OP_READ 注册

Tomcat 注册读事件的处理逻辑比较简单一些,入口 `NioEndpoint.Acceptor#run()` 方法:

```
1.  @Override
2.  public void run() {
3.      SocketChannel socket = null;
4.      try {
5.          socket = serverSock.accept();
6.      }
7.
8.      if (running && !paused) {
9.          if (!setSocketOptions(socket)) {    // XXX: 处理接收到的连接
10.         }
11.     }
12. }
```

setSocketOptions() 方法:

```
1.  protected boolean setSocketOptions(SocketChannel socket) {
2.      getPoller0().register(channel);
3.  }
```

`NioEndpoint.Poller#register()` 的操作就是将 `PollerEvent` 添加到 `Poller` 的队列中.

```
1.  public void register(final NioChannel socket) {
2.      PollerEvent r = eventCache.pop();
3.      // 转入 OP_READ
4.      ka.interestOps(SelectionKey.OP_READ); // this is what OP_REGISTER turns into.
5.      if (r == null)
6.          r = new PollerEvent(socket, ka, OP_REGISTER);
7.      else
8.          r.reset(socket, ka, OP_REGISTER);
9.
10.     addEvent(r);    // 添加到事件队列
11. }
```

注意 此时 PollerEvent 中的 interestOps 变量值为 OP_REGISTER .

`Poller` 线程会取出 `PollerEvent` 事件并执行.

看下 `PollerEvent#run()` 所做的操作:

```
1.  @Override
2.  public void run() {
3.      if (interestOps == OP_REGISTER) {
4.          try {
5.              // XXX: 注册 OP_READ 事件
6.              socket.getIOChannel().register(socket.getPoller().getSelector(), SelectionKey.O
    P_READ, key);
7.          } catch (Exception x) {
8.
9.          }
10.
11.     }
```

```
12.     }
```

上面已经说过 `interestOps` 的值为 `OP_REGISTER`，自然这里就是执行注册 `OP_READ`.

数据的读取与 OP_READ 的注销及重新注册

NioEndpoint.Poller#run()

```
1.  @Override
2.  public void run() {
3.      ...
4.
5.      while (iterator != null && iterator.hasNext()) {
6.          SelectionKey sk = iterator.next();
7.
8.          KeyAttachment attachment = (KeyAttachment) sk.attachment();
9.          if (attachment == null) {
10.             iterator.remove();
11.         } else {
12.             attachment.access();
13.             iterator.remove();
14.             // XXX: 处理事件
15.             processKey(sk, attachment);
16.         }
17.     }// while
18.
19.     ...
20. }
```

processKey()

```
1.  NioChannel channel = attachment.getChannel();
2.  if (sk.isReadable() || sk.isWritable()) {
3.      if (...) {
4.      } else {
5.          if (isWorkerAvailable()) {
6.              // 注销事件
7.              unreg(sk, attachment, sk.readyOps());
8.              if (sk.isReadable()) {
9.                  if (!processSocket(channel, SocketStatus.OPEN_READ, true)) {
10.                 }
11.             }
12.
13.         }
14.     }
15. }
```

这里最主要的就是 `unreg()` 方法,看看它做了哪些事情:

```
1.  protected void unreg(SelectionKey sk, KeyAttachment attachment, int readyOps) {
2.      // XXX: 注销事件 -> 防止多线程干扰
3.      // this is a must, so that we don't have multiple threads messing
4.      // with the socket
5.      reg(sk, attachment, sk.interestOps() & (~readyOps));
6.  }
7.
```

```
    protected void reg(SelectionKey sk, KeyAttachment attachment, int intops) {
        sk.interestOps(intops);    // 更新 Ops -> 这里就是注销 OP_READ
        attachment.interestOps(intops);
    }
```

很明显, 这里有个 `sk.interestOps() & (~readyOps)` 操作, 也就是将 `OP_READ` 事件给注销掉.然后继续看下 `processSocket()` 方法:

```
    protected boolean processSocket(NioChannel socket, SocketStatus status, boolean dispatch) {
        try {
            SocketProcessor sc = processorCache.pop();
            if (sc == null)
                sc = new SocketProcessor(socket, status);
            else
                sc.reset(socket, status);

            Executor executor = getExecutor();
            if (dispatch && executor != null) {
                // 交由线程池执行
                executor.execute(sc);
            } else {
                sc.run();
            }
        }
    }
```

`SocketProcessor` 就是读取数据,处理请求.

```
    @Override
    public void run() {
        synchronized (socket) {
            // XXX: 处理
            doRun(key, ka);
        }
    }

    // 省略了大部分的代码
    private void doRun(SelectionKey key, KeyAttachment ka) {
        SocketState state = SocketState.OPEN;
        // 执行 Http11ConnectionHandler#process() -> 数据的读取, 解析, 处理
        state = handler.process(ka, status);
    }
```

如果我们完整的看 `doRun()` 方法的话发现没有重新注册 `OP_READ` 的操作.难道没有重新注册 `OP_READ` ,那肯定不是.因为之前知道 `Poller` 线程注册了 `OP_READ` 事件,那么重新注册 `OP_READ` 的操作是不是也在这个类中呢? 好,让我们重新回到 `Poller#run()` 方法中来:

```
    @Override
    public void run() {
        if (interestOps == OP_REGISTER) {
            try {
                // XXX: 注册 OP_READ 事件
                socket.getIOChannel().register(socket.getPoller().getSelector(), SelectionKey.O
    P_READ, key);
            }
        } else {
```

```
            final SelectionKey key = socket.getIOChannel().keyFor(socket.getPoller().getSelecto
    r());
            try {
                if (key != null) {
                    final KeyAttachment att = (KeyAttachment) key.attachment();
                    if (att != null) {
                        interestOps = (interestOps & (~OP_CALLBACK));

                        int ops = key.interestOps() | interestOps;
                        att.interestOps(ops);
                        if (att.getCometNotify())
                            key.interestOps(0);
                        else {
                            System.err.println("----------Update Key: " + key.interestOps() + "
     -> " + ops);
                            key.interestOps(ops);    // XXX: 这里就是更新 key 为 OP_READ
                        }
                    }
                }
            }
        }
    }
```

这里我们主要关注 `else` 部分的代码, 发现与 `SelectionKey` 更新 ops 操作有两个, 一个是 `key.interestOps(0)` ,显然在这里应该不是它, 那就是 `key.interestOps(ops);` 咯? 是不是测试下即可, 这里我们加上了打印语句.测试发现确实一个请求处理完之后会打印如下的内容,也就是重新注册了 `OP_READ` :

```
    ----------Update Key: 0 -> 1
```

但是这个操作是咋产生呢? 显然,既然是通过 `Poller` 执行,那么肯定是被添加到了 `Poller` 的事件队列中,而此类中有个 `addEvent()` 方法可以完成添加事件到队列中的操作.那么就是说在请求处理完成之后到底在哪里调用了 `Poller#addEvent()` 方法? 如果想了解的话,那么可以深入 `Http11ConnectionHandler#process()` 方法.如果实在目前不想关注这个处理的流程,该咋办?

那么这里有一个小技巧, 通过如下的一个工具类:

```
    public class MyDebug {

        public static void printStackTrace() {
            String info = getStackTrace();
            System.err.println(info);
        }

        public static String getStackTrace() {
            StringBuilder buf = new StringBuilder();
            buf.append("------------------------------------------------------------");
            buf.append("\r\n");
            StackTraceElement[] es = Thread.currentThread().getStackTrace();

            for (StackTraceElement ste : es) {
                buf.append("\tat ");
                buf.append(ste.getClassName());
                buf.append(".");
                buf.append(ste.getMethodName());
                buf.append("(");
                buf.append(ste.getFileName());
```

```
                buf.append(":");
                buf.append(ste.getLineNumber());
                buf.append(")");
                buf.append("\r\n");
            }
            buf.append("-----------------------------------------------------");

            return buf.toString();
        }
    }
```

现在在 `Poller#addEvent()` 中添加如下的代码:

```
    private void addEvent(PollerEvent event) {
        MyDebug.printStackTrace();

        events.offer(event);
    }
```

现在在测试下,我们就看到了原来是这样的:

```
-----------------------------------------------------
    at java.lang.Thread.getStackTrace(Thread.java:1552)
    at mydebug.MyDebug.getStackTrace(MyDebug.java:16)
    at mydebug.MyDebug.printStackTrace(MyDebug.java:8)
    at org.apache.tomcat.util.net.NioEndpoint$Poller.addEvent(NioEndpoint.java:979)
    at org.apache.tomcat.util.net.NioEndpoint$Poller.add(NioEndpoint.java:1018)
    at org.apache.tomcat.util.net.NioEndpoint$Poller.add(NioEndpoint.java:1006)
    at org.apache.coyote.http11.Http11NioProtocol$Http11ConnectionHandler.release(Http11NioProtocol.java:228)
    at org.apache.coyote.AbstractProtocol$AbstractConnectionHandler.process(AbstractProtocol.java:687)
    at org.apache.coyote.http11.Http11NioProtocol$Http11ConnectionHandler.process(Http11NioProtocol.java:209)
    at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1830)
    at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.run(NioEndpoint.java:1783) <2 internal calls>
    at java.lang.Thread.run(Thread.java:745)
-----------------------------------------------------
----------Update Key: 0 -> 1
```

可以看到这里就是将一个重新注册 `OP_READ` 事件的任务(`PollerEvent`)添加到队列中,并会由 `Poller` 执行. 再汇过来看看 `Poller#run()` 方法, 现在以下面的 `(1) -> (2)` 顺序来看.

```
    @Override
    public void run() {
        while (true) {
            try {
                boolean hasEvents = false;

                if (close) {
                } else {
                    // (2) 这里执行 PollerEvent, 重新注册 OP_READ 事件
                    hasEvents = events();
                }

                try {
                    if (!close) {
                        if (wakeupCounter.getAndSet(-1) > 0) {
                            keyCount = selector.selectNow();
                        } else {
                            keyCount = selector.select(selectorTimeout);
                        }
                    }
                }
```

```
22.
23.                 if (keyCount == 0)
24.                     hasEvents = (hasEvents | events());
25.
26.                 Iterator<SelectionKey> iterator = keyCount > 0 ? selector.selectedKeys().iterat
    or() : null;
27.                 while (iterator != null && iterator.hasNext()) {
28.                     SelectionKey sk = iterator.next();
29.
30.                     KeyAttachment attachment = (KeyAttachment) sk.attachment();
31.                     if (attachment == null) {
32.                         iterator.remove();
33.                     } else {
34.                         iterator.remove();
35.                         // (1) 处理请求 -> 会注销 OP_READ 事件
36.                         processKey(sk, attachment);
37.                     }
38.                 }
39.             }
40.         }
41.
42.     }
```

总结

可以看到 Tomcat 在接受到 `OP_READ` 事件之后就将其注销掉了, 在处理完请求之后再重新注册 `OP_READ` 等待后续的操作.