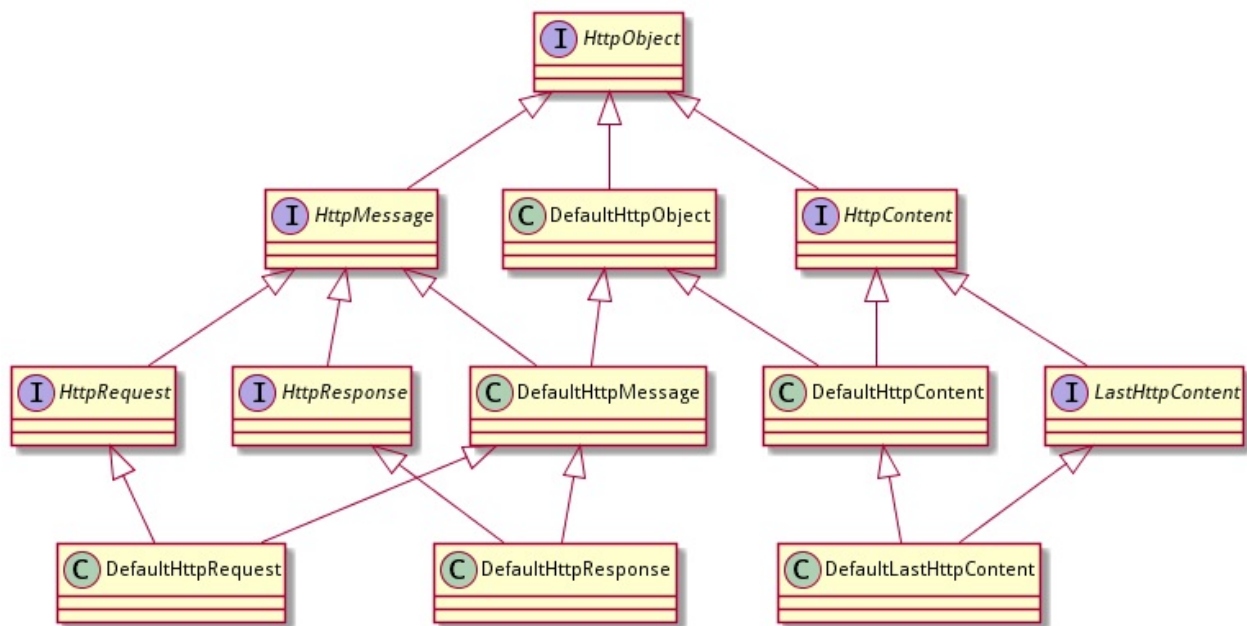
对照继承结构图，总结：



1. 这三个接口都是简单的接口定义，标明继承关系而已，没有实质内容
2. 除了FullHttpRequest接口多了一个copy(ByteBuf newContent)方法

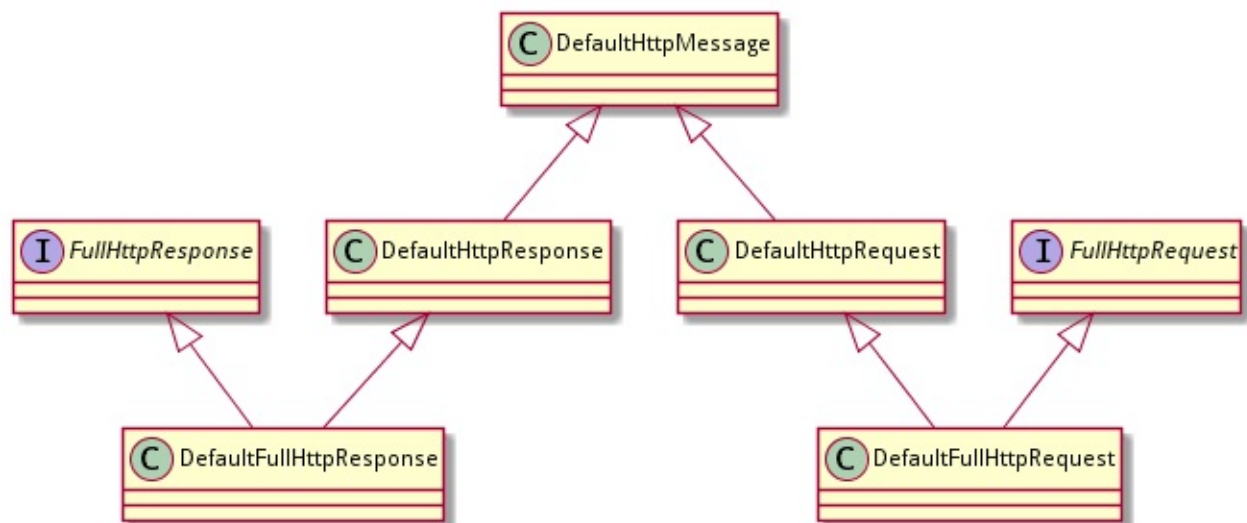
实现类

基础实现类



基础实现类的结构和基础接口呈现一一对应的关系：每个基础接口都有一个default实现类，实现类在实现该接口的同时，维持一致的继承关系。

全数据实现类



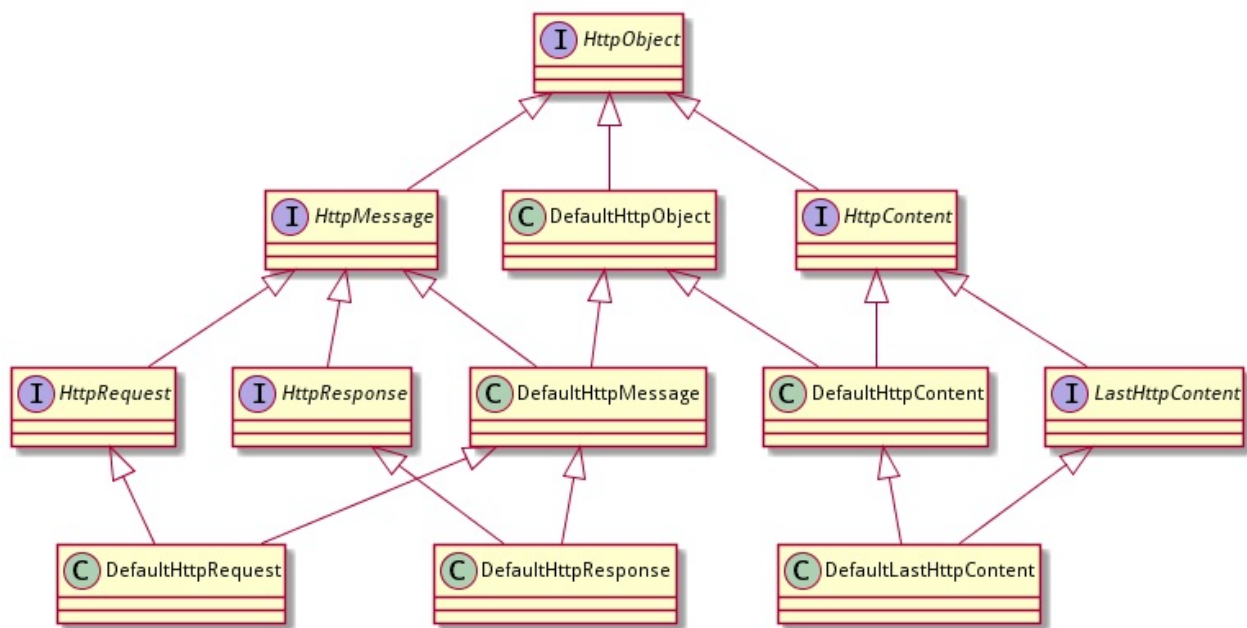
两个DefaultFull的实现类，从对应的基础实现类下继承，然后实现了FullHttp接口。

具体的类实现细节见下一章。

http类实现

注：没啥特别内容，可以跳过。

基础实现类



类DefaultHttpObject

提供了类成员decoderResult，仅此而已。

```
public class DefaultHttpObject implements HttpObject {
    private DecoderResult decoderResult = DecoderResult.SUCCESS;
}
```

类DefaultHttpMessage

提供了类成员version和headers，。

```
public abstract class DefaultHttpMessage extends DefaultHttpObject
implements HttpMessage {
    private HttpVersion version;
    private final HttpHeaders headers;
}
```

有两个细节，后面再看：

1. headers是final，但是version不是，而是有一个
setProtocolVersion(HttpVersion version)方法可以用来设置
2. DefaultHttpMessage是抽象类，但是它继承的DefaultHttpObject却不是抽象类

```
protected DefaultHttpMessage(final HttpVersion version, boolean
validateHeaders, boolean singleFieldHeaders) {
    this.version = checkNotNull(version, "version");
    headers = singleFieldHeaders ?
        new CombinedHttpHeaders(validateHeaders):
        new DefaultHttpHeaders(validateHeaders);
}
```

类DefaultHttpContent

提供成员变量content，这是一个ByteBuf。注意是final，构造函数初始化之后不再容许改动：

```
public class DefaultHttpContent extends DefaultHttpObject implements
HttpContent {
    private final ByteBuf content;
}
```

HttpContent继承自ByteBufHolder接口，就是对应这个ByteBuf，所有ByteBufHolder接口的方法实现都是操作这个ByteBuf，如：

```
@Override
public HttpContent retain() {
    content.retain();
    return this;
}
```

类DefaultLastHttpContent

提供成员变量trailingHeaders，用于保存trunk编码中的trailing header：

```
public class DefaultLastHttpContent extends DefaultHttpContent implements LastHttpContent {
    private final HttpHeaders trailingHeaders;
}
```

类DefaultHttpRequest

提供成员变量method和uri：

```
public class DefaultHttpRequest extends DefaultHttpMessage implements HttpRequest {
    private HttpMethod method;
    private String uri;
}
```

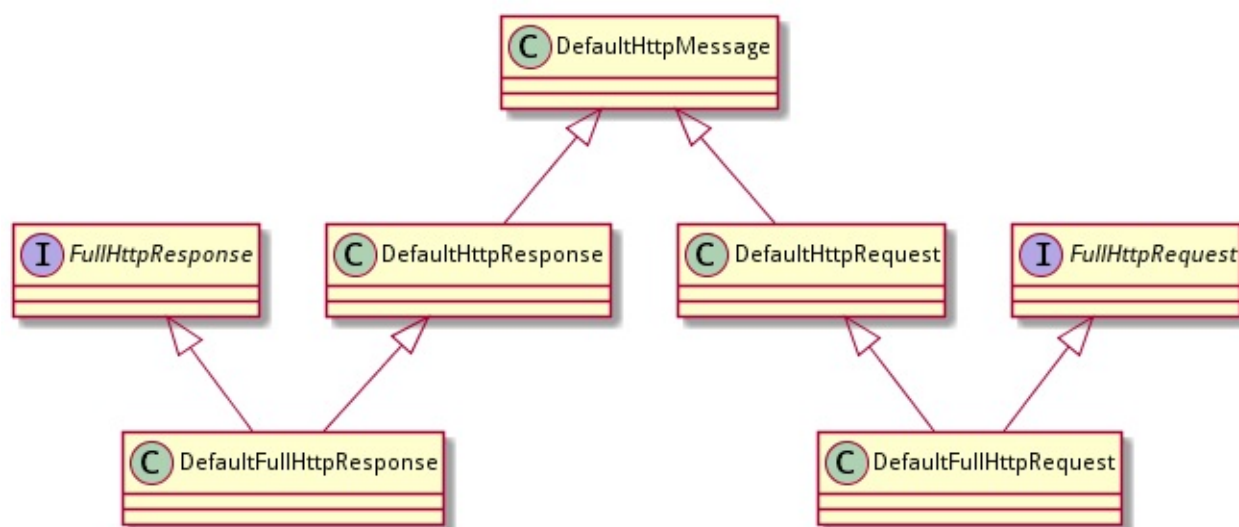
类DefaultHttpResponse

提供成员变量status：

```
public class DefaultHttpResponse extends DefaultHttpMessage implements HttpResponse {
    private HttpResponseStatus status;
}
```

总结：基础实现类基本没有特别的地方，按照继承结构简单实现而已。

全数据实现类



类DefaultFullHttpResponse

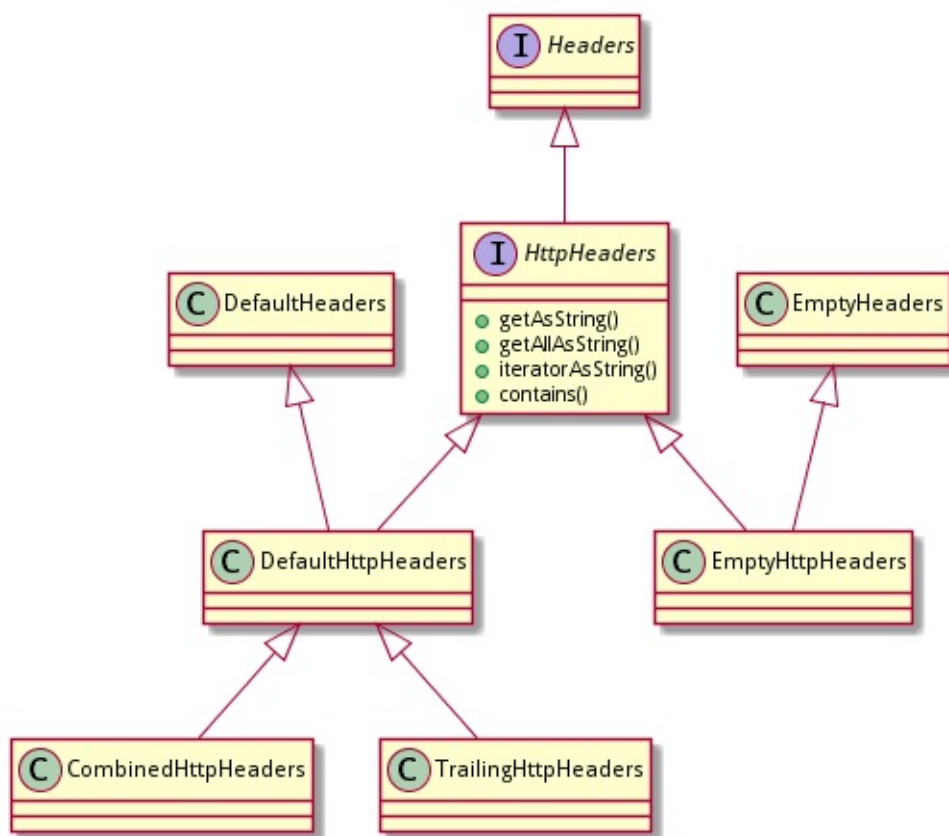
类DefaultFullHttpResponse继承自DefaultHttpResponse，为了实现FullHttpResponse接口，携带内容和trailingHeaders，增加了两个成员变量：

```
public class DefaultFullHttpResponse extends DefaultHttpResponse
    implements FullHttpResponse {
    private final ByteBuf content;
    private final HttpHeaders trailingHeaders;
}
```

类DefaultFullHttpRequest

类似类DefaultFullHttpResponse，没啥特别内容。

header



接口 HttpHeaders

提供访问HttpMessage的常用工具方法，继承自Headers，加了几个方法。

```

public interface HttpHeaders extends Headers<CharSequence, CharSequence, HttpHeaders> {
    String getAsString(CharSequence name);
    List<String> getAllAsString(CharSequence name);
    Iterator<Entry<String, String>> iteratorAsString();
    boolean contains(CharSequence name, CharSequence value, boolean ignoreCase);
}
  
```

类 DefaultHttpHeaders


```
public interface HttpHeaders extends Headers<CharSequence, CharSequence, HttpHeaders> {  
    String getAsString(CharSequence name);  
    List<String> getAllAsString(CharSequence name);  
    Iterator<Entry<String, String>> iteratorAsString();  
    boolean contains(CharSequence name, CharSequence value, boolean ignoreCase);  
}
```

TODO：后面再细看validate的内容。

类EmptyHttpHeaders

没啥特别内容。

```
public class EmptyHttpHeaders extends EmptyHeaders<CharSequence, CharSequence, HttpHeaders> implements HttpHeaders {  
}
```



类CombinedHttpHeaders

CombinedHttpHeaders将多个相同名字的header合并到一个header，值为逗号分隔的多个值

```
public class EmptyHttpHeaders extends EmptyHeaders<CharSequence, CharSequence, HttpHeaders> implements HttpHeaders {  
}
```



类TrailingHttpHeaders

TrailingHttpHeaders是DefaultLastHttpContent中私有内嵌类，用于表示trunk编码中的trailing header:

```
public class DefaultLastHttpContent{  
    private static final class TrailingHttpHeaders extends DefaultHttpHeaders {  
    }  
}
```

http codec

类HttpServerCodec

类HttpServerCodec是HttpRequestDecoder和HttpResponseEncoder的组合，可以简化HTTP的服务器端实现。

类定义

```
public final class HttpServerCodec extends CombinedChannelDuplex
Handler<HttpRequestDecoder, HttpResponseEncoder>
    implements HttpServerUpgradeHandler.SourceCodec {
}
```

Tags