- NETTY

精尽 Netty 源码解析 —— ChannelPipeline（四）之 Outbound 事件的传播

1. 概述

本文我们来分享，在 pipeline 中的 **Outbound 事件的传播**。我们先来回顾下 Outbound 事件的定义：

> 老芳芳：A01、A02 等等，是我们每条定义的编号。

- [x] A01：Outbound 事件是【请求】事件(由 Connect 发起一个请求, 并最终由 Unsafe 处理这个请求)

> 老芳芳：A01 = A02 + A03

文章目录

1. 概述
2. ChannelOutboundInvoker
3. DefaultChannelPipeline
4. AbstractChannelHandlerContext
5. HeadContext
6. 关于其他 Outbound 事件
666. 彩蛋

nel

fe

传输方向是 `tail -> head`

时, 如果这个 Handler 不是最后一个 Handler ，则需要调用 `ctx.xxx` (例如

如果不这样做, 那么此事件的传播会提前终止.

T_EVT -> Connect.findContextOutbound ->

tHandler.OUT_EVT -> nextContext.OUT_EVT

下面，我们来跟着代码，理解每条定义。

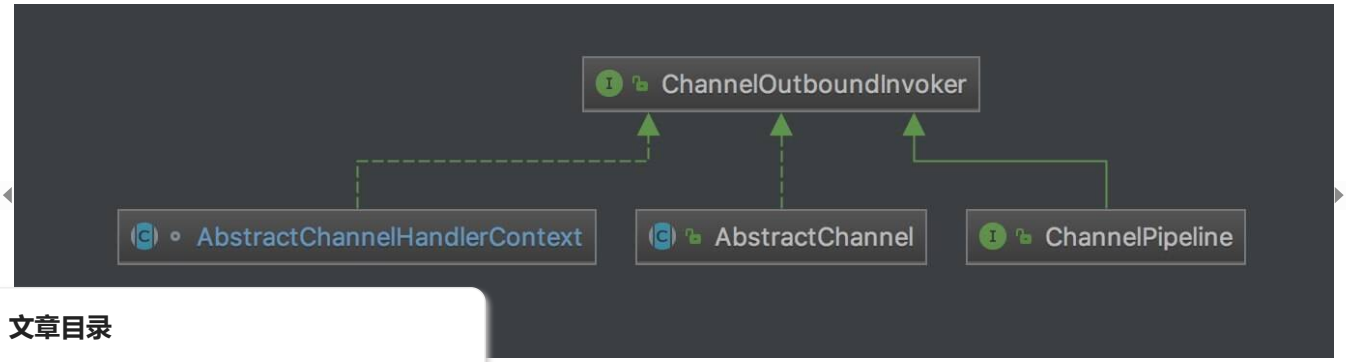
2. ChannelOutboundInvoker

在 `io.netty.channel.ChannelOutboundInvoker` 接口中，定义了所有 Outbound 事件对应的方法：

```
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);

ChannelOutboundInvoker read();

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

ChannelOutboundInvoker flush();

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

而 ChannelOutboundInvoker 的**部分**子类/接口如下图：

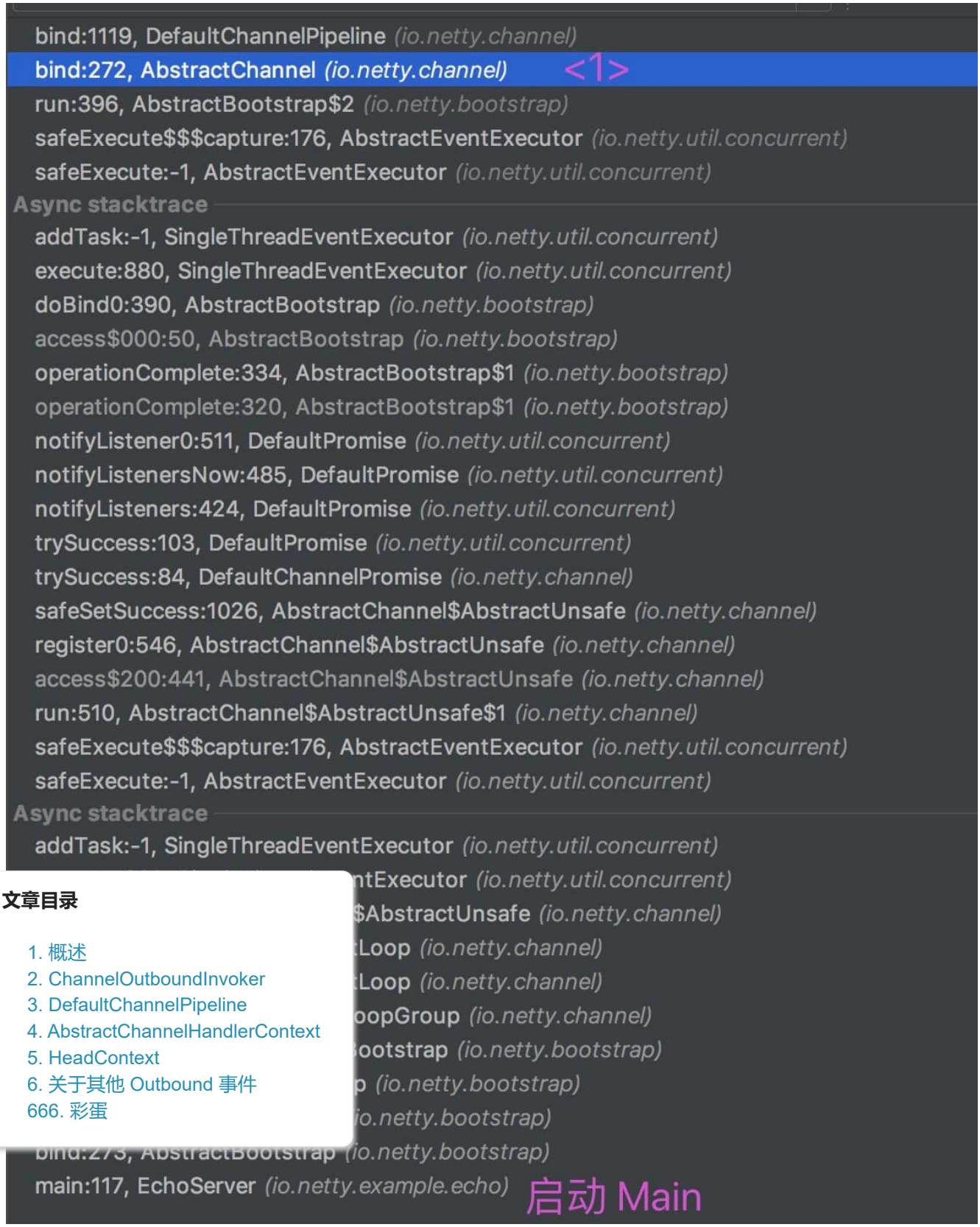


文章目录

1. 概述
2. ChannelOutboundInvoker
3. DefaultChannelPipeline
4. AbstractChannelHandlerContext
5. HeadContext
6. 关于其他 Outbound 事件
666. 彩蛋

lPipeline、AbstractChannelHandlerContext 都继承/实现了该接口。那这意味

服务端》 中，我们可以看到 Outbound 事件的其中之一 **bind** ，本文就以 **bind**



文章目录

1. 概述
2. ChannelOutboundInvoker
3. DefaultChannelPipeline
4. AbstractChannelHandlerContext
5. HeadContext
6. 关于其他 Outbound 事件
666. 彩蛋

- `AbstractChannel#bind(SocketAddress localAddress, ChannelPromise promise)` 方法，代码如下：

```
@Override
public ChannelFuture bind(SocketAddress localAddress, ChannelPromise promise) {
    return pipeline.bind(localAddress, promise);
}
```

  - `AbstractChannel#bind(SocketAddress localAddress, ChannelPromise promise)` 方法，实现的自 ChannelOutboundInvoker 接口。
    - Channel 是 **bind** 的发起者，**这符合 Outbound 事件的定义 A02** 。

- 在方法内部，会调用 `ChannelPipeline#bind(SocketAddress localAddress, ChannelPromise promise)` 方法，而这个方法，也是实现的自 ChannelOutboundInvoker 接口。*从这里可以看出，对于 ChannelOutboundInvoker 接口方法的实现，Channel 对它的实现，会调用 ChannelPipeline 的对应方法*(有一点绕，胖友理解下)。
  - 那么接口下，让我们看看 `ChannelPipeline#bind(SocketAddress localAddress, ChannelPromise promise)` 方法的具体实现。

3. DefaultChannelPipeline

`DefaultChannelPipeline#bind(SocketAddress localAddress, ChannelPromise promise)` 方法的实现，代码如下：

```
@Override
public final ChannelFuture bind(SocketAddress localAddress, ChannelPromise promise) {
    return tail.bind(localAddress, promise);
}
```

- 在方法内部，会调用 `TailContext#bind(SocketAddress localAddress, ChannelPromise promise)` 方法。**这符合 Outbound 事件的定义 A04** 。
  - 实际上，TailContext 的该方法，继承自 AbstractChannelHandlerContext 抽象类，而 AbstractChannelHandlerContext 实现了 ChannelOutboundInvoker 接口。*从这里可以看出，对于 ChannelOutboundInvoker 接口方法的实现，ChannelPipeline 对它的实现，会调用 AbstractChannelHandlerContext 的对应方法*(有一点绕，胖友理解下)。

4. AbstractChannelHandlerContext

`AbstractChannelHandlerContext#bind(SocketAddress localAddress, ChannelPromise promise)` 方法的实现，代码如下：

文章目录

1. 概述
2. ChannelOutboundInvoker
3. DefaultChannelPipeline
4. AbstractChannelHandlerContext
5. HeadContext
6. 关于其他 Outbound 事件
666. 彩蛋

```
 SocketAddress localAddress, final ChannelPromise promise) {

eption("localAddress");

对象
e, false)) {

 9:         return promise;
10:     }
11:
12:     // 获得下一个 Outbound 节点
13:     final AbstractChannelHandlerContext next = findContextOutbound();
14:     // 获得下一个 Outbound 节点的执行器
15:     EventExecutor executor = next.executor();
16:     // 调用下一个 Outbound 节点的 bind 方法
17:     if (executor.inEventLoop()) {
18:         next.invokeBind(localAddress, promise);
19:     } else {
20:         safeExecute(executor, new Runnable() {
21:             @Override
22:             public void run() {
```

```
23:                 next.invokeBind(localAddress, promise);
24:             }
25:         }, promise, null);
26:     }
27:     return promise;
28: }
```

- 第 6 至 10 行：判断 `promise` 是否为合法的 Promise 对象。代码如下：

```
private boolean isNotValidPromise(ChannelPromise promise, boolean allowVoidPromise) {
    if (promise == null) {
        throw new NullPointerException("promise");
    }

    // Promise 已经完成
    if (promise.isDone()) {
        // Check if the promise was cancelled and if so signal that the processing of the operatic
        // should not be performed.
        //
        // See https://github.com/netty/netty/issues/2349
        if (promise.isCancelled()) {
            return true;
        }
        throw new IllegalArgumentException("promise already done: " + promise);
    }

    // Channel 不符合
    if (promise.channel() != channel()) {
        throw new IllegalArgumentException(String.format(
                does not match: %s (expected: %s)", promise.channel(), channel())


                // <1>
                ultChannelPromise.class) {



                ise instanceof VoidChannelPromise) {
                Exception(
                StringUtil.simpleClassName(VoidChannelPromise.class) + " not allowed for this oper
    }
    // 禁止 CloseFuture
    if (promise instanceof AbstractChannel.CloseFuture) {
        throw new IllegalArgumentException(
                StringUtil.simpleClassName(AbstractChannel.CloseFuture.class) + " not allowed in a
    }
    return false;
}
```

文章目录

1. 概述
2. ChannelOutboundInvoker
3. DefaultChannelPipeline
4. AbstractChannelHandlerContext
5. HeadContext
6. 关于其他 Outbound 事件
666. 彩蛋

- 虽然方法很长，重点是 `<1>` 处， `promise` 的类型为 DefaultChannelPromise 。

- 第 13 行：【重要】调用 `#findContextOutbound()` 方法，获得下一个 Outbound 节点。代码如下：

```
private AbstractChannelHandlerContext findContextOutbound() {
    // 循环，向前获得一个 Outbound 节点
    AbstractChannelHandlerContext ctx = this;
    do {
        ctx = ctx.prev;
    } while (!ctx.outbound);
    return ctx;
}
```

- 循环，**向前**获得一个 Outbound 节点。
- 循环，**向前**获得一个 Outbound 节点。
- 循环，**向前**获得一个 Outbound 节点。
- 😈 重要的事情说三遍，对于 Outbound 事件的传播，是从 pipeline 的尾巴到头部，**这符合 Outbound 事件的定义 A04** 。

- 第 15 行：调用 `AbstractChannelHandlerContext#executor()` 方法，获得下一个 Outbound 节点的执行器。代码如下：

```
// Will be set to null if no child executor should be used, otherwise it will be set to the
// child executor.
/**
 * EventExecutor 对象
 */
final EventExecutor executor;

@Override
public EventExecutor executor() {
    if (executor == null) {
        return channel().eventLoop();
    } else {
        return executor;
```

文章目录

1. 概述
2. ChannelOutboundInvoker
3. DefaultChannelPipeline
4. AbstractChannelHandlerContext
5. HeadContext
6. 关于其他 Outbound 事件

666. 彩蛋

的 EventLoop 作为执行器。😈 一般情况下，我们可以忽略**子执行器**的逻
nnel 的 EventLoop 作为执行器。

调用**下一个节点**的
okeBind(SocketAddress localAddress, ChannelPromise promise)

的线程中，会调用 `#safeExecute(EventExecutor executor, Runnable runnable, ChannelPromise promise, Object msg)` 方法，提交到 EventLoop 的线程中执行。代码如下：

```
private static void safeExecute(EventExecutor executor, Runnable runnable, ChannelPromise pro
    try {
        // 提交 EventLoop 的线程中，进行执行任务
        executor.execute(runnable);
    } catch (Throwable cause) {
        try {
            // 发生异常，回调通知 promise 相关的异常
            promise.setFailure(cause);
        } finally {
            // 释放 msg 相关的资源
```

```
            if (msg != null) {
                ReferenceCountUtil.release(msg);
            }
        }
    }
}
```

- x

AbstractChannelHandlerContext#invokeBind(SocketAddress localAddress, ChannelPromise promise) 方法，代码如下：

```
1: private void invokeBind(SocketAddress localAddress, ChannelPromise promise) {
2:     if (invokeHandler()) { // 判断是否符合的 ChannelHandler
3:         try {
4:             // 调用该 ChannelHandler 的 bind 方法
5:             ((ChannelOutboundHandler) handler()).bind(this, localAddress, promise);
6:         } catch (Throwable t) {
7:             notifyOutboundHandlerException(t, promise); // 通知 Outbound 事件的传播，发生异常
8:         }
9:     } else {
10:        // 跳过，传播 Outbound 事件给下一个节点
11:        bind(localAddress, promise);
12:    }
13: }
```

- 第 2 行：调用 #invokeHandler() 方法，判断是否符合的 ChannelHandler 。代码如下：

```
/**
                                                detect if {@link ChannelHandler#handlerAdded(ChannelHandlerContex
                                                e} and if called or could not detect return {@code true}.

                                                false} we will not invoke the {@link ChannelHandler} but just for
                                                ltChannelPipeline} may already put the {@link ChannelHandler} in
                                                andler#handlerAdded(ChannelHandlerContext)}.


                                                reduce volatile reads.
                                                erState;
    return handlerState == ADD_COMPLETE || (!ordered && handlerState == ADD_PENDING);
}
```

  - 对于 ordered = true 的节点，必须 ChannelHandler 已经添加完成。
  - 对于 ordered = false 的节点，没有 ChannelHandler 的要求。
- 第 9 至 12 行：若是**不符合**的 ChannelHandler ，则**跳过**该节点，调用 AbstractChannelHandlerContext#bind(SocketAddress localAddress, ChannelPromise promise) 方法，传播 Outbound 事件给下一个节点。即，又回到 「4. AbstractChannelHandlerContext」 的开头。
- 第 2 至 8 行：若是**符合**的 ChannelHandler ：
  - 第 5 行：调用 ChannelHandler 的 #bind(ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise promise) 方法，处理 bind 事件。

文章目录

1. 概述
2. ChannelOutboundInvoker
3. DefaultChannelPipeline
4. AbstractChannelHandlerContext
5. HeadContext
6. 关于其他 Outbound 事件
666. 彩蛋

- 😈 实际上，此时节点的数据类型为 DefaultChannelHandlerContext 类。若它被认为是 Outbound 节点，那么他的处理器的类型会是 **ChannelOutboundHandler** 。而 `io.netty.channel.ChannelOutboundHandler` 类似 ChannelOutboundInvoker ，定义了对每个 Outbound 事件的处理。代码如下：

```
void bind(ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise promise)

void connect(ChannelHandlerContext ctx, SocketAddress remoteAddress, SocketAddress localA

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 Exceptic

void flush(ChannelHandlerContext ctx) throws Exception;
```

  - 胖友自己对比下噢。

- 如果节点的 `ChannelOutboundHandler#bind(ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise promise)` 方法的实现，不调用 `AbstractChannelHandlerContext#bind(SocketAddress localAddress, ChannelPromise promise)` 方法，就不会传播 Outbound 事件给下一个节点。**这就是 Outbound 事件的定义 A05** 。可能有点绕，我们来看下 Netty LoggingHandler 对该方法的实现代码：

```
final class LoggingHandler implements ChannelInboundHandler, ChannelOutboundHandler {

andlerContext ctx, SocketAddress localAddress, ChannelPromis

lAddress=" + localAddress);
下一个节点
, promise); // <1>

}
```

文章目录

1. 概述
2. ChannelOutboundInvoker
3. DefaultChannelPipeline
4. AbstractChannelHandlerContext
5. HeadContext
6. 关于其他 Outbound 事件
666. 彩蛋

  - 如果把 `<1>` 处的代码去掉，bind 事件将不会传播给下一个节点！！！ **一定要注意。**
- 这块的逻辑非常重要，如果胖友觉得很绕，一定要自己多调试 + 调试 + 调试。

- 第 7 行：如果发生异常，调用 `#notifyOutboundHandlerException(Throwable, Promise)` 方法，通知 Outbound 事件的传播，发生异常。详细解析，见 《精尽 Netty 源码解析 —— ChannelPipeline（六）之异常事件的传播》 。

本小节的整个代码实现，**就是 Outbound 事件的定义 A06**的体现。而随着 Outbound 事件在节点不断从 pipeline 的尾部到头部的传播，最终会到达 HeadContext 节点。

5. HeadContext

`HeadContext#bind(ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise promise)` 方法，代码如下：

```
@Override
public void bind(ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise promise) throws
    unsafe.bind(localAddress, promise);
}
```

- 调用 `Unsafe#bind(SocketAddress localAddress, ChannelPromise promise)` 方法，进行 bind 事件的处理。也就是说 Unsafe 是 **bind** 的处理着，**这符合 Outbound 事件的定义 A03** 。
- 而后续的逻辑，就是《精尽 Netty 源码分析 —— 启动（一）之服务端》的「3.13.2 doBind0」小节，从 `Unsafe#bind(SocketAddress localAddress, ChannelPromise promise)` 方法，开始。
- 至此，整个 pipeline 的 Outbound 事件的传播结束。

6. 关于其他 Outbound 事件

本文暂时只分享了 **bind** 这个 Outbound 事件。剩余的其他事件，胖友可以自己进行调试和理解。例如：**connect** 事件，并且结合《精尽 Netty 源码分析 —— 启动（二）之客户端》一文。

666. 彩蛋

*推荐阅读文章：

- 闪电侠《netty 源码分析之 pipeline(二)》*

文章目录

1. 概述
2. ChannelOutboundInvoker
3. DefaultChannelPipeline
4. AbstractChannelHandlerContext
5. HeadContext
6. 关于其他 Outbound 事件
666. 彩蛋

次