11.jetty分析

1. new Server()

```
public Server(@Name("port")int port)
{
    this((ThreadPool)null);
    ServerConnector connector=new ServerConnector(this);
    connector.setPort(port);
    setConnectors(new Connector[]{connector});
}
```

1.1. 初始化线程池

```
public Server(@Name("threadpool") ThreadPool pool)
{
    _threadPool=pool!=null?pool:new QueuedThreadPool();
    addBean(_threadPool);
    setServer(this);
}
```

1.1.1. QueuedThreadPool

参见: 启动QueuedThreadPool

实现了SizedThreadPool

execute()方法

```
@Override
public void execute(Runnable job)
{
    if (!isRunning() || !_jobs.offer(job))
    {
        LOG.warn("{} rejected {}", this, job);
        throw new RejectedExecutionException(job.toString());
    }
    else
    {
        // Make sure there is at least one thread executing the job.
        if (getThreads() == 0)
            startThreads(1);
    }
}
```

BlockingQueue

将任务推入
BlockingQueue<Runnable> org.eclipse.jetty.util.thread.QueuedThreadPool._jobs

1.2. 初始化ServerConnector

HTTP connector using NIO ByteChannels and Selectors
继承自 AbstractConnector

1.2.1. 初始化ScheduledExecutorScheduler

based on JDK's {@link ScheduledThreadPoolExecutor}.

1.2.2. 初始化ByteBufferPool

在数据传输过程中，不可避免需要byte数组

buffer池

默认产生 ArrayByteBufferPool

ByteBufferPool 接口有2个方法：

```
public ByteBuffer acquire(int size, boolean direct);
public void release(ByteBuffer buffer);
```

这是一个很好的对象池范本

ArrayByteBufferPool

public ArrayByteBufferPool(int minSize, int increment, int maxSize)

```
public ArrayByteBufferPool()
{
    this(0,1024,64*1024);
}

  _direct=new Bucket[maxSize/increment];
  _indirect=new Bucket[maxSize/increment];
```

结构

Bucket

_direct Bucket数组

_indirect   Bucket数组

为每一个大小，新建一个Bucket

但不初始化ByteBuffer

```
    int size=0;
    for (int i=0;i<_direct.length;i++)
    {
      size+=_inc;
      _direct[i]=new Bucket(size);
      _indirect[i]=new Bucket(size);
    }
```

一个Bucekt存放**大小相同**的所有的ByteBuffer

_size

bytebuffer大小

_queue

```
public final Queue<ByteBuffer> _queue= new ConcurrentLinkedQueue<>();
```

acquire

```
public ByteBuffer acquire(int size, boolean direct)
```

取得合适的Bucket

每个Bucket的大小不同，这里找到最合适的

```
Bucket bucket = bucketFor(size,direct);
```

从Bucket中取得ByteBuffer

```
ByteBuffer buffer = bucket==null?null:bucket._queue.poll();
```

不存在则新建

```
    if (buffer == null)
    {
      int capacity = bucket==null?size:bucket._size;
      buffer = direct ? BufferUtil.allocateDirect(capacity) : BufferUtil.allocate(capacity);
    }
```

release

```
public void release(ByteBuffer buffer)
{
  if (buffer!=null)
  {
    Bucket bucket = bucketFor(buffer.capacity(),buffer.isDirect());
    if (bucket!=null)
    {
      BufferUtil.clear(buffer);
      bucket._queue.offer(buffer);
    }
  }
}
```

取得合适的Bucket

```
Bucket bucket = bucketFor(buffer.capacity(),buffer.isDirect());
```

清空Buffer

```
BufferUtil.clear(buffer);
```

归还Pool

```
bucket._queue.offer(buffer);
```

例外处理

如果申请的ByteBuffer过大或者过小，

无法在POOL中满足，则可以申请成功，但无法归还给POOL。

1.2.3. 维护ConnectionFactory

HttpConnectionFactory

用于创建连接，
比如Accept后，需要创建一个表示连接的对象

1.2.4. 取得可用CPU数量

```
int cores = Runtime.getRuntime().availableProcessors();
```

1.2.5. 更新acceptor数量

```
if (acceptors < 0)
    acceptors=Math.max(1, Math.min(4,cores/8));
```

1.2.6. 创建acceptor线程组

参见: 创建Acceptor线程

```
_acceptors = new Thread[acceptors];
```

1.2.7. 初始化ServerConnectorManager

继承自 SelectorManager

```
 _manager = new ServerConnectorManager(getExecutor(), getScheduler(),
        selectors>0?selectors:Math.max(1,Math.min(4,Runtime.getRuntime().availableProcessors()/2)));
```

保存selector线程数量

Math.min(4,Runtime.getRuntime().availableProcessors()/2))

1.3. 设置port

connector.setPort(port);

1.4. 关联Sever和Connector

setConnectors(new Connector[]{connector});

2. Server.start()

org.eclipse.jetty.server.Server
启动web服务器

```
WebAppContext context = new WebAppContext();


context.setContextPath("/");
context.setResourceBase("./web/");
context.setClassLoader(Thread.currentThread().getContextClassLoader());
server.setHandler(context);
server.start();
```

2.1. 设置启动状态

AbstractLifeCycle

```
private void setStarting()
  {
    if (LOG.isDebugEnabled())
      LOG.debug("starting {}",this);
    _state = __STARTING;
    for (Listener listener : _listeners)
      listener.lifeCycleStarting(this);
  }
```

2.2. 启动过程doStart()

Server
启动整个server

```
protected void doStart() throws Exception
```

```
  {
    //If the Server should be stopped when the jvm exits, register
    //with the shutdown handler thread.
    if (getStopAtShutdown())
      ShutdownThread.register(this);

    //Register the Server with the handler thread for receiving
    //remote stop commands
    ShutdownMonitor.register(this);

    //Start a thread waiting to receive "stop" commands.
    ShutdownMonitor.getInstance().start(); // initialize

    LOG.info("jetty-" + getVersion());
    HttpGenerator.setJettyVersion(HttpConfiguration.SERVER_VERSION);
    MultiException mex=new MultiException();

    // check size of thread pool
    SizedThreadPool pool = getBean(SizedThreadPool.class);
    int max=pool==null?-1:pool.getMaxThreads();
    int selectors=0;
    int acceptors=0;
    if (mex.size()==0)
    {
      for (Connector connector : _connectors)
      {
        if (connector instanceof AbstractConnector)
          acceptors+=((AbstractConnector)connector).getAcceptors();

        if (connector instanceof ServerConnector)
          selectors+=((ServerConnector)connector).getSelectorManager().getSelectorCount();
      }
    }

    int needed=1+selectors+acceptors;
    if (max>0 && needed>max)
      throw new IllegalStateException(String.format("Insufficient threads: max=%d < needed(acceptors=%d +
selectors=%d + request=1)",max,acceptors,selectors));

    try
    {
      super.doStart();
    }
    catch(Throwable e)
    {
        mex.add(e);
    }

    // start connectors last
    for (Connector connector : _connectors)
    {
        try
        {
            connector.start();
        }
        catch(Throwable e)
        {
            mex.add(e);
        }
    }

    if (isDumpAfterStart())
        dumpStdErr();

    mex.ifExceptionThrow();

    LOG.info(String.format("Started @%dms",Uptime.getUptime()));
}
```

2.2.1. 注册ShutdownMonitor

远程控制接口

```
        //Register the Server with the handler thread for receiving
        //remote stop commands
        ShutdownMonitor.register(this);

        //Start a thread waiting to receive "stop" commands.
        ShutdownMonitor.getInstance().start(); // initialize
```

2.2.2. 获取化线程池

```
        // check size of thread pool
        SizedThreadPool pool = getBean(SizedThreadPool.class);
```

QueuedThreadPool

2.2.3. 设置selector数量

根据Connector数量进行累计

大部分情况下，只有一个ServerConnector

```
for (Connector connector : _connectors)
        {
            if (connector instanceof AbstractConnector)
                acceptors+=((AbstractConnector)connector).getAcceptors();

            if (connector instanceof ServerConnector)
                selectors+=((ServerConnector)connector).getSelectorManager().getSelectorCount();
        }
```

累计所有Connector的需求

2.2.4. 计算所需的所有线程数量

```
int needed=1+selectors+acceptors;
```

如果大于默认的200则中断程序

```
if (max>0 && needed>max)
            throw new IllegalStateException(String.format("Insufficient threads: max=%d < needed(acceptors=%d
+ selectors=%d + request=1)",max,acceptors,selectors));
```

2.2.5. 维护Bean

启动QueuedThreadPool

参见: QueuedThreadPool

doStart()

startThreads()

建立需要的线程

创建线程

```
Thread thread = newThread(_runnable);
```

_runnable

_jobs中取任务并执行

设置线程的属性

```
thread.setDaemon(isDaemon());
thread.setPriority(getThreadsPriority());
thread.setName(_name + "-" + thread.getId());
_threads.add(thread);
```

启动线程

```
thread.start();
```

启动WebAppContext

如果需要使用，在此处启动

2.2.6. 启动Connector

取得ConnectionFactory

```
_defaultConnectionFactory = getConnectionFactory(_defaultProtocol);
```

创建selector线程并启动

```
for (int i = 0; i < _selectors.length; i++)
{
  ManagedSelector selector = newSelector(i);
  _selectors[i] = selector;
  selector.start();
  execute(new NonBlockingThread(selector));
}
```

newSelector()

参见: ManagedSelector处理

```
protected ManagedSelector newSelector(int id)
{
   return new ManagedSelector(id);
}
```

创建Acceptor线程

参见: 创建acceptor线程组

```
_stopping=new CountDownLatch(_acceptors.length);
for (int i = 0; i < _acceptors.length; i++)
{
   Acceptor a = new Acceptor(i);
   addBean(a);
   getExecutor().execute(a);
}
```

Acceptor

参见: Accept成功

设置线程名字

```
final Thread thread = Thread.currentThread();
String name=thread.getName();
_name=String.format("%s-acceptor-%d@%x-%s",name,_acceptor,hashCode(),AbstractConnector.this.toString());
thread.setName(_name);
```

设置优先级

将自己放入_acceptors数组

```
synchronized (AbstractConnector.this)
{
   _acceptors[_acceptor] = thread;
}
```

监听端口

```
try
{
   while (isAccepting())
   {
      try
      {
```

```
                accept(_acceptor);
            }
            catch (Throwable e)
            {
                if (isAccepting())
                    LOG.warn(e);
                else
                    LOG.ignore(e);
            }
        }
    }
    finally
    {
        thread.setName(name);
        if (_acceptorPriorityDelta!=0)
            thread.setPriority(priority);

        synchronized (AbstractConnector.this)
        {
            _acceptors[_acceptor] = null;
        }
        CountDownLatch stopping=_stopping;
        if (stopping!=null)
            stopping.countDown();
    }
```

ServerConnector.accept()

```
public void accept(int acceptorID) throws IOException
{
    ServerSocketChannel serverChannel = _acceptChannel;
    if (serverChannel != null && serverChannel.isOpen())
    {
        SocketChannel channel = serverChannel.accept();
        accepted(channel);
    }
}
```

在accept的地方等待

没有Acceptor的情况

channle默认是blocking的

如果acceptor数量为0，没有安排线程专门进行accept，则设置为非阻塞模式

若是非0，有专门线程进行accept，因此，为阻塞模式

```
protected void doStart() throws Exception
{
    super.doStart();

    if (getAcceptors()==0)
    {
        _acceptChannel.configureBlocking(false);
        _manager.acceptor(_acceptChannel);
    }
}
```

2.3. 启动完毕

AbstractLifeCycle

```
private void setStarted()
{
    _state = __STARTED;
    if (LOG.isDebugEnabled())
        LOG.debug(STARTED+" @{}ms {}",Uptime.getUptime(),this);
    for (Listener listener : _listeners)
        listener.lifeCycleStarted(this);
}
```

3. Http请求

3.1. Accept成功

参见: Acceptor

```
private void accepted(SocketChannel channel) throws IOException
{
    channel.configureBlocking(false);
    Socket socket = channel.socket();
    configure(socket);
    _manager.accept(channel);
}
```

3.1.1. 设置为非阻塞模式

```
channel.configureBlocking(false);
```

3.1.2. 配置Socket

```
Socket socket = channel.socket();
configure(socket);
```

3.1.3. 正式处理

```
SelectorManager _manager ;
_manager.accept(channel);
```

选择可用的ManagedSelector线程

```
private ManagedSelector chooseSelector()
{
    // The ++ increment here is not atomic, but it does not matter,
    // so long as the value changes sometimes, then connections will
    // be distributed over the available selectors.
    long s = _selectorIndex++;
    int index = (int)(s % getSelectorCount());
    return _selectors[index];
}
```

ManagedSelector处理

参见: newSelector()

ManagedSelector 是一个线程

封装了Selector 的使用

提交任务

参见: runChanges()

```
selector.submit(selector.new Accept(channel, attachment));
```

提交这个处理任务到ManagedSelector:

```
private final Queue<Runnable> _changes = new ConcurrentArrayQueue<>();
_changes.offer(change);
```

ConcurrentArrayQueue

与ConcurrentLinkedQueue相似的性能，但直接保存元素

而不是node，因此需要更少的对象，更少的GC

3.2. 请求处理

3.2.1. ManagedSelector.run()

```
while (isRunning())
    select();
```

select()

发现有任务就执行

```
runChanges();
```

runChanges()

参见: 提交任务

```
private void runChanges()
{
    Runnable change;
    while ((change = _changes.poll()) != null)
        runChange(change);
}
```

runChange()

```
change.run();
```

Accept.run

```
SelectionKey key = channel.register(_selector, 0, attachment);
EndPoint endpoint = createEndPoint(channel, key);
key.attach(endpoint);
```

select()

```
int selected = _selector.select();
```

处理SelectionKey

```
Set<SelectionKey> selectedKeys = _selector.selectedKeys();
for (SelectionKey key : selectedKeys)
{
    if (key.isValid())
    {
        processKey(key);
    }
    else
    {
        if (debug)
            LOG.debug("Selector loop ignoring invalid key for channel {}", key.channel());
        Object attachment = key.attachment();
        if (attachment instanceof EndPoint)
            ((EndPoint)attachment).close();
    }
}
selectedKeys.clear();
```

processKey()

```
private void processKey(SelectionKey key)
{
    Object attachment = key.attachment();
    try
    {
        if (attachment instanceof SelectableEndPoint)
        {
            ((SelectableEndPoint)attachment).onSelected();
        }
        else if (key.isConnectable())
        {
            processConnect(key, (Connect)attachment);
        }
        else if (key.isAcceptable())
```

```
            {
                processAccept(key);
            }
            else
            {
                throw new IllegalStateException();
            }
        }
        catch (CancelledKeyException x)
        {
            LOG.debug("Ignoring cancelled key for channel {}", key.channel());
            if (attachment instanceof EndPoint)
                closeNoExceptions((EndPoint)attachment);
        }
        catch (Throwable x)
        {
            LOG.warn("Could not process key for channel " + key.channel(), x);
            if (attachment instanceof EndPoint)
                closeNoExceptions((EndPoint)attachment);
        }
    }
```

onSelected()

```
    @Override
    public void onSelected()
    {
        assert _selector.isSelectorThread();
        int oldInterestOps = _key.interestOps();
        int readyOps = _key.readyOps();
        int newInterestOps = oldInterestOps & ~readyOps;
        setKeyInterests(oldInterestOps, newInterestOps);
        updateLocalInterests(readyOps, false);
        if (_key.isReadable())
            getFillInterest().fillable();
        if (_key.isWritable())
            getWriteFlusher().completeWrite();
    }
```

会使用新的线程进行HTTP业务处理 (提交到线程池)