网络IO框架 - Netty

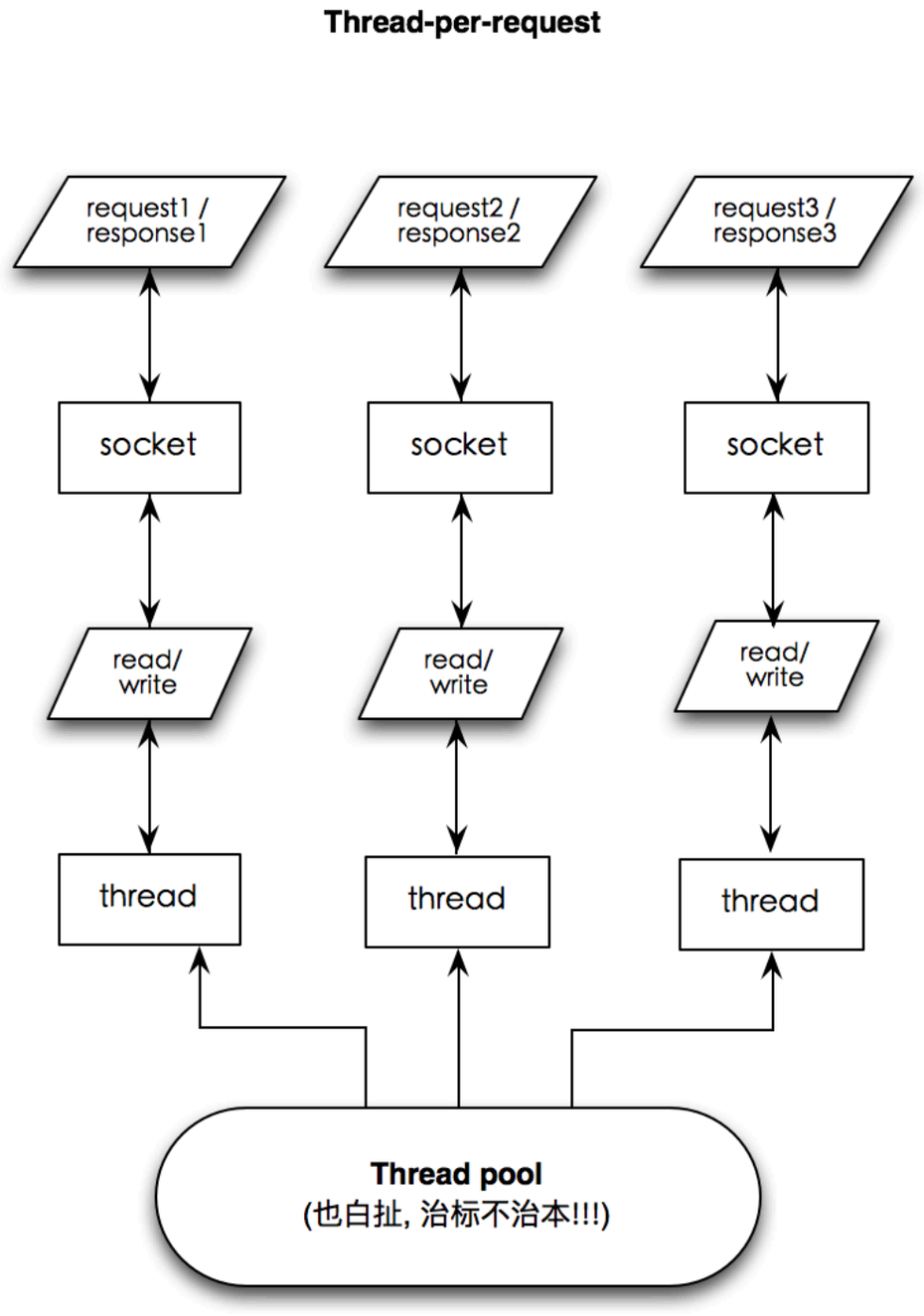
by 家纯

Why Netty?

- Blocking IO的瓶颈
- Non-blocking IO
- Java原生nio api从入门到放弃
- Nio代码实现方面的一些缺点
- Netty Native Transport
- epoll介绍
- Netty Thread Model
- ChannelPipeline + 异步事件驱动
- Pooling & reuse
- 最佳实践
- 从Netty源码中学到的代码技巧
- 如何利用Netty写一个高性能RPC框架

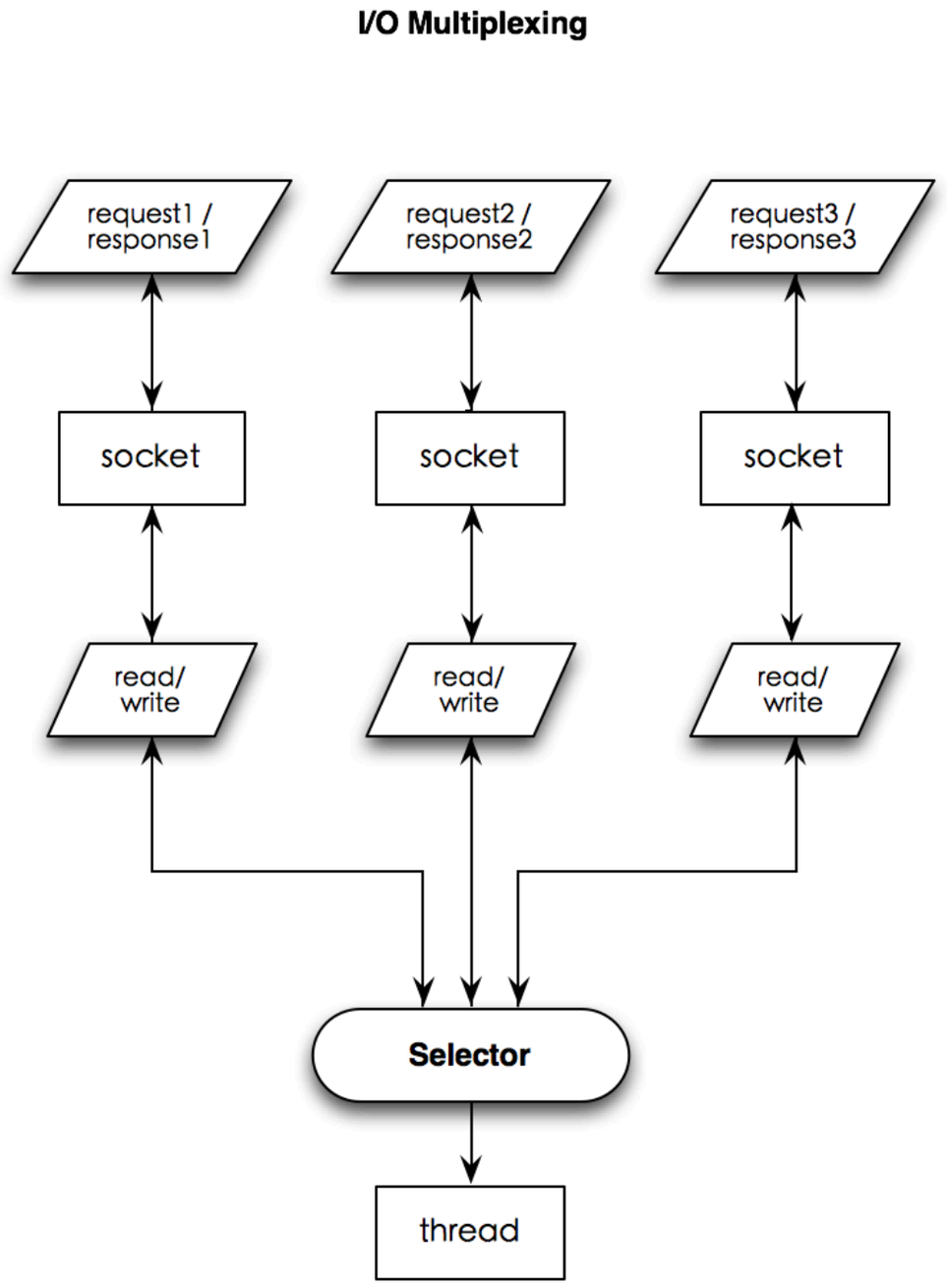
Blocking IO 的瓶颈

- 性能:
  - Thread-per-request导致服务端在并发接入和吞吐量方面受到极大的限制, 高并发场景会带来较多的context-switch开销;
  - 应用内存吃紧(Java中每个线程默认线程栈大小1M);
- 可靠性:
  - 很难做业务级别的线程池隔离,个别业务处理慢会直接挂住IO线程, 最终阻塞整个系统;



Non-blocking IO

- IO多路复用:
  - 虽然IO多路复用也是阻塞的, 但只是阻塞在select/poll/epoll这样的system call上, 并不需要阻塞recvfrom这样的IO操作;
- IO处理和业务处理可以隔离:
  - IO处理通常为CPU密集型任务
  - 业务处理通常为IO密集型任务



Java原生nio api从入门到放弃

- 复杂度高:
  - api复杂难懂, 入门困难;
  - 粘包/半包问题比较难处理;
  - 需超强的并发编程功底, 否则很难写出高效稳定的实现;
- 稳定性差, 坑多且深:
  - 调试困难, 偶尔遭遇匪夷所思极难重现的bug, 边哭边查是常有的事儿;
  - linux下EPollArrayWrapper.epollWait直接返回导致空轮训进而导致100% cpu的bug一直也没解决利索, netty帮你work around(通过rebuilding selector);

Nio代码实现方面的一些缺点

- Selector.selectedKeys() 产生太多垃圾:
  - netty修改了sun.nio.ch.SelectorImpl的实现, 使用双数组代替HashSet存储来selectedKeys:
    - - 相比*HashSet(*迭代器, 包装对象等*)*少的多的垃圾制造*(help GC);*
    - - 轻微的性能收益*(1~2%);*
- nio的代码到处是synchronized (比如allocate direct buffer):
  - 对于allocate direct buffer, netty的pooledBytebuf有前置TLAB(Thread-local allocation buffer);
  - netty native transport中锁少了很多;

Nio代码实现方面的一些缺点

- fdToKey映射:
  - EPollSelectorImpl#fdToKey维持着所有连接的fd(描述符)对应SelectionKey的映射, 是个HashMap;
  - 每个worker线程有一个selector, 也就是每个worker有一个fdToKey, 这些fdToKey大致均分了所有连接;
  - 想象一下单机hold几十万的连接的场景, HashMap从默认size=16, 一步一步rehash...
- Selector在linux平台是epoll LT实现:
  - netty native transport支持epoll ET;

Nio代码实现方面的一些缺点

- Direct Buffers事实上还是由GC管理:
  - DirectByteBuffer.cleaner这个虚引用负责free direct memory, DirectByteBuffer只是个壳子, 这个壳子如果坚强的活下去熬过新生代的年龄限制最终晋升到老年代将是一件让人伤心的事情…
  - 无法申请到足够的direct memory会显式触发GC, Bits.reserveMemory() -> { System.gc() }, 首先因为GC中断整个进程不说, 代码中还sleep 100毫秒, 醒了要是发现还不行就OOM;
  - 更糟的是如果你听信了个别<XX优化指南/大法>的谗言设置了-XX:+DisableExplicitGC参数, 悲剧终于还是发生了...
  - netty的UnpooledUnsafeNoCleanerDirectByteBuf去掉了cleaner, 由netty框架维护引用计数来实时的去释放;

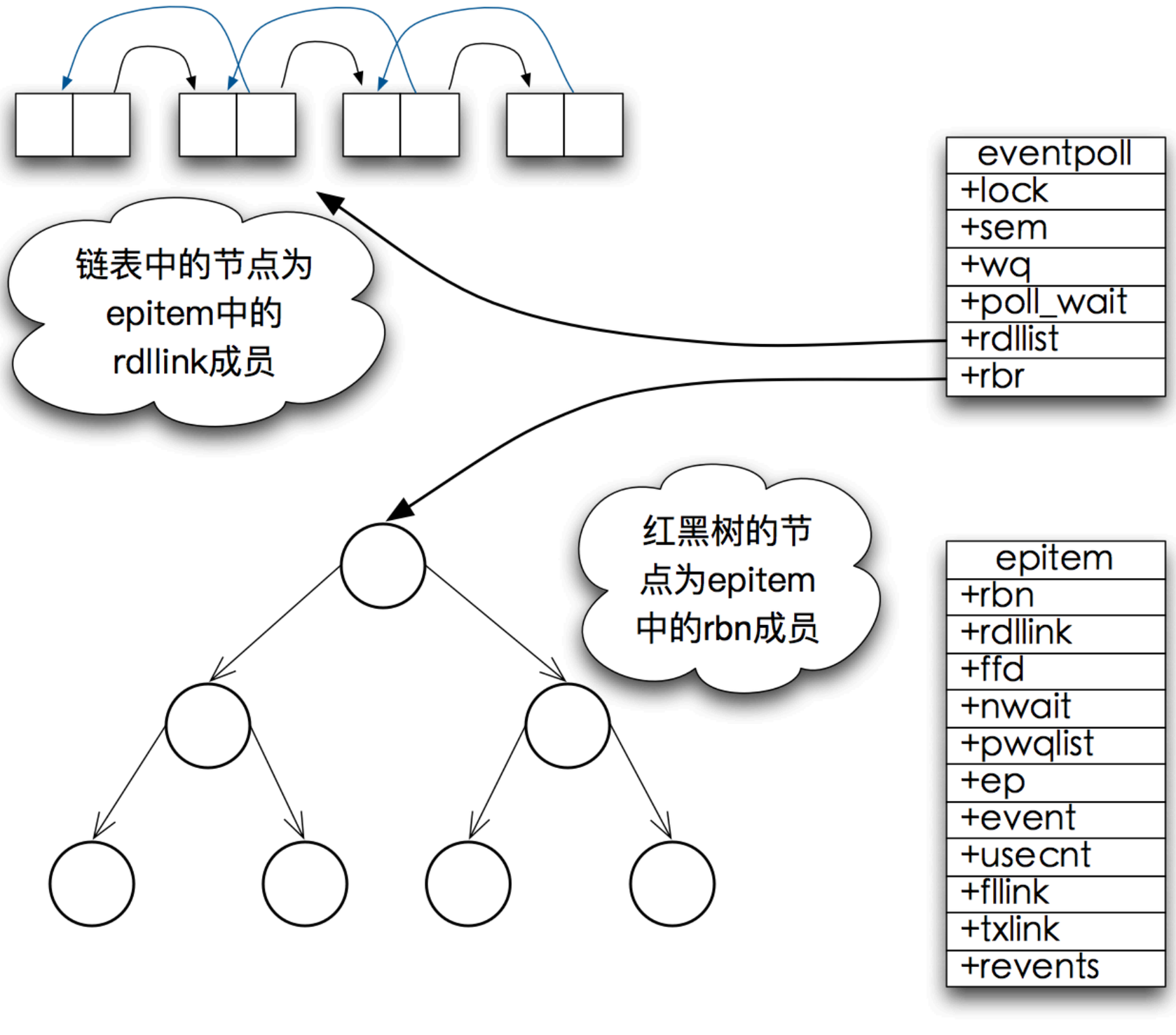
Netty Native Transport

- 相比nio创建更少的对象, 更小的GC压力;
- 针对linux平台优化, 一些specific features:
  - SO_REUSEPORT - 端口复用(允许多个socket监听同一个IP+端口, 与RPS/RFS协作, 可进一步提升性能);
  - TCP_FASTOPEN - 3次握手时也用来交换数据;
  - 等等…
- 支持 epoll ET(关键哥);

多路复用器: select/poll/epoll之间的区别

- select/poll:
  - 本身的实现机制上的限制(采用轮询方式检测就绪事件, 时间复杂度: O(n), 每次还要将肥大的fd_set在用户空间和内核空间拷贝来拷贝去), 并发连接越大, 性能越差;
  - poll相比select没有很大差异, 只是取消了最大文件描述符个数的限制;
  - select/poll都是LT模式;
- epoll:
  - 采用回调方式检测就绪事件, 时间复杂度: O(1), 每次epoll_wait调用只返回已就绪的文件描述符;
  - epoll支持LT和ET模式;

epoll涉及的数据结构



epoll三个方法简介

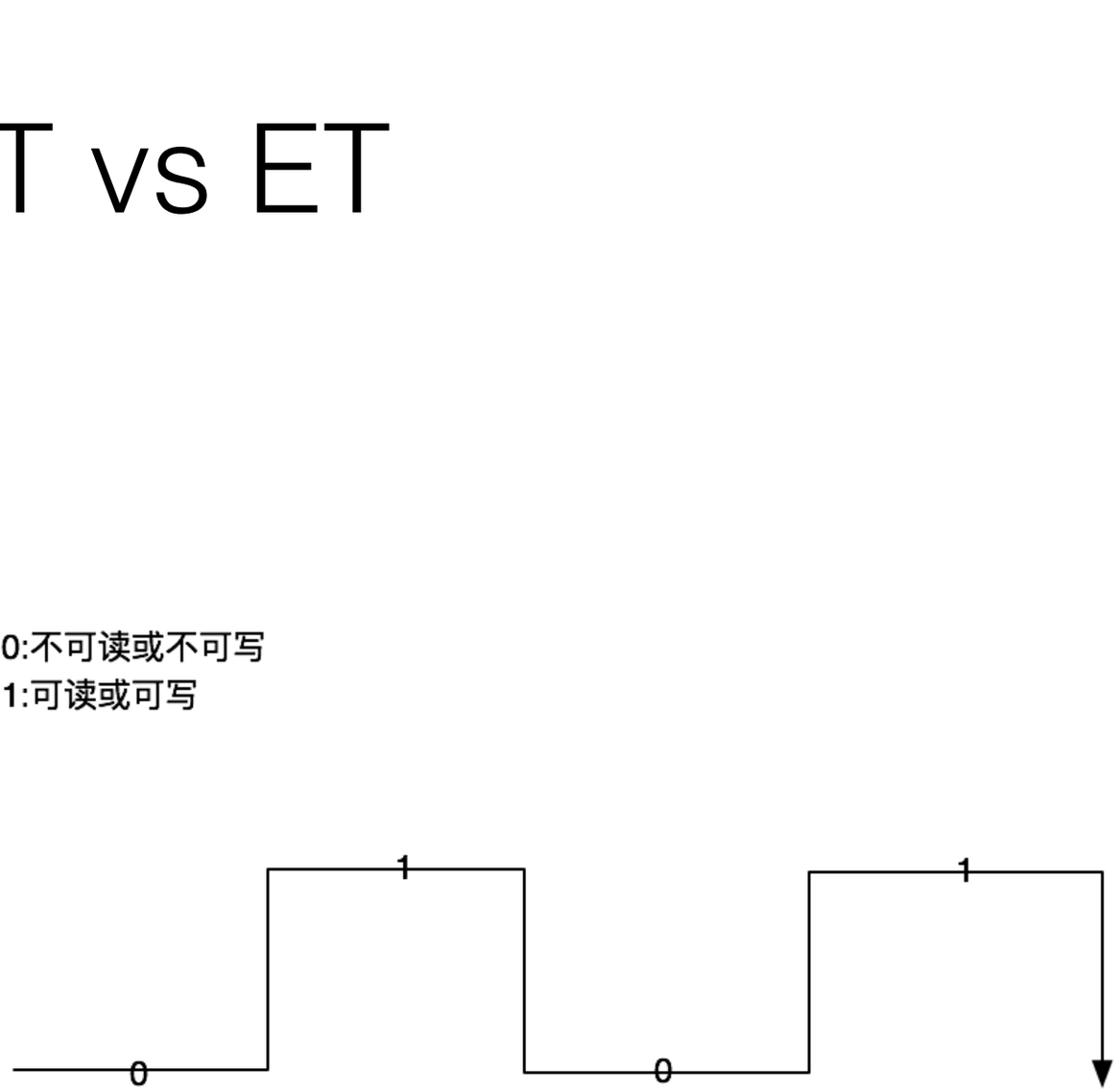
- 主要代码: linux-2.6.11.12/fs/eventpoll.c
- **int epoll_create(int size)**
  - 创建rb-tree(红黑树)和ready-list(就绪链表);
    - - 红黑树O(logN), 平衡效率和内存占用, 在容量需求不能确定并可能量很大的情况下红黑树是最佳选择;
      - - size参数已经没什么意义, 早期epoll实现是hash表, 所以需要size参数;
- **int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)**
  - 把epitem放入rb-tree并向内核中断处理程序注册ep_poll_callback, callback触发时把该epitem放进ready-list;
- **int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout)**
  - ready-list —> events[];

epoll_wait工作流程概述

- epoll_wait调用ep_poll:
  - 当rdlist(ready-list)为空(无就绪fd)时挂起当前线程, 直到rdlist不为空时线程才被唤醒;
- 文件描述符fd的events状态改变:
  - buffer由不可读变为可读或由不可写变为可写, 导致相应fd上的回调函数ep_poll_callback被触发;
- ep_poll_callback被触发:
  - 将相应fd对应epitem加入rdlist, 导致rdlist不空, 线程被唤醒, epoll_wait得以继续执行;
- 执行ep_events_transfer函数:
  - 将rdlist中的epitem拷贝到txlist中, 并将rdlist清空;
  - 如果是epoll LT, 并且fd.events状态没有改变(比如buffer中数据没读完并不会改变状态), 会再重新将epitem放回rdlist;
- 执行ep_send_events函数(关键哥来了):
  - 扫描txlist中的每个epitem, 调用其关联fd对应的poll方法取得较新的events;
  - 将取得的events和相应的fd发送到用户空间;

epoll LT vs ET

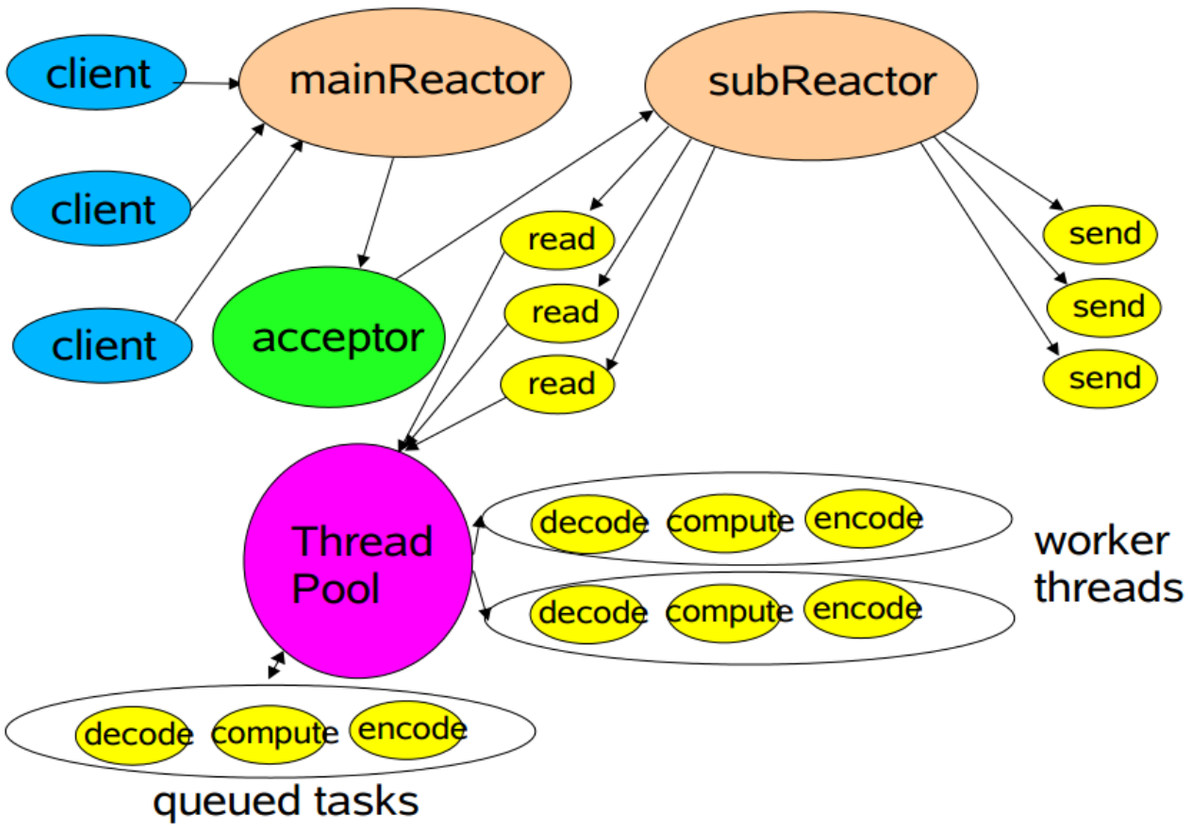
- 概念:
  - LT: level-triggered 水平触发;
  - ET: edge-triggered 边沿触发;
- 可读:
  - buffer不为空的时候fd的events中对应的可读状态就被置为1, 否则为0;
- 可写:
  - buffer中有空间可写的时候fd的events中对应的可写状态就被置为1, 否则为0;

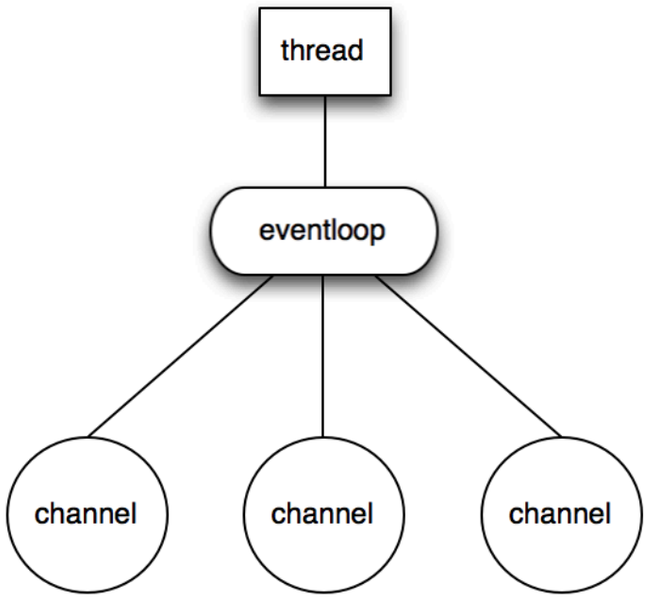


LT: 在1处, 内核会不断通知进程该描述符已就绪
ET: 只有在0->1处, 内核才会通知一次进程该描述符已就绪

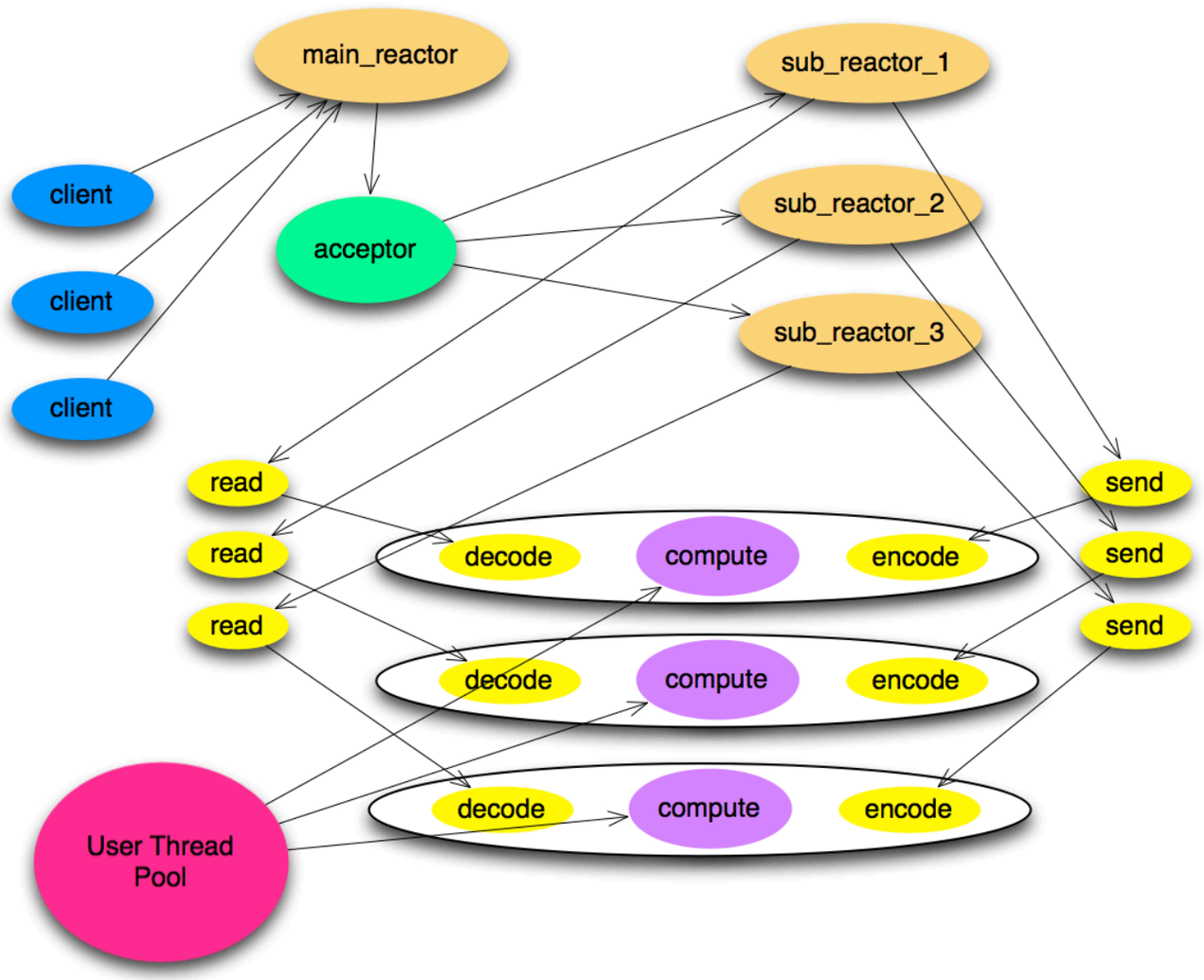
Netty Thread Model

multi-reactor





netty-reactor



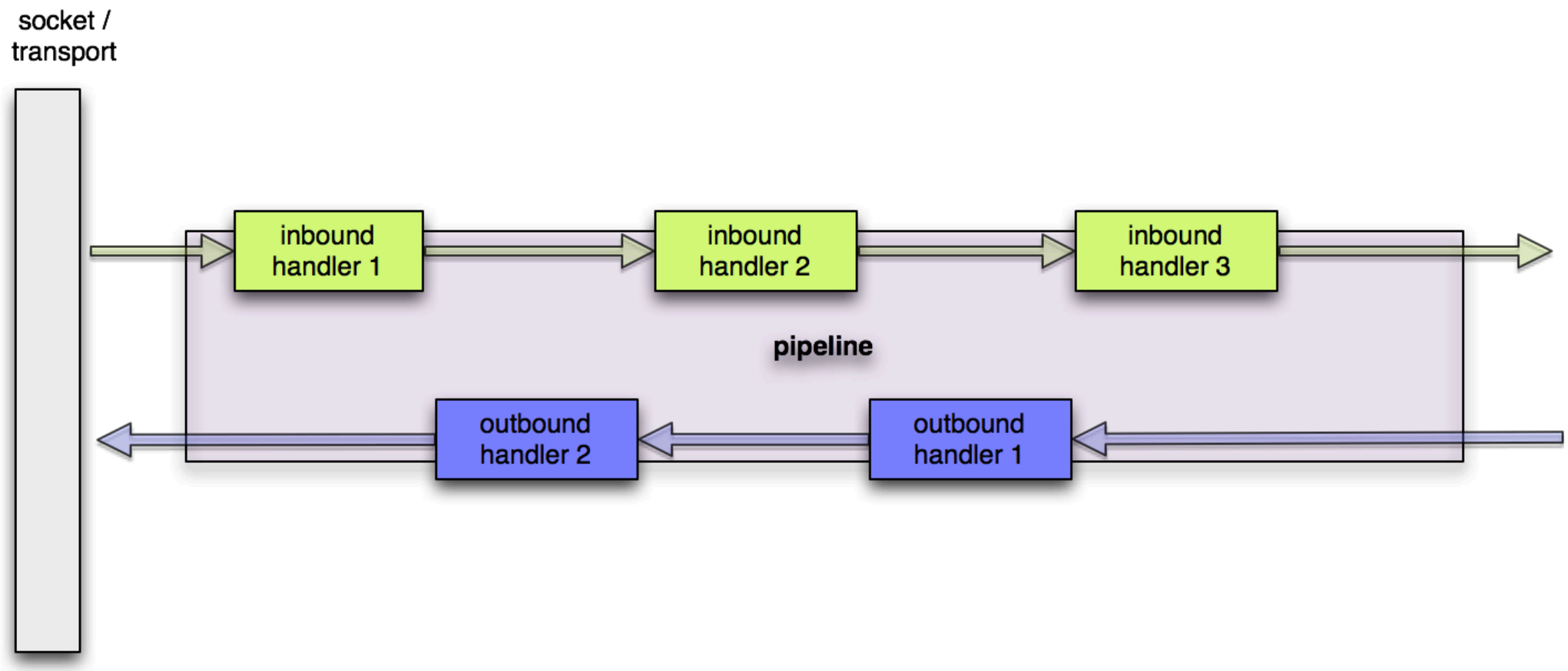
Netty中几个重要概念

- EventLoopGroup
- EventLoop
- Boss/Worker
- Channel
- Pipeline
- ChannelHandler

Netty中几个重要概念及其关系

- EventLoopGroup中包含一组EventLoop;
- EventLoop的大致数据结构是:
  - 一个任务队列(mpsc_queue: 多生产者单消费者 lock-free);
  - 一个延迟任务队列(delay_queue: 一个二叉堆结构的优先级队列, 复杂度为O(log n));
  - EventLoop绑定了一个Thread, 这直接避免了pipeline中的线程竞争;
  - 每个EventLoop有一个Selector, Boss用Selector处理accept, Worker用Selector处理read, write等;
- Boss可理解为Reactor模式中的mainReactor角色, Worker可理解为subReactor角色;
  - Boss和Worker共用EventLoop的代码逻辑;
  - 在不bind多端口的情况下BossEventLoopGroup中只需要包含一个EventLoop, 也只能用上一个, 多了没用;
  - WorkerEventLoopGroup中一般包含多个EventLoop, 经验值一般为 cpu cores * 2(根据场景测试找出最佳值才是王道);
  - Netty server启动后会把一个监听套接字ServerSocketChannel注册到Boss中;
  - Boss主要责任就是accept连接(channel), 然后轮询打赏给Worker;
  - Worker接到Boss打赏的channel后负责处理此channel后续的read/write等IO事件;
- Channel分两大类ServerChannel和Channel, ServerChannel对应着监听套接字(ServerSocketChannel), Channel对应着一个网络连接;
- Pipeline是责任链模式的设计, 里面有多个handler的, 上行事件顺序执行pipeline, 下行事件逆序执行pipeline;

ChannelPipeline + 异步事件驱动



Pooling & reuse

- PooledByteBufAllocator
  - 基于 jemalloc paper (3.x);
  - ThreadLocal caches for lock free;
    - 这个做法导致曾经有坑: 申请(Bytebuf)线程与归还(Bytebuf)线程不是同一个导致内存泄漏, 后来用一个mpsc_queue解决, 代价就是牺牲了一点点性能;
  - Different size classes;
- Recycler
  - ThreadLocal + Stack
    - 曾经有坑, 申请(元素)线程与归还(元素)线程不是同一个导致内存泄漏;
    - 后来改进为不同线程归还元素的时候放入一个WeakOrderQueue中并关联到stack上, 下次pop时如果stack为空则先扫描所有关联到当前stack上的weakOrderQueue;
    - WeakOrderQueue是多个数组的链表, 每个数组默认size=16;
  - 问题: 老年代对象引用新生代对象对GC的影响;

最佳实践

- 业务线程池必要性:
  - 业务逻辑尤其是阻塞时间较长的逻辑, 不要占用netty的IO线程, dispatch到业务线程池中去;
- WriteBufferWaterMark, 注意默认的高低水位线设置(32K~64K), 根据场景适当调整;
- 重写MessageSizeEstimator来反应真实的高低水位线:
  - 默认实现不能计算对象size, 由于write时还没路过任何一个outboundHandler就已经开始计算message size, 此时对象还没有被encode成Bytebuf, 所以size计算肯定是不准确的(偏低);
- 注意EventLoop#ioRatio的设置(默认50), 这是EventLoop执行IO任务和非IO任务的一个时间比例上的控制;
- 对象的序列化/反序列化操作建议在IO线程之外处理, 一个是这类操作占用IO线程的cpu时间, 再一个是可能有略严重的锁竞争;
- 空闲链路检测用谁调度?
  - Netty4.x默认使用IO线程调度, 使用eventLoop的delayQueue, 一个二叉堆实现的优先级队列, 复杂度为O(log N), 每个worker处理自己的链路监测, 有助于减少上下文切换, 但是网络IO操作与idle会相互影响;
  - 如果总的连接数小, 比如几万以内, 上面的实现并没什么问题, 连接数大建议用HashedWheelTimer实现一个IdleStateHandler, HashedWheelTimer复杂度为 O(1), 同时可以让网络IO操作和idle互不影响, 但有上下文切换开销;

最佳实践

- 使用ctx.writeAndFlush还是channel.writeAndFlush?
  - ctx.write直接走到下一个outbound handler, 注意别让它违背你的初衷绕过了空闲链路检测;
  - channel.write从末尾开始倒着向前挨个路过pipeline中的所有outbound handlers;
- 使用Bytebuf.forEachByte() 来代替循环 ByteBuf.readByte()的遍历操作, 避免rangeCheck();
- 使用CompositeByteBuf来避免不必要的内存拷贝:
  - 缺点是索引计算时间复杂度高, 请根据自己场景衡量;
- 如果要读一个int, 用Bytebuf.readInt(), 不要Bytebuf.readBytes(buf, 0, 4):
  - 这能避免一次memory copy (long, short等同理);
- 配置UnpooledUnsafeNoCleanerDirectByteBuf来代替jdk的DirectByteBuf, 让netty框架基于引用计数来释放堆外内存:
  - io.netty.maxDirectMemory
    - < 0: 不使用cleaner, netty方面直接继承jdk设置的最大direct memory size, (jdk的direct memory size是独立的, 这将导致总的direct memory size将是jdk配置的2倍);
    - == 0: 使用cleaner, netty方面不设置最大direct memory size;
    - > 0: 不使用cleaner, 并且这个参数将直接限制netty的最大direct memory size, (jdk的direct memory size是独立的, 不受此参数限制);

最佳实践

- 最佳连接数:
  - 一条连接有瓶颈, 无法有效利用cpu;
  - 连接太多也白扯, 最佳实践是根据自己场景测试;
- 使用PooledBytebuf时要善于利用 -Dio.netty.leakDetection.level 参数:
  - 四种级别: DISABLED(禁用), SIMPLE(简单), ADVANCED(高级), PARANOID(偏执);
  - SIMPLE, ADVANCED采样率相同, 不到1%(按位与操作 mask ==128 - 1);
  - 默认是SIMPLE级别, 开销不大;
  - 出现泄漏时日志会出现”LEAK: ”字样, 请时不时grep下日志, 一旦出现”LEAK: ”立刻改为ADVANCED级别再跑, 可以报告泄漏对象在哪被访问的;
  - PARANOID: 测试的时候建议使用这个级别, 100%采样;
- Channel.attr(), 将自己的对象attach到channel上;
  - 拉链法实现的线程安全的hash表, 也是分段锁(锁链表头), 只有hash冲突的情况下才有锁竞争;
  - 默认hash表只有4个桶哦, 使用时不要太任性;

从Netty源码中学到的代码技巧

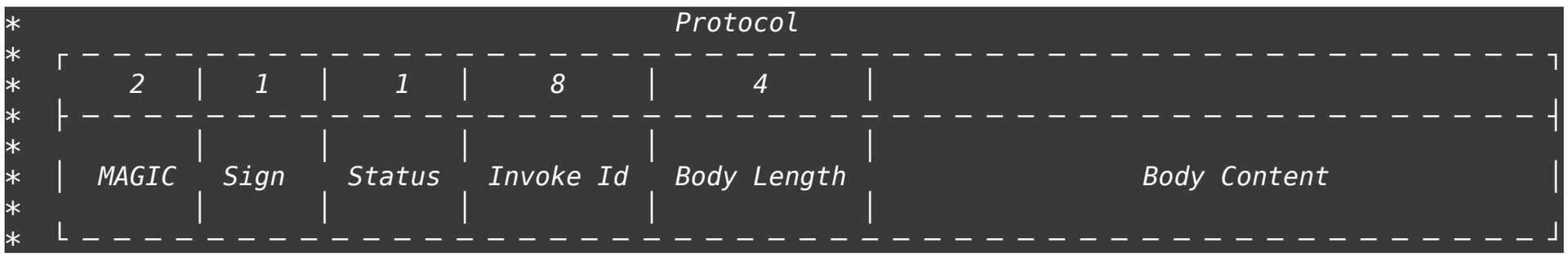
- 海量对象场景中 AtomicIntegerFieldUpdater --> AtomicInteger:
  - Java中对象头12 bytes(开启压缩指针的情况下), 又因为Java对象按照8字节对齐, 所以对象最小16 bytes, AtomicInteger大小为16 bytes, AtomicLong大小为 24 bytes;
  - AtomicIntegerFieldUpdater作为static field去操作volatile int;
- FastThreadLocal, 相比jdk的实现更快:
  - index原子自增的数组存储 —> 拉链法的Hash表;
- IntObjectHashMap / LongObjectHashMap …
  - 这个流行不是一天两天了, 并不是netty先这样玩的, 首先int代替Integer作为key直接省了12个字节的内存, 哈希冲突的解决方式是线性扩展而非HashMap里面的拉链法;
- RecyclableArrayList, 基于前面说的Recycler, 频繁new ArrayList的场景可考虑;
- JCTools: 一些jdk没有的 SPSC/MPSC/SPMC/MPMC 并发队列;

如何利用Netty写一个高性能RPC框架

- 协议格式
- Proxy
- 网络层可扩展
- 泛化调用
- 压榨性能(Don’t trust it, Test it)
- Example

协议格式

- 协议头:

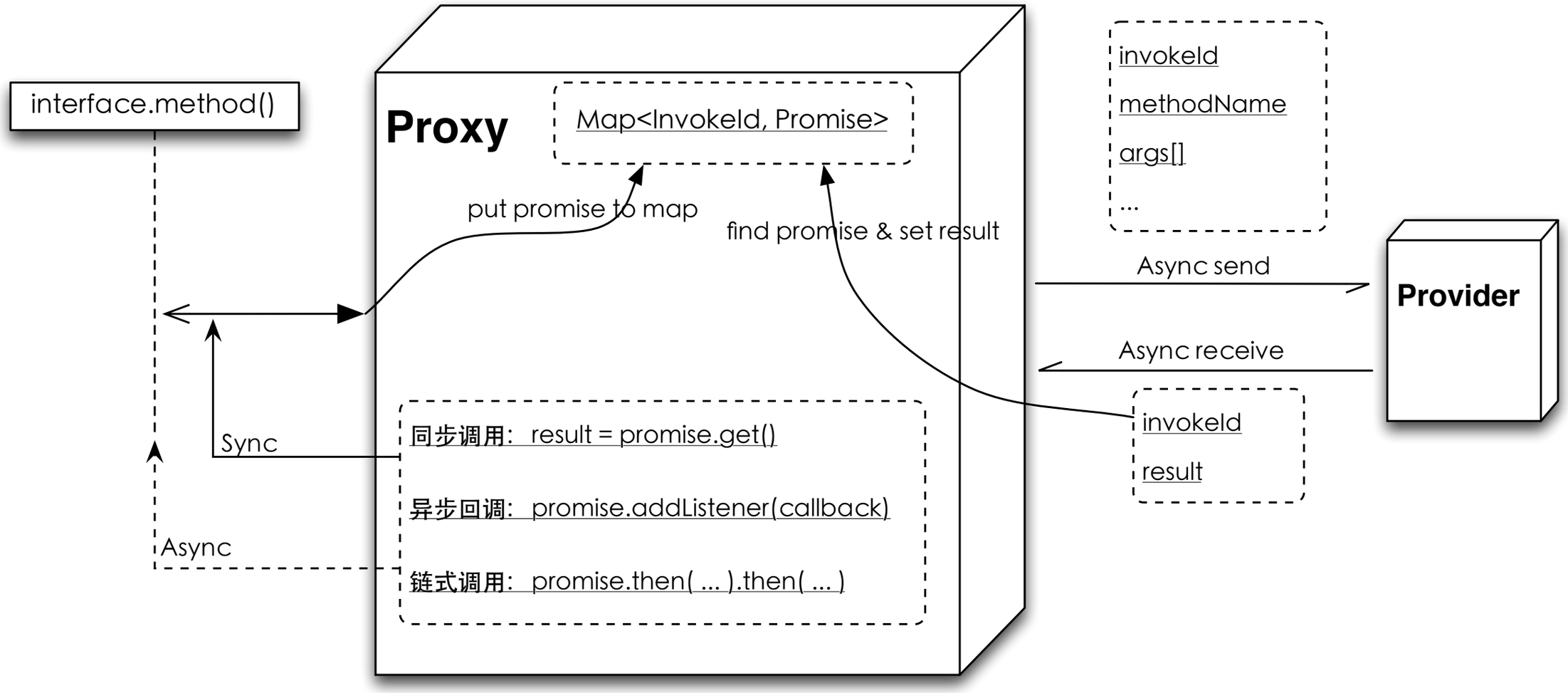



```
* 消息头16个字节定长
* = 2 // MAGIC = (short) 0xbabe
* + 1 // 消息标志位, 低地址4位用来表示消息类型Request/Response/Heartbeat等, 高地址4位用来表示序列化类型
* + 1 // 状态位, 设置请求响应状态
* + 8 // 消息 id, long 类型
* + 4 // 消息体 body 长度, int类型
```

```
(byte) (sign & 0x0f)                    // 低地址4位
(byte) ((((int) sign) & 0xff) >> 4)     // 高地址4位
```

- 协议体:
  - metadata: group, version, providerName;
  - methodName
  - ~~parameterTypes[]~~ 真的需要?
    - 问题: 1. 反序列化时触发ClassLoader.loadClass()潜在的锁竞争; 2. 协议体码流大小; 3: 泛化调用多了这个恼人的参数类型;
    - 方法重载问题解决: 根据参数类型找到最匹配的方法, 规则参考JLS <Java语言规范> $15.12.2.5 Choosing the Most Specific Method 章节; 或者直接使用现成的实现commons-lang#MethodUtils.getMatchingAccessibleMethod();
  - args[]
  - 其他: traceId, appName…

Proxy



* 若是netty4.x的线程模型: IO Thread(worker) —> Map<InvokeId, Promise> 代替全局Map能更好的避免线程竞争

网络层可扩展&泛化调用

- SPI:
  - -java.util.ServiceLoader
  - -META-INF/services/com.xxx.Xxx
- 剥离对netty#channel的依赖:
  - Map.put(nettyChannel, myChannel)?
  - nettyChannel.attr(name).setIfAbsent(myChannel)或许更有利于减少锁竞争;
- 泛化: Object $invoke(String methodName, Object... args);
  - 不需要Class[] parameterTypes;

压榨性能(Don’t trust it, Test it)

- 客户端Proxy对象(jdkProxy&javassist&cglib&asm&bytebuddy);

```
new ByteBuddy()
        .subclass(interfaceType)
        .method(isDeclaredBy(interfaceType))
        .intercept(to(handler, "handler").filter(not(isDeclaredBy(Object.class))))
        .make()
        .load(interfaceType.getClassLoader(), INJECTION)
        .getLoaded().newInstance()
```

```
1. filter(not(isDeclaredBy(Object.class)))), 避免toString/equals等方法被代理进行远程调用;
2. 轻微的性能优势(不要相信我, JMH test之 http://openjdk.java.net/projects/code-tools/jmh/);
```

- 服务端不可避免的反射用cglib#FastClass代替(记得FastClass要缓存);
- TCP_NODELAY设置为true;
- 避免反序列化占用IO线程:
  - decoder中根据header拿到bytes[]的body就好, 反序列化交给业务线程;
- 选择高效的序列化/反序列化框架(kryo/protobuf/protostuff/hessian/fastjson/…);
  - https://github.com/eishay/jvm-serializers/wiki
- IO线程绑定cpu? netty没做这个, 提前意淫下;
  - linux平台参考: https://github.com/OpenHFT/Java-Thread-Affinity
- 压测时你会发现并发量大时瓶颈在客户端, 客户端改为协程? (kilim&quasar)
- Netty Native Transport & PooledByteBufAllocator (减小GC带来的波动);
- 减少IO线程占用时间, 尽量减少线程上下文切换;

Example

- 玩票性质: https://github.com/fengjiachun/Jupiter
- 心中标杆: https://github.com/alibaba/dubbo

参考资料:

- Netty:
    - https://github.com/netty/netty
    - https://www.infoq.com/presentations/apple-netty
- JDK-source:
    - /jdk/src/solaris/classes/sun/nio/ch/EPollSelectorImpl.java
    - /jdk/src/solaris/classes/sun/nio/ch/EPollArrayWrapper.java
    - /jdk/src/solaris/native/sun/nio/ch/EPollArrayWrapper.c
- Linux-source:
    - linux-2.6.11.12/fs/eventpoll.c
    - https://code.csdn.net/chenyu105/linux_kernel_2-6-11-12_comment/tree/master
    - https://github.com/torvalds/linux
- RPS/RFS:
    - https://my.oschina.net/guol/blog/113144
- 请翻页

参考资料:

- I/O Multiplexing:
    - <UNIX网络编程>卷1 第六章
    - http://www.python4science.eu/multiplex_io.html
    - http://blog.csdn.net/russell_tao/article/details/17119729
- jemalloc:
    - http://jemalloc.net/
    - https://www.bsdcan.org/2006/papers/jemalloc.pdf
- SO_REUSEPORT:
    - https://my.oschina.net/miffa/blog/390931
- TCP_FASTOPEN:
    - https://www.oschina.net/question/12_137950
- 最佳实践主要参考来源:
    - http://calvin1978.blogcn.com/articles/netty-info.html