文章目录

op（二）之

对 EventLoopGroup 和 EventLoop 做了定义，我们再来回

Loop 负责处理注册到其上的
O 操作。
组，它可以获取到一个或者多个
ventLoop 对象的方法。

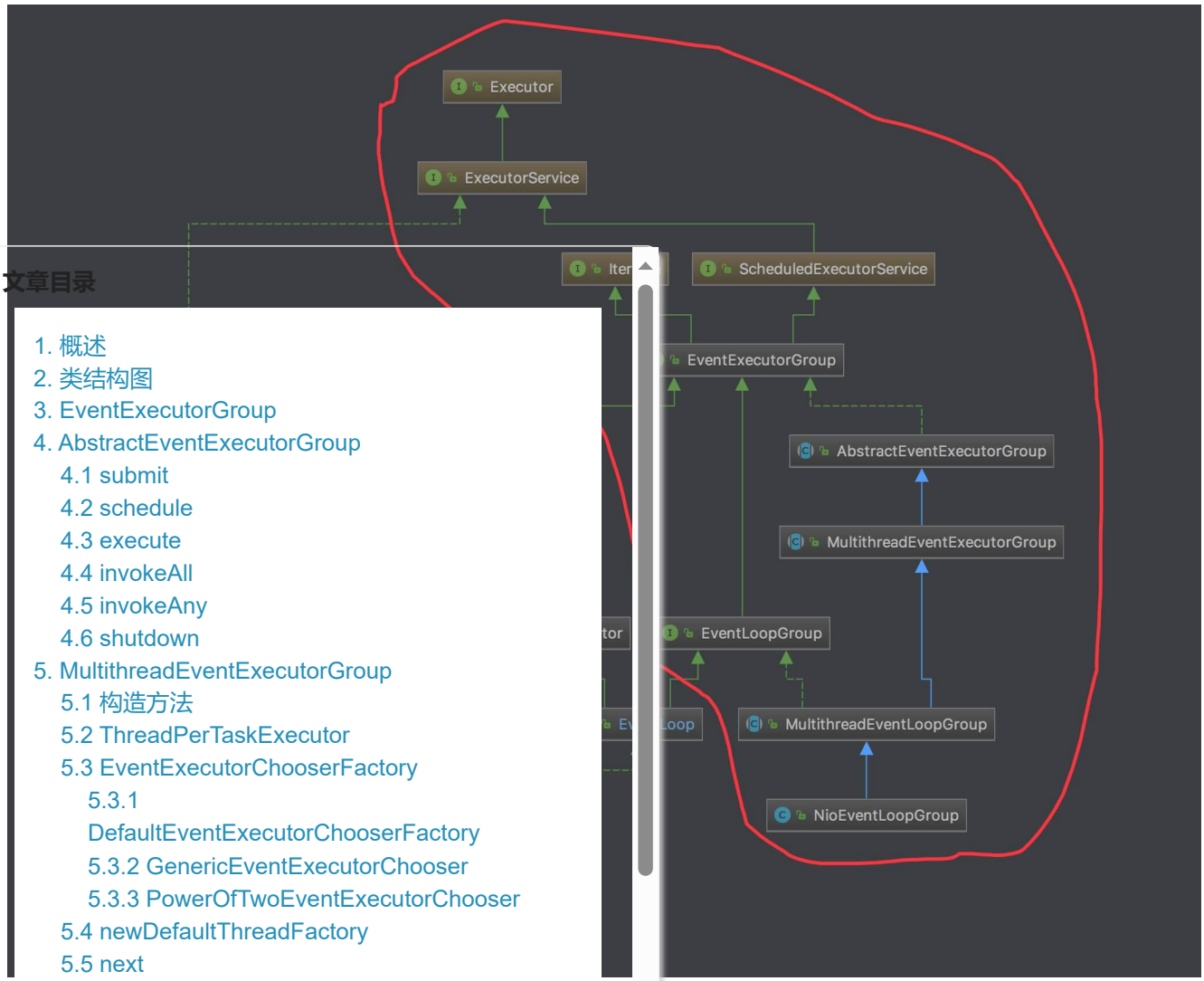
就是：

- `new NioEventLoopGroup()` ，创建一个 EventLoopGroup 对象。
- `EventLoopGroup#register(Channel channel)` ，将 Channel 注册到 EventLoopGroup 上。

那么，本文我们分享 EventLoopGroup 的具体代码实现，来一探究竟。

2. 类结构图

EventLoopGroup 的整体类结构如下图：



文章目录

为 E ntLoop 相关的类关系。

tLc ，所以笔者省略了其它相关的类，例如

下面，我们逐步看看每个接口相关的实现代码。

3. EventExecutorGroup

`io.netty.util.concurrent.EventExecutorGroup` ，实现 Iterable、ScheduledExecutorService 接口，EventExecutor (事件执行器)的分组接口。代码如下：

```
// ========== 自定义接口 ==========

boolean isShuttingDown();

// 优雅关闭
Future<?> shutdownGracefully();
Future<?> shutdownGracefully(long quietPeriod, long timeout, TimeUnit unit);

Future<?> terminationFuture();

// 选择一个 EventExecutor 对象
EventExecutor next();
```

```
// ========== 实现自 Iterable 接口 ==========

@Override
Iterator<EventExecutor> iterator();

// ========== 实现自 ExecutorService 接口 ==========

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

文章目录

- 每个接口的方法的意思比较好理解，笔者就不一一赘述了。
- 比较特殊的是，接口方法返回类型为 Future 不是 Java 原生的 `java.util.concurrent.Future` ，而是 Netty 自己实现的 Future 接口。详细解析，见后续文章。
- EventExecutorGroup 自身不执行任务，而是将任务 `#submit(...)` 或 `#schedule(...)` 给自己管理的 EventExecutor 的分组。至于提交给哪一个 EventExecutor ，一般是通过 `#next()` 方法，选择一个 EventExecutor 。

4. AbstractEventExecutorGroup

`io.netty.util.concurrent.AbstractEventExecutorGroup` ，实现 EventExecutorGroup 接口，EventExecutor (事件执行器)的分组抽象类。

4.1 submit

`#submit(...)` 方法，提交**一个**普通任务到 EventExecutor 中。代码如下：

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

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

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

文章目录

r 中 代码如下：

```
d, ng delay, TimeUnit unit) {

 c able, long delay, TimeUnit unit) {

ab command, long initialDelay, long period, TimeUn
ti elay, period, unit);

public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, Time
    return next().scheduleWithFixedDelay(command, initialDelay, delay, unit);
}
```

- 提交的 EventExecutor ， 通过 `#next()` 方法选择。

4.3 execute

`#execute(...)` 方法，在 EventExecutor 中执行**一个**普通任务。代码如下：

```
@Override
public void execute(Runnable command) {
    next().execute(command);
}
```

- 执行的 EventExecutor ， 通过 `#next()` 方法选择。
- 看起来 `#execute(...)` 和 `#submit(...)` 方法有几分相似，具体的差异，由 EventExecutor 的实现决定。

4.4 invokeAll

`#invokeAll(...)` 方法，在 EventExecutor 中执行**多个**普通任务。代码如下：

```
@Override
public <T> List<java.util.concurrent.Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) thr
    return next().invokeAll(tasks);
```

```
ll(Collection<? extends Callable<T>> tasks, lon
```

多 任务使用同一个 EventExecutor 。

任务 有**一个**执行完成即可。代码如下：

```
le  > tasks) throws InterruptedException, Execution
```

```
le  > tasks, long timeout, TimeUnit unit) throws In
```

- 执行的 EventExecutor ，通过 `#next()` 方法选择。并且，多个任务使用同一个 EventExecutor 。

4.6 shutdown

`#shutdown(...)` 方法，关闭 EventExecutorGroup 。代码如下：

```
@Override
public Future<?> shutdownGracefully() {
    return shutdownGracefully(DEFAULT_SHUTDOWN_QUIET_PERIOD /* 2 */, DEFAULT_SHUTDOWN_TIMEOUT /* 15 */
}

@Override
@Deprecated
public List<Runnable> shutdownNow() {
```

文章目录

g t eout, TimeUnit unit) 和 #shutdown() 方法，

OU)

Grc ，继承 AbstractEventExecutorGroup 抽象类，**基于多

```
/**
 * 不可变( 只读 )的 EventExecutor 数组
 *
 * @see #MultithreadEventExecutorGroup(int, Executor, EventExecutorChooserFactory, Object...)
 */
private final Set<EventExecutor> readonlyChildren;
/**
 * 已终止的 EventExecutor 数量
 */
private final AtomicInteger terminatedChildren = new AtomicInteger();
/**
 * 用于终止 EventExecutor 的异步 Future
 */
private final Promise<?> terminationFuture = new DefaultPromise(GlobalEventExecutor.INSTANCE);
/**
 * EventExecutor 选择器
 */
private final EventExecutorChooserFactory.EventExecutorChooser chooser;

protected MultithreadEventExecutorGroup(int nThreads, ThreadFactory threadFactory, Object... args) {
```

```
    this(nThreads, threadFactory == null ? null : new ThreadPerTaskExecutor(threadFactory), args);
}

protected MultithreadEventExecutorGroup(int nThreads, Executor executor, Object... args) {
    this(nThreads, executor, DefaultEventExecutorChooserFactory.INSTANCE, args);
}
```



```
protected MultithreadEventExecutorGroup(int nTh ds, Executor executor, EventExecutorChooserFact
    if (nThreads <= 0) {
                                                  ri format("nThreads: %d (expected: > 0)", nThreads
                                                  ne faultThreadFactory());
                                                  功
                                                  ar ;
                                                  pti 异常
                                                  g exception type
                                                  'fa d to create a child event loop", e);
                                                  Exe tor
                // 关闭所有已创建的 EventExecutor
                for (int j = 0; j < i; j ++) {
                    children[j].shutdownGracefully();
                }
                // 确保所有已创建的 EventExecutor 已关闭
                for (int j = 0; j < i; j ++) {
                    EventExecutor e = children[j];
                    try {
                        while (!e.isTerminated()) {
                            e.awaitTermination(Integer.MAX_VALUE, TimeUnit.SECONDS);
                        }
                    } catch (InterruptedException interrupted) {
                        // Let the caller handle the interruption.
                        Thread.currentThread().interrupt();
                        break;
                    }
                }
            }
        }
    }
```

```
48:
49:     // 创建 EventExecutor 选择器
50:     chooser = chooserFactory.newChooser(children);
51:
52:     // 创建监听器，用于 EventExecutor 终止时的监听
53:     final FutureListener<Object> terminationListener = new FutureListener<Object>() {
54:
55:         @Override
56:         public void operationComplete(Future<Ob      t> future) throws Exception {
                                                  and(      () == children.length) { // 全部关闭
                                                  nu        ; // 设置结果，并通知监听器们。
                                                  mi        ionListener);
                                                  ted       hSet<EventExecutor>(children.length);
                                                  ;
                                                  les       (childrenSet);
                                                  adP       askExecutor」 。
                                                  ect       . args) 方法，创建 EventExecutor 对象，然后
```


- 第 21 至 24 行：创建失败，抛出 IllegalStateException 异常。
- 第 25 至 45 行：创建失败，关闭所有已创建的 EventExecutor 。
- 第 50 行：调用 `EventExecutorChooserFactory#newChooser(EventExecutor[] executors)` 方法，创建 EventExecutor 选择器。详细解析，见「5.3 EventExecutorChooserFactory」 。
- 第 52 至 62 行：创建监听器，用于 EventExecutor 终止时的监听。
  - 第 55 至 60 行：回调的具体逻辑是，当所有 EventExecutor 都终止完成时，通过调用 `Future#setSuccess(V result)` 方法，通知监听器们。至于为什么设置的值是 `null` ，因为监听器们不关注具体的结果。
  - 第 63 至 66 行：设置监听器到每个 EventExecutor 上。
- 第 68 至 71 行：创建不可变(只读)的 EventExecutor 数组。

5.2 ThreadPerTaskExecutor

`io.netty.util.concurrent.ThreadPerTaskExecutor` ，实现 Executor 接口，每个任务一个线程的执行器实现类。代码如下：

```
public final class ThreadPerTaskExecutor implements Executor {

    /**
     * 线程工厂对象
     */
    private final ThreadFactory threadFactory;
```

```
    public ThreadPerTaskExecutor(ThreadFactory threadFactory) {
        if (threadFactory == null) {
            throw new NullPointerException("threadFactory");
        }
        this.threadFactory = threadFactory;
    }
```

文章目录

的 T eadFactory 类，为

，关 DefaultThreadFactory 比较简单，胖友可以自己看

tor newThread(Runnable) 方法，创建一个 Thread ，

ory EventExecutorChooser 工厂接口。代码如下：

```
     * 创建一个 EventExecutorChooser 对象
     *
     * Returns a new {@link EventExecutorChooser}.
     */
    EventExecutorChooser newChooser(EventExecutor[] executors);

    /**
     *  EventExecutor 选择器接口
     *
     * Chooses the next {@link EventExecutor} to use.
     */
    @UnstableApi
    interface EventExecutorChooser {

        /**
         * 选择下一个 EventExecutor 对象
         *
         * Returns the new {@link EventExecutor} to use.
         */
        EventExecutor next();
```

```
    }

}
```

- `#newChooser(EventExecutor[] executors)` 方法，创建一个 EventExecutorChooser 对象。
- EventExecutorChooser 接口，EventExecutor 选择器接口。
- `ext()` 方法，选择下一个 EventExecutor 对象。

文章目录

`ser ctory` ，实现 EventExecutorChooserFactory 接口，默

```
tor INSTANCE = new DefaultEventExecutorChooserFacto

tor executors) {
 2 的幂次方
ex tors);

cu s);

}
```

- `INSTANCE` **静态**属性，单例。
- `#newChooser(EventExecutor[] executors)` 方法，调用 `#isPowerOfTwo(int val)` 方法，判断 EventExecutor 数组的大小是否为 2 的幂次方。
  - 若是，创建 PowerOfTwoEventExecutorChooser 对象。详细解析，见「5.3.3 PowerOfTwoEventExecutorChooser」。
  - 若否，创建 GenericEventExecutorChooser 对象。详细解析，见「5.3.2 GenericEventExecutorChooser」。
- `#isPowerOfTwo(int val)` 方法，为什么 `(val & -val) == val` 可以判断数字是否为 2 的幂次方呢?
  - 我们以 8 来举个例子。
    - 8 的二进制为 `1000` 。
    - -8 的二进制使用补码表示。所以，先求反生成反码为 `0111` ，然后加一生成补码为 `1000` 。
    - 8 和 -8 并操作后，还是 8 。
    - 实际上，以 2 为幂次方的数字，都是最高位为 1 ，剩余位为 0 ，所以对应的负数，求完补码还是自己。
  - 胖友也可以自己试试非 2 的幂次方数字的效果。

5.3.2 GenericEventExecutorChooser

GenericEventExecutorChooser 实现 EventExecutorChooser 接口，通用的 EventExecutor 选择器实现类。代码如下：

> GenericEventExecutorChooser 内嵌在 DefaultEventExecutorChooserFactory 类中。

```
private static final class GenericEventExecutorChoos implements EventExecutorChooser {
```

文章目录


```
ite  ();
ec  rs) {
ien  % executors.length)];
```

组的 小来取余。

PowerOfTwoEventExecutorChooser 实现 EventExecutorChooser 接口，基于 EventExecutor 数组的大小为 2 的幂次方的 EventExecutor 选择器实现类。这是一个优化的实现，代码如下：

> PowerOfTwoEventExecutorChooser 内嵌在 DefaultEventExecutorChooserFactory 类中。

```
private static final class PowerOfTwoEventExecutorChooser implements EventExecutorChooser {

    /**
     * 自增序列
     */
    private final AtomicInteger idx = new AtomicInteger();
    /**
     * EventExecutor 数组
     */
    private final EventExecutor[] executors;
```

```
    PowerOfTwoEventExecutorChooser(EventExecutor[] executors) {
        this.executors = executors;
    }

    @Override
    public EventExecutor next() {
        return executors[idx.getAndIncrement() & exe ors.length - 1];
    }
```

文章目录

数组 大小 - 1】进行进行 & 并操作。

符

那 减一后，除了最高位是 0 ，剩余位都为 1 (例如 8 减

无 如何递增，再进行 & 并操作，都不会超过

代码如下：

{

oo ype 。

```
@Override
public EventExecutor next() {
    return chooser.next();
}
```

5.6 iterator

`#iterator()` 方法，获得 EventExecutor 数组的迭代器。代码如下：

```
@Override
public Iterator<EventExecutor> iterator() {
    return readonlyChildren.iterator();
}
```

- 为了避免调用方，获得迭代器后，对 EventExecutor 数组进行修改，所以返回是**不可变**的 EventExecutor 数组 `readonlyChildren` 的迭代器。

5.7 executorCount

#executorCount() 方法，获得 EventExecutor 数组的大小。代码如下：

```
public final int executorCount() {
    return children.length;
}
```

5.8 newChild

方法，创建 EventExecutor 对象。代码如下：

```
... executor, Object... args) throws Exception;
```

象。

...TimeUnit unit)

...Group 接口，EventLoop 的分组接口。代码如下：

```
// ========== 自定义接口 ==========

/**
 * Register a {@link Channel} with this {@link EventLoop}. The returned {@link ChannelFuture}
 * will get notified once the registration was complete.
 */
ChannelFuture register(Channel channel);
ChannelFuture register(ChannelPromise promise);
@Deprecated
ChannelFuture register(Channel channel, ChannelPromise promise);

// ========== 实现自 EventExecutorGroup 接口 ==========

@Override
EventLoop next();
```

- #next() 方法，选择下一个 EventLoop 对象。
- #register(...) 方法，注册 Channel 到 EventLoopGroup 中。实际上，EventLoopGroup 会分配一个 EventLoop 给该 Channel 注册。

7. MultithreadEventLoopGroup

`io.netty.channel.MultithreadEventLoopGroup` ，实现 EventLoopGroup 接口，继承 MultithreadEventExecutorGroup 抽象类，**基于多线程**的 EventLoop 的分组抽象类。

7.1 构造方法

- `DEFAULT_EVENT_LOOP_THREADS` 属性，EventLoopGroup 默认拥有的 EventLoop 数量。因为一个 EventLoop 对应一个线程，所以为 CPU 数量 * 2 。
  - 为什么会 * 2 呢？因为目前 CPU 基本都是超线程，**一个 CPU 可对应 2 个线程。**
  - 在构造方法未传入 `nThreads` 方法参数时，使用 `DEFAULT_EVENT_LOOP_THREADS` 。

7.2 newDefaultThreadFactory

newDefaultThreadFactory

`#newDefaultThreadFactory()` 方法，创建线程工厂对象。代码如下：

```
@Override
protected ThreadFactory newDefaultThreadFactory() {
    return new DefaultThreadFactory(getClass(), Thread.MAX_PRIORITY);
}
```

- 覆盖父类方法，增加了线程优先级为 `Thread.MAX_PRIORITY` 。

7.3 next

`#next()` 方法，选择下一个 EventLoop 对象。代码如下：

```
@Override
public EventLoop next() {
    return (EventLoop) super.next();
}
```


- 覆盖父类方法，将返回值转换成 EventLoop 类。

方法，创建 EventExecutor 对象。代码如下：

```
, Object... args) throws Exception;
```

实际上，EventLoopGroup 会分配一个 EventLoop 给该

```
@Deprecated
@Override
public ChannelFuture register(Channel channel, ChannelPromise promise) {
    return next().register(channel, promise);
}
```

- Channel 注册的 EventLoop ，通过 `#next()` 方法来选择。

8. NioEventLoopGroup

`io.netty.channel.nio.NioEventLoopGroup` ，继承 MultithreadEventLoopGroup 抽象类，NioEventLoop 的分组实现类。

8.1 构造方法

```
public NioEventLoopGroup() {
    this(0);
```

```
}

public NioEventLoopGroup(int nThreads) {
    this(nThreads, (Executor) null);
}

public NioEventLoopGroup(int nThreads, ThreadFactory threadFactory) {
    this(nThreads, threadFactory, SelectorProvider.pr ider());
}

                                                      ec r) {
                                                      id ));

                                                        f l SelectorProvider selectorProvider) {
                                                      , D aultSelectStrategyFactory.INSTANCE);

                                                      ry readFactory,
                                                      . S ctStrategyFactory selectStrategyFactory) {
                                                      r, lectStrategyFactory, RejectedExecutionHandlers.

                                                      ec r, final SelectorProvider selectorProvider) {
                                                      au electStrategyFactory.INSTANCE);

                                                      ec r, final SelectorProvider selectorProvider,
                                                      ry lectStrategyFactory) {
                                                      le trategyFactory, RejectedExecutionHandlers.rejec

public NioEventLoopGroup(int nThreads, Executor executor, EventExecutorChooserFactory chooserFactory,
                         final SelectorProvider selectorProvider,
                         final SelectStrategyFactory selectStrategyFactory) {
    super(nThreads, executor, chooserFactory, selectorProvider, selectStrategyFactory,
            RejectedExecutionHandlers.reject());
}

public NioEventLoopGroup(int nThreads, Executor executor, EventExecutorChooserFactory chooserFactory,
                         final SelectorProvider selectorProvider,
                         final SelectStrategyFactory selectStrategyFactory,
                         final RejectedExecutionHandler rejectedExecutionHandler) {
    super(nThreads, executor, chooserFactory, selectorProvider, selectStrategyFactory, rejectedExecuti
}
```

文章目录

- 构造方法比较多，主要是明确了父构造方法的 `Object ... args` 方法参数：
  - 第一个参数，`selectorProvider` ，`java.nio.channels.spi.SelectorProvider` ，用于创建 Java NIO Selector 对象。
  - 第二个参数， `selectStrategyFactory` ， `io.netty.channel.SelectStrategyFactory` ，选择策略工厂。详细解析，见后续文章。

- 第三个参数， `rejectedExecutionHandler` ， `io.netty.channel.SelectStrategyFactory` ，拒绝执行处理器。详细解析，见后续文章。

8.2 newChild

`#newChild(Executor executor, Object... args)` 方法，创建 NioEventLoop 对象。代码如下：

```
@Override
protected EventLoop newChild(Executor executor, Object... args) throws Exception {
    return new NioEventLoop(this, executor, (SelectorProvider) args[0],
        ((SelectStrategyFactory) args[1]).newSelectStrategy(), (Reject
```

创建

要的参数。

务占用执行时间的比例。代码如下：

```
me
ent for I/O in the child event loops.  The defa
pend the same amount of time for I/O as for non
```


`#rebuildSelectors()` 方法，重建所有 EventLoop 的 Selector 对象。代码如下：

```
/**
 * Replaces the current {@link Selector}s of the child event loops with newly created {@link Selector}
 * around the  infamous epoll 100% CPU bug.
 */
public void rebuildSelectors() {
    for (EventExecutor e: this) {
        ((NioEventLoop) e).rebuildSelector();
    }
}
```

- 因为 JDK 有 epoll 100% CPU Bug 。实际上，NioEventLoop 当触发该 Bug 时，也会**自动**调用 `NioEventLoop#rebuildSelector()` 方法，进行重建 Selector 对象，以修复该问题。

666. 彩蛋

还是比较简单的文章。如果有不清晰的地方，也可以阅读如下文章：

- 永顺 《Netty 源码分析之 三 我就是大名鼎鼎的 EventLoop(一)》 的 「NioEventLoopGroup 实例化过程」 小节。
- Hypercube 《自顶向下深入分析Netty（四）—— EventLoop-1》

文章目录