

20 | 总结：Tomcat和Jetty中的对象池技术

2019-06-25 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)



讲述：李号双

时长 07:03 大小 5.66M




Java 对象，特别是一个比较大、比较复杂的 Java 对象，它们的创建、初始化和 GC 都需要耗费 CPU 和内存资源，为了减少这些开销，Tomcat 和 Jetty 都使用了对象池技术。所谓的对象池技术，就是说一个 Java 对象用完之后把它保存起来，之后再拿出来重复使用，省去了对象创建、初始化和 GC 的过程。对象池技术是典型的以**空间换时间**的思路。

由于维护对象池本身也需要资源的开销，不是所有场景都适合用对象池。如果你的 Java 对象数量很多并且存在的时间比较短，对象本身又比较大比较复杂，对象初始化的成本比较高，这样的场景就适合用对象池技术。比如 Tomcat 和 Jetty 处理 HTTP 请求的场景就符合这个特征，请求的数量很多，为了处理单个请求需要创建不少的复杂对象（比如 Tomcat 连接器中 SocketWrapper 和 SocketProcessor），而且一般来说请求处理的时间比较短，一旦请求处理完毕，这些对象就需要被销毁，因此这个场景适合对象池技术。

Tomcat 的 SynchronizedStack

Tomcat 用 SynchronizedStack 类来实现对象池，下面我贴出它的关键代码来帮助你理解。

 复制代码

```
1 public class SynchronizedStack<T> {
2
3     // 内部维护一个对象数组，用数组实现栈的功能
4     private Object[] stack;
5
6     // 这个方法用来归还对象，用 synchronized 进行线程同步
7     public synchronized boolean push(T obj) {
8         index++;
9         if (index == size) {
10             if (limit == -1 || size < limit) {
11                 expand(); // 对象不够用了，扩展对象数组
12             } else {
13                 index--;
14                 return false;
15             }
16         }
17         stack[index] = obj;
18         return true;
19     }
20
21     // 这个方法用来获取对象
22     public synchronized T pop() {
23         if (index == -1) {
24             return null;
25         }
26         T result = (T) stack[index];
27         stack[index--] = null;
28         return result;
29     }
30
31     // 扩展对象数组长度，以 2 倍大小扩展
32     private void expand() {
33         int newSize = size * 2;
34         if (limit != -1 && newSize > limit) {
35             newSize = limit;
36         }
37         // 扩展策略是创建一个数组长度为原来两倍的新数组
38         Object[] newStack = new Object[newSize];
39         // 将老数组对象引用复制到新数组
40         System.arraycopy(stack, 0, newStack, 0, size);
41         // 将 stack 指向新数组，老数组可以被 GC 掉了
42         stack = newStack;
43         size = newSize;
```

```
44     }  
45 }
```


这个代码逻辑比较清晰，主要是 `SynchronizedStack` 内部维护了一个对象数组，并且用数组来实现栈的接口：`push` 和 `pop` 方法，这两个方法分别用来归还对象和获取对象。你可能好奇为什么 Tomcat 使用一个看起来比较简单的 `SynchronizedStack` 来做对象容器，为什么不使用高级一点的并发容器比如 `ConcurrentLinkedQueue` 呢？

这是因为 `SynchronizedStack` 用数组而不是链表来维护对象，可以减少结点维护的内存开销，并且它本身只支持扩容不支持缩容，也就是说数组对象在使用过程中不会被重新赋值，也就不会被 GC。这样设计的目的是用最低的内存和 GC 的代价来实现无界容器，同时 Tomcat 的最大同时请求数是有限制的，因此不需要担心对象的数量会无限膨胀。

Jetty 的 ByteBufferPool

我们再来看 Jetty 中的对象池 `ByteBufferPool`，它本质是一个 `ByteBuffer` 对象池。当 Jetty 在进行网络数据读写时，不需要每次都在 JVM 堆上分配一块新的 Buffer，只需在 `ByteBuffer` 对象池里拿到一块预先分配好的 Buffer，这样就避免了频繁的分配内存和释放内存。这种设计你同样可以在高性能通信中间件比如 Mina 和 Netty 中看到。

`ByteBufferPool` 是一个接口：

 复制代码


```
1 public interface ByteBufferPool  
2 {  
3     public ByteBuffer acquire(int size, boolean direct);  
4  
5     public void release(ByteBuffer buffer);  
6 }
```

接口中的两个方法：`acquire` 和 `release` 分别用来分配和释放内存，并且你可以通过 `acquire` 方法的 `direct` 参数来指定 buffer 是从 JVM 堆上分配还是从本地内存分配。`ArrayByteBufferPool` 是 `ByteBufferPool` 的实现类，我们先来看看它的成员变量和构造函数：

 复制代码

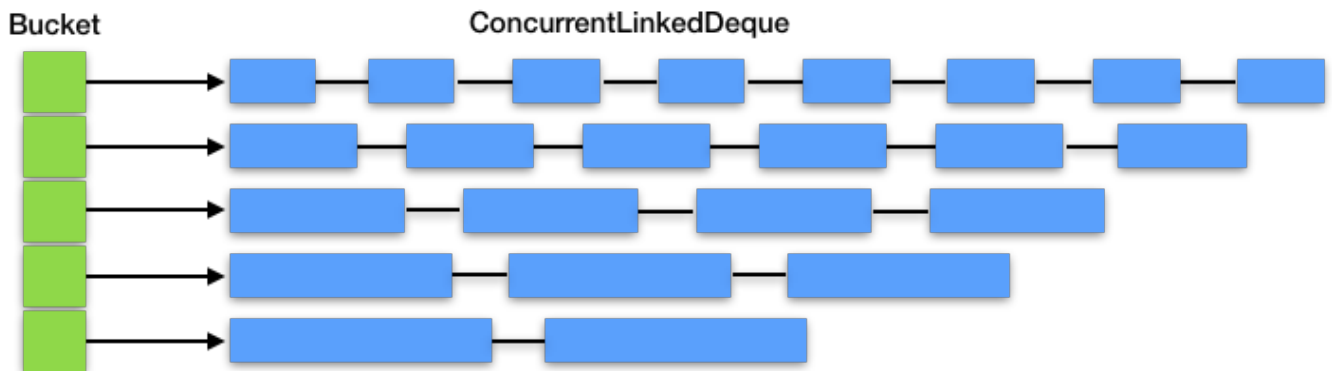
```
1 public class ArrayByteBufferPool implements ByteBufferPool
2 {
3     private final int _min;// 最小 size 的 Buffer 长度
4     private final int _maxQueue;//Queue 最大长度
5
6     // 用不同的 Bucket(桶) 来持有不同 size 的 ByteBuffer 对象, 同一个桶中的 ByteBuffer size
7     private final ByteBufferPool.Bucket[] _direct;
8     private final ByteBufferPool.Bucket[] _indirect;
9
10    //ByteBuffer 的 size 增量
11    private final int _inc;
12
13    public ArrayByteBufferPool(int minSize, int increment, int maxSize, int maxQueue)
14    {
15        // 检查参数值并设置默认值
16        if (minSize<=0)//ByteBuffer 的最小长度
17            minSize=0;
18        if (increment<=0)
19            increment=1024;// 默认以 1024 递增
20        if (maxSize<=0)
21            maxSize=64*1024;//ByteBuffer 的最大长度默认是 64K
22
23        //ByteBuffer 的最小长度必须小于增量
24        if (minSize>=increment)
25            throw new IllegalArgumentException("minSize >= increment");
26
27        // 最大长度必须是增量的整数倍
28        if ((maxSize%increment)!=0 || increment>=maxSize)
29            throw new IllegalArgumentException("increment must be a divisor of maxSize")
30
31        _min=minSize;
32        _inc=increment;
33
34        // 创建 maxSize/increment 个桶, 包含直接内存的与 heap 的
35        _direct=new ByteBufferPool.Bucket[maxSize/increment];
36        _indirect=new ByteBufferPool.Bucket[maxSize/increment];
37        _maxQueue=maxQueue;
38        int size=0;
39        for (int i=0;i<_direct.length;i++)
40        {
41            size+=_inc;
42            _direct[i]=new ByteBufferPool.Bucket(this,size,_maxQueue);
43            _indirect[i]=new ByteBufferPool.Bucket(this,size,_maxQueue);
44        }
45    }
46 }
```

从上面的代码我们看到，ByteBufferPool 是用不同的桶（Bucket）来管理不同长度的 ByteBuffer，因为我们可能需要分配一块 1024 字节的 Buffer，也可能需要一块 64K 字节的 Buffer。而桶的内部用一个 ConcurrentLinkedDeque 来放置 ByteBuffer 对象的引用。


 复制代码

```
1 private final Deque<ByteBuffer> _queue = new ConcurrentLinkedDeque<>();
```

你可以通过下面的图再来理解一下：



而 Buffer 的分配和释放过程，就是找到相应的桶，并对桶中的 Deque 做出队和入队的操作，而不是直接向 JVM 堆申请和释放内存。

 复制代码

```
1 // 分配 Buffer
2 public ByteBuffer acquire(int size, boolean direct)
3 {
4     // 找到对应的桶，没有的话创建一个桶
5     ByteBufferPool.Bucket bucket = bucketFor(size,direct);
6     if (bucket==null)
7         return newByteBuffer(size,direct);
8     // 这里其实调用了 Deque 的 poll 方法
9     return bucket.acquire(direct);
10
11 }
12
13 // 释放 Buffer
14 public void release(ByteBuffer buffer)
15 {
16     if (buffer!=null)
17     {
```

```
18      // 找到对应的桶
19      ByteBufferPool.Bucket bucket = bucketFor(buffer.capacity(),buffer.isDirect());
20
21      // 这里调用了 Deque 的 offerFirst 方法
22      if (bucket!=null)
23          bucket.release(buffer);
24      }
25 }
```

对象池的思考

对象池作为全局资源，高并发环境中多个线程可能同时需要获取对象池中的对象，因此多个线程在争抢对象时会因为锁竞争而阻塞，因此使用对象池有线程同步的开销，而不使用对象池则有创建和销毁对象的开销。对于对象池本身的设计来说，需要尽量做到无锁化，比如 Jetty 就使用了 ConcurrentLinkedDeque。如果你的内存足够大，可以考虑用**线程本地 (ThreadLocal) 对象池**，这样每个线程都有自己的对象池，线程之间互不干扰。

为了防止对象池的无限膨胀，必须要对池的大小做限制。对象池太小发挥不了作用，对象池太大的话可能有空闲对象，这些空闲对象会一直占用内存，造成内存浪费。这里你需要根据实际情况做一个平衡，因此对象池本身除了应该有自动扩容的功能，还需要考虑自动缩容。

所有的池化技术，包括缓存，都会面临内存泄露的问题，原因是对象池或者缓存的本质是一个 Java 集合类，比如 List 和 Stack，这个集合类持有缓存对象的引用，只要集合类不被 GC，缓存对象也不会被 GC。维持大量的对象也比较占用内存空间，所以必要时我们需要主动清理这些对象。以 Java 的线程池 ThreadPoolExecutor 为例，它提供了 allowCoreThreadTimeOut 和 setKeepAliveTime 两种方法，可以在超时报后销毁线程，我们在实际项目中也可以参考这个策略。

另外在使用对象池时，我这里还有一些小贴士供你参考：

对象在用完后，需要调用对象池的方法将对象归还给对象池。

对象池中的对象在再次使用时需要重置，否则会产生脏对象，脏对象可能持有上次使用的引用，导致内存泄漏等问题，并且如果脏对象下一次使用时没有被清理，程序在运行过程中会发生意想不到的问题。

对象一旦归还给对象池，使用者就不能对它做任何操作了。

向对象池请求对象时有可能出现的阻塞、异常或者返回 null 值，这些都需要我们做一些额外的处理，来确保程序的正常运行。

本期精华

Tomcat 和 Jetty 都用到了对象池技术，这是因为处理一次 HTTP 请求的时间比较短，但是这个过程中又需要创建大量复杂对象。

对象池技术可以减少频繁创建和销毁对象带来的成本，实现对象的缓存和复用。如果你的系统需要频繁的创建和销毁对象，并且对象的创建代价比较大，这种情况下，一般来说你会观察到 GC 的压力比较大，占用 CPU 率比较高，这个时候你就可以考虑使用对象池了。

还有一种情况是你需要对资源的使用做限制，比如数据库连接，不能无限制地创建数据库连接，因此就有了数据库连接池，你也可以考虑把一些关键的资源池化，对它们进行统一管理，防止滥用。

课后思考

请你想想在实际工作中，有哪些场景可以用“池化”技术来优化。

不知道今天的内容你消化得如何？如果还有疑问，请大胆的在留言区提问，也欢迎你把你的课后思考和心得记录下来，与我和其他同学一起讨论。如果你觉得今天有所收获，欢迎你把它分享给你的朋友。

深入拆解 Tomcat & Jetty

从源码角度深度探索 Java 中间件

李号双

eBay 技术主管



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 19 | 比较：Jetty的线程策略EatWhatYouKill

下一篇 21 | 总结：Tomcat和Jetty的高性能、高并发之道

精选留言 (8)

写留言



z.l

2019-06-27

赞，如果能有jetty和netty对象池实现的对比就更好了



802.11

2019-06-27

工厂模式和池化思想有什么区别呢

作者回复: 工厂模式没有一个池来存对象，并且侧重不同，工厂模式是设计上的考虑，不是性能方面的



kyon

2019-06-26

您好，请问 ArrayByteBufferPool 中，direct 和 indirect 都是 new 出来的，区别是什么？另外在 new ByteBufferPool.Bucket(this,size,_maxQueue) 中，参数 _maxQueue 的作用是什么？

作者回复: APR那篇有详细解释，HeapByteBuffer与DirectByteBuffer的区别。

maxqueue 的作用是控制内存池的总大小



TJ

2019-06-25

为什么tomcat不使用java本身的stack class? 它也是基于数组的。自己再加一个同步就可以了

作者回复: java本身的stack是不是实现上有点复杂，这里要尽量简单



WL

2019-06-25

请问老师Tomcat为什么用栈做对象池，那要去栈底的对象不是很麻烦很不灵活吗？为啥不用map的方式呢？

作者回复: 不会要去栈底找对象的情况，对象都是无差别的



-W.LI-

2019-06-25

我之前听人说事务里面的数据库链接就是通过threadLocal来共享的(事务结束后会从threadlocal删除当前链接么?)。那这个数据库的连接如果和Tomcat的线程数一对一绑定上能提高效率么?



-W.LI-

2019-06-25

老师好学到了。通过threadlocal来减少锁竞争上下文切换的开销。

可是我看见好多帖子说threadlocal容易内存泄露啥的肯比较多需要慎用。五年码龄从没用过😁。

请教一个问题threadLocal中的对象如果用完不清。下次别请求Tomcat线程池中拿到同个线程，能取到之前请求存入的数据么？

作者回复: 会的，所以要及时清理



1



Liam

2019-06-25

tomcat和jetty的对象池没有空闲超时/超量回收的机制吗？

作者回复: 似乎没有，对象池大小靠连接数限制

