

NIO trick and trap

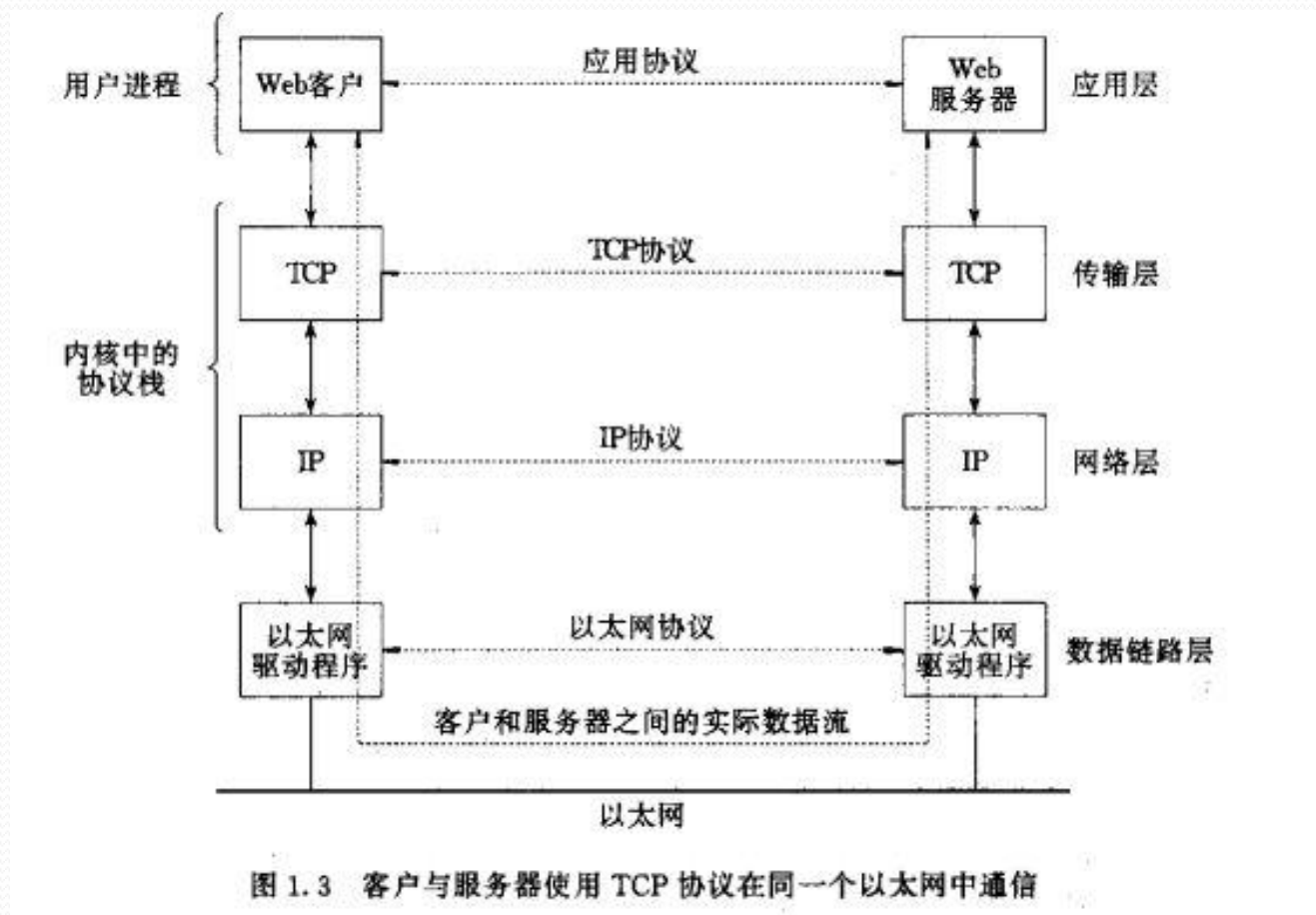
编写高性能Java NIO网络框架

伯岩 boyan@taobao.com

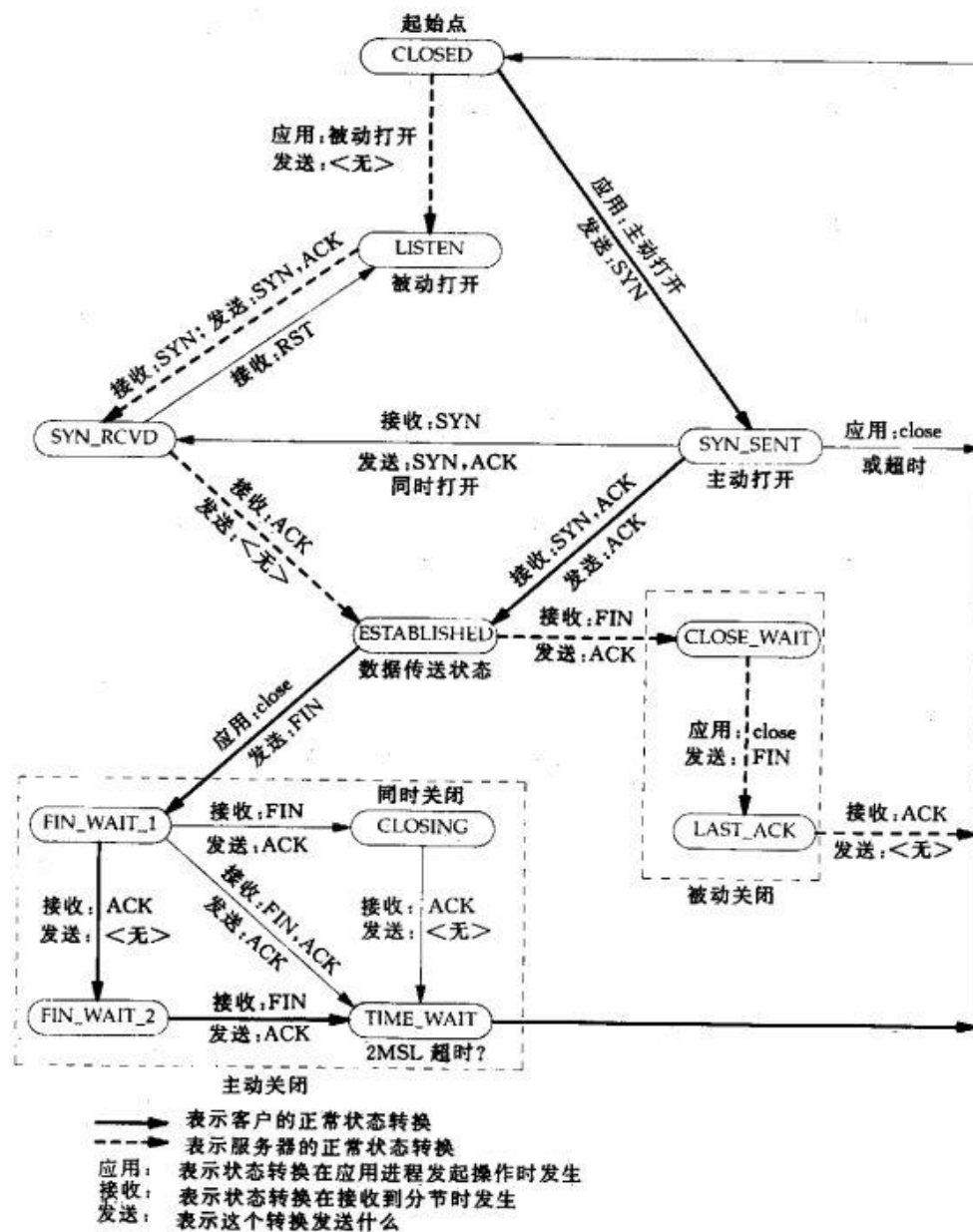
Agenda

- 基础
- nio概览
- Nio trick
- Nio trap

IP协议簇



TCP状态图



IO模型

- IO请求划分两个阶段：
 - 等待数据就绪
 - 从内核缓冲区拷贝到进程缓冲区
 - 按照请求是否阻塞
 - 同步IO
 - 异步IO
 - Unix的5种IO模型
 - 阻塞IO
 - 非阻塞IO
 - IO复用
 - 信号驱动IO
 - 异步IO
-
- ```
graph LR; A[阻塞IO] --> B[同步IO]; C[非阻塞IO] --> B; D[IO复用] --> B; E[信号驱动IO] --> F[异步IO]; G[异步IO] --> F;
```

# I/O模型之间的区别

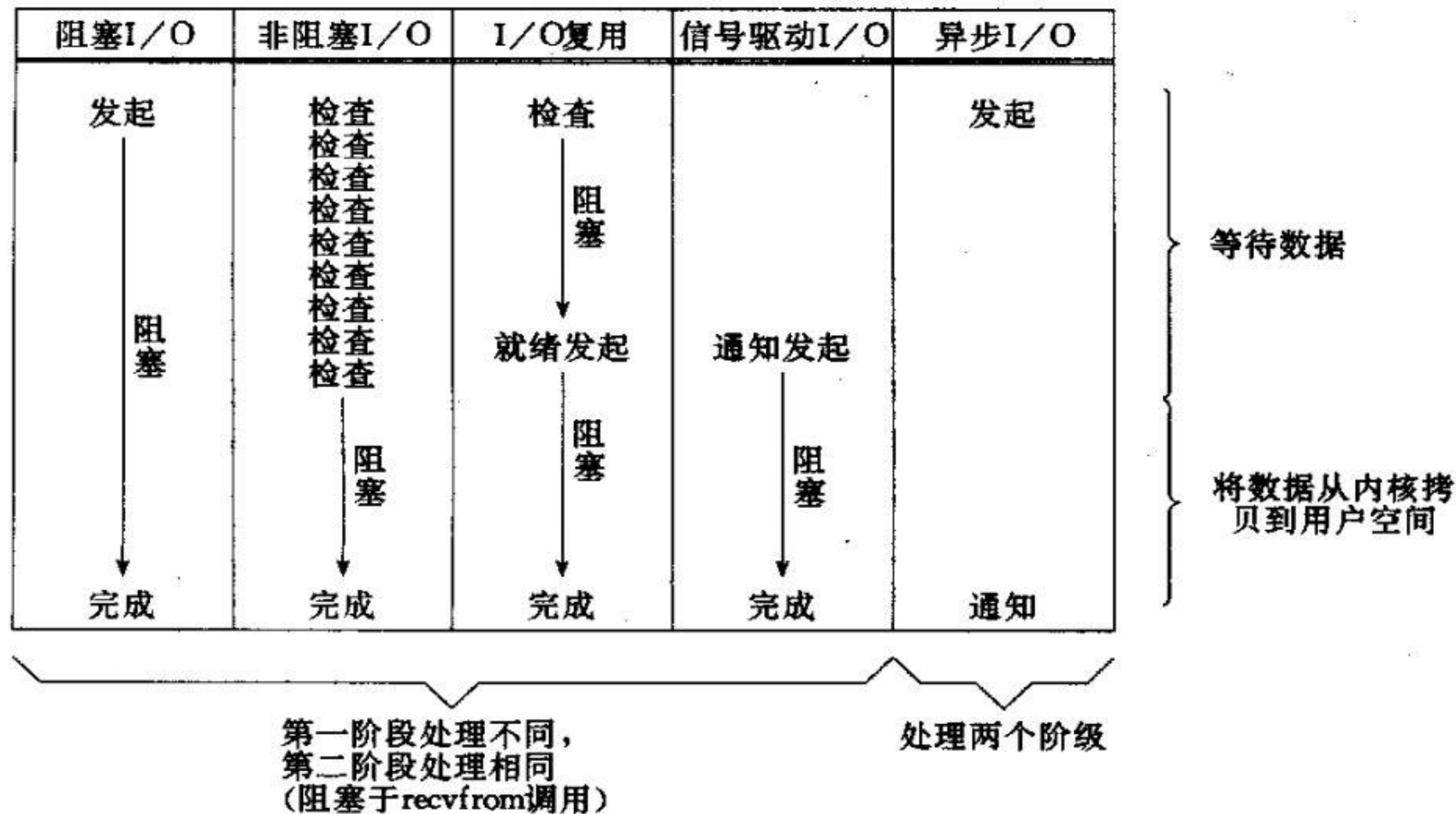


图 6.6 五个 I/O 模型的比较

# IO策略

- 每客户一线程，阻塞IO
- 单线程，多客户，非阻塞→IO多路复用
  - 水平触发
  - 边缘触发
- 单线程，多客户，AIO
- 内核中的服务器

# Java NIO概览——变迁

- NIO = New I/O
- NIO 1: JSR 51
  - JDK 1.4引入
  - <http://jcp.org/en/jsr/detail?id=051>
- NIO 2: JSR203
  - JDK 7
  - <http://jcp.org/en/jsr/detail?id=203>



# NIO Buffers

- [java.nio.buffer](#), 缓冲区抽象
- ByteBuffer
  - 理解capacity、limit、position、mark
  - $0 - \text{mark} - \text{position} - \text{limit} - \text{capacity}$
- Direct ByteBuffer VS. non-direct ByteBuffer
- Non-direct ByteBuffer
  - HeapByteBuffer, 标准的java类
  - 维护一份byte[]在JVM堆上
  - 创建开销小
- Direct ByteBuffer
  - 底层存储在非JVM堆上, 通过native代码操作
  - `-XX:MaxDirectMemorySize=<size>`
  - 创建开销大

# NIO channels

- 数据传输通道的抽象
- 批量数据传输，与Buffer配合很好
- [java.nio.channels.Channel](#)
- [java.nio.channels.FileChannel](#)
- [java.nio.channels.SocketChannel](#)
- [java.nio.channels.ServerSocketChannel](#)
- Blocking or non-blocking

# NIO selectors

- [java.nio.channels.Selector](#)
- 支持IO多路复用的抽象实体
- 注册Selectable Channel
- SelectionKey —— 表示Selector和被注册的channel之间关系，一份凭证
- SelectionKey 保存channel感兴趣的事件
- Selector.select 更新所有就绪的SelectionKey的状态，并返回就绪的channel个数
- 迭代Selected Key集合并处理就绪channel

# NIO带来了什么

- 事件驱动模型
  - 避免多线程
  - 单线程处理多任务
- 非阻塞IO, IO读写不再阻塞, 而是返回0
- 基于block的传输, 通常比基于流的传输更高效
- 更高级的IO函数, zero-copy
- IO多路复用大大提高了java网络应用的可伸缩性和实用性

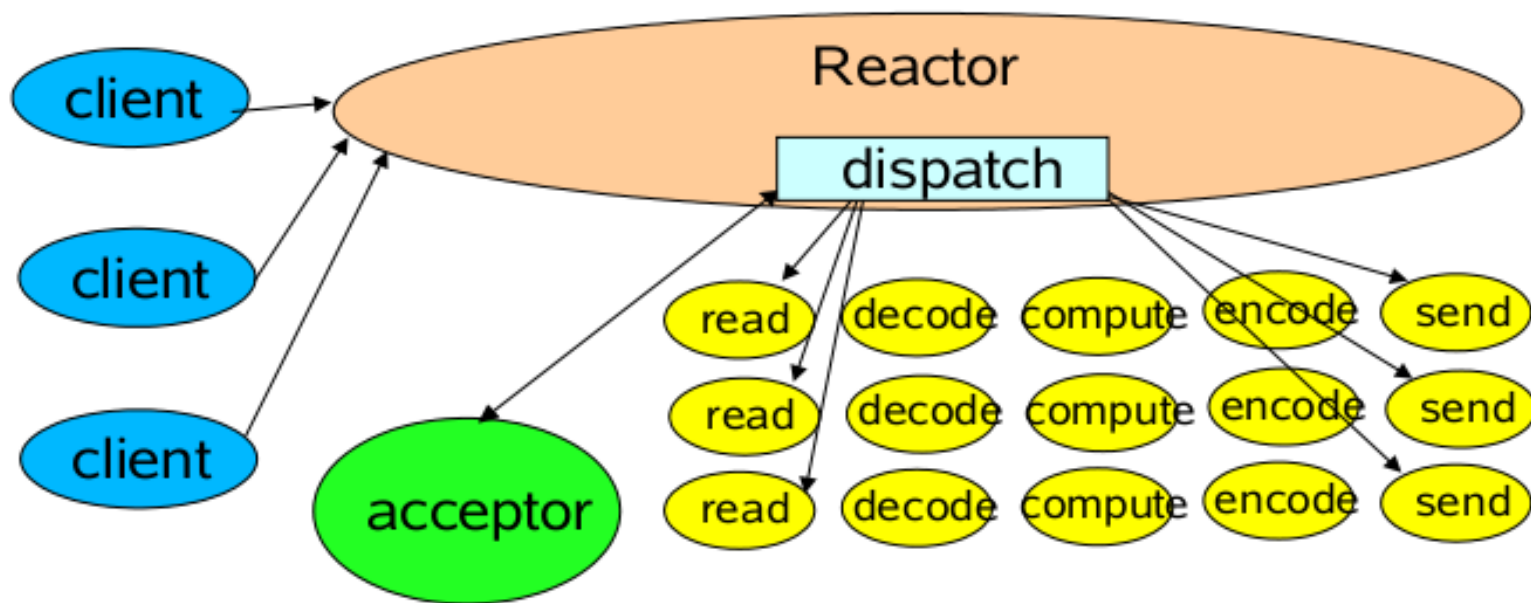
# NIO神话

- 使用NIO = 高性能
  - NIO不一定更快的场景
    - 客户端应用
    - 连接数<1000
    - 并发程度不高
    - 局域网环境下
- NIO完全屏蔽了平台差异
  - NIO仍然是基于各个OS平台的IO系统实现的，差异仍然存在
- 使用NIO做网络编程很容易
  - 离散的事件驱动模型，编程困难
  - 陷阱重重

# Reactor模式

- NIO网络框架的典型模式
- 核心组件
  - Synchronous Event Demultiplexer
    - Event loop + 事件分离
  - Dispatcher
    - 事件派发, 可以多线程
  - Request Handler
    - 事件处理, 业务代码
- Mina Netty Cindy都是此模式的实现

# Reactor示意图



# 理想的NIO框架

- 优雅地隔离IO代码和业务代码
- 易于扩展
- 易于配置，包括框架自身参数和协议参数
- 提供良好的codec框架，方便marshal/unmarshal
- 透明性，内置良好的日志记录和数据统计
- 高性能



# NIO框架性能的关键因素

- 数据的拷贝
- 上下文切换
- 内存管理
- TCP选项、高级IO函数
- 框架设计

# 减少数据拷贝

- ByteBuffer的选择
- View ByteBuffer
- `FileChannel.transferTo/transferFrom`
- `FileChannel.map/MappedByteBuffer`

# ByteBuffer的选择

|      | DirectByteBuffer            | HeapByteBuffer                                              |
|------|-----------------------------|-------------------------------------------------------------|
| 创建开销 | 大                           | 小                                                           |
| 存储位置 | Native heap                 | Java heap                                                   |
| 数据拷贝 | 无需临时缓冲区做拷贝                  | 拷贝到临时<br>DirectByteBuffer，但临时缓冲区使用缓存。<br>聚集写/发散读时没有缓存临时缓冲区。 |
| GC影响 | 每次创建或者释放的时候都调用一次System.gc() |                                                             |

注意：mina 1.1封装的ByteBuffer.allocate默认都是创建DirectByteBuffer, Mina 2.0就不是了

# ByteBuffer的选择

| 场景                      | 选择                                          |
|-------------------------|---------------------------------------------|
| 不知道该用哪种buffer           | Non-Direct                                  |
| 没有参与IO操作                | Non-Direct                                  |
| 中小规模应用(<=1K并发连接)        | Non-Direct                                  |
| 长生命周期, 较大的缓冲区           | Direct                                      |
| 测试证明Direct比Non-Direct更快 | Direct                                      |
| 进程间数据共享(JNI)            | Direct                                      |
| 一个Buffer发给多个Client      | 考虑使用View ByteBuffer共享数据<br>(buffer.slice()) |

# HeapByteBuffer缓存

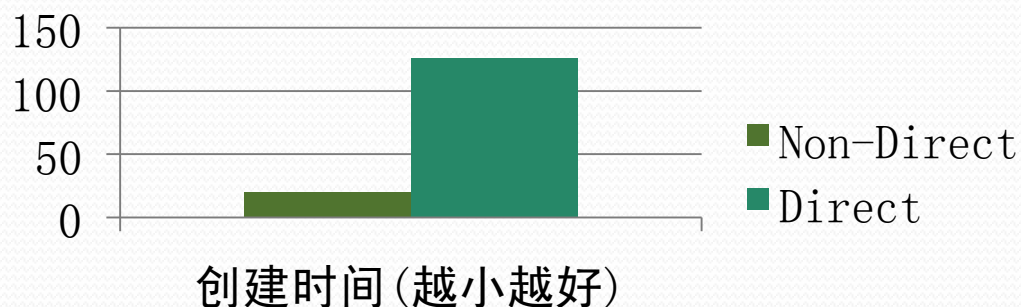
```
static int write(FileDescriptor fd, ByteBuffer src, long position,
 NativeDispatcher nd, Object lock)
 throws IOException
{
 if (src instanceof DirectBuffer)
 return writeFromNativeBuffer(fd, src, position, nd, lock);

 // Substitute a native buffer
 int pos = src.position();
 int lim = src.limit();
 int rem = (pos <= lim ? lim - pos : 0);
 ByteBuffer bb = null;
 try {
 bb = Util.getTemporaryDirectBuffer(rem);
 bb.put(src);
 bb.flip();
 // Do not update src until we see how many bytes were written
 src.position(pos);

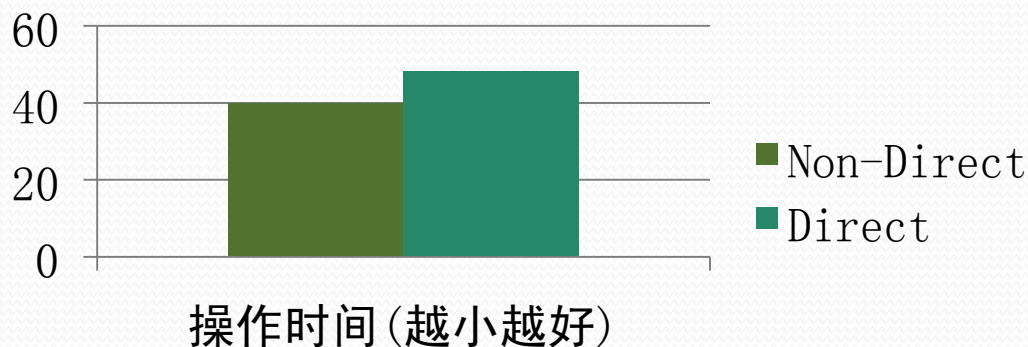
 int n = writeFromNativeBuffer(fd, bb, position, nd, lock);
 if (n > 0) {
 // now update src
 src.position(pos + n);
 }
 return n;
 } finally {
 Util.releaseTemporaryDirectBuffer(bb);
 }
}
```

# Direct Vs. Non-Direct ByteBuffer

- 创建4K缓冲区



- 拷贝16K数组



# View ByteBuffer

```
ByteBuffer buf1 = ByteBuffer.allocate(8);
buf1.putInt(1);
buf1.putInt(2);
buf1.flip();
ByteBuffer buf2 = buf1.slice();
buf1.position(4);
System.out.println(buf1.getInt());
System.out.println(buf2.getInt());
System.out.println(buf2.getInt());
```

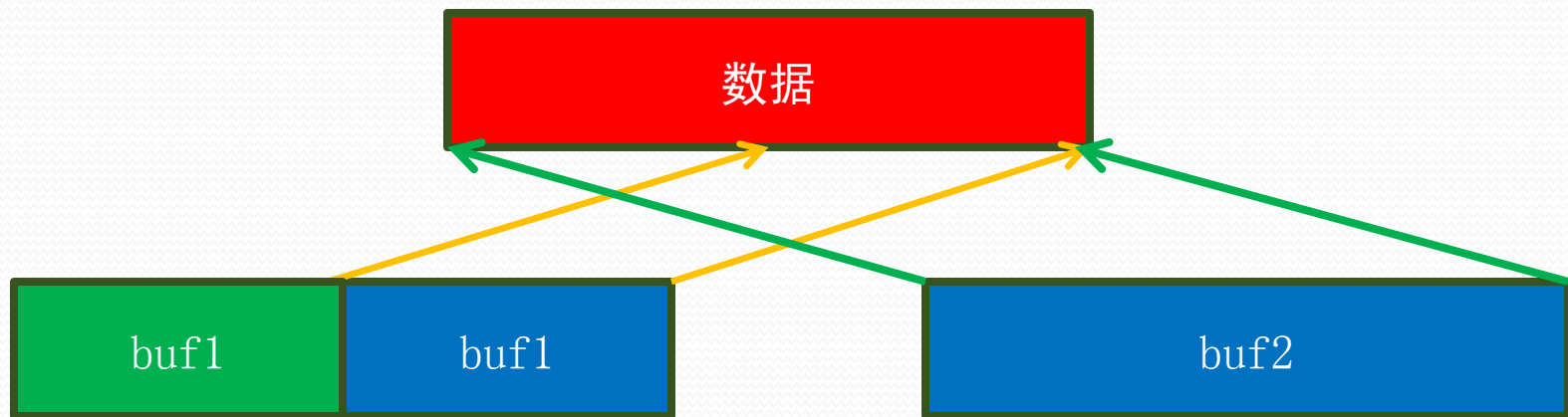
输出:

2

1

2

# View ByteBuffer

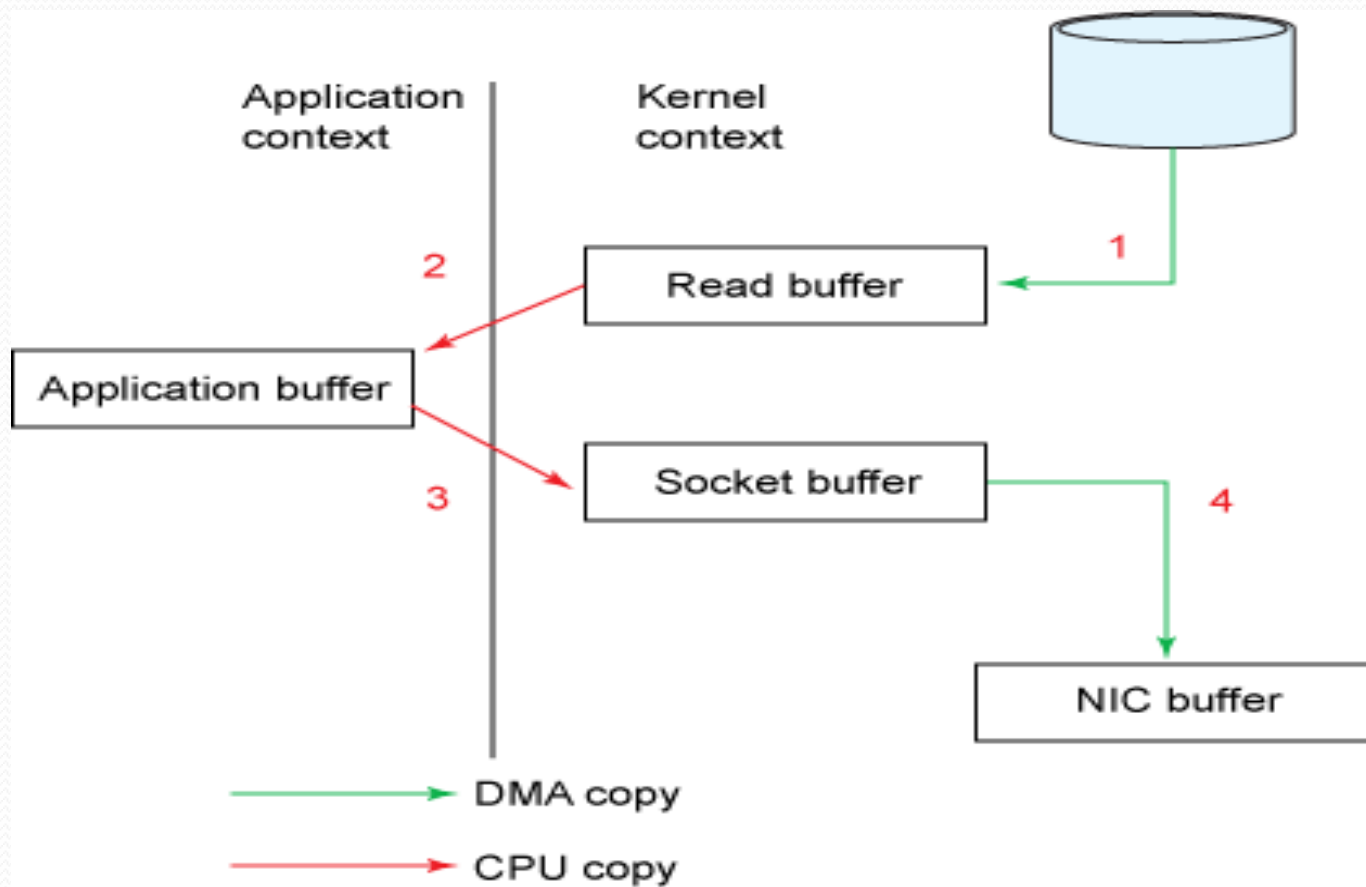




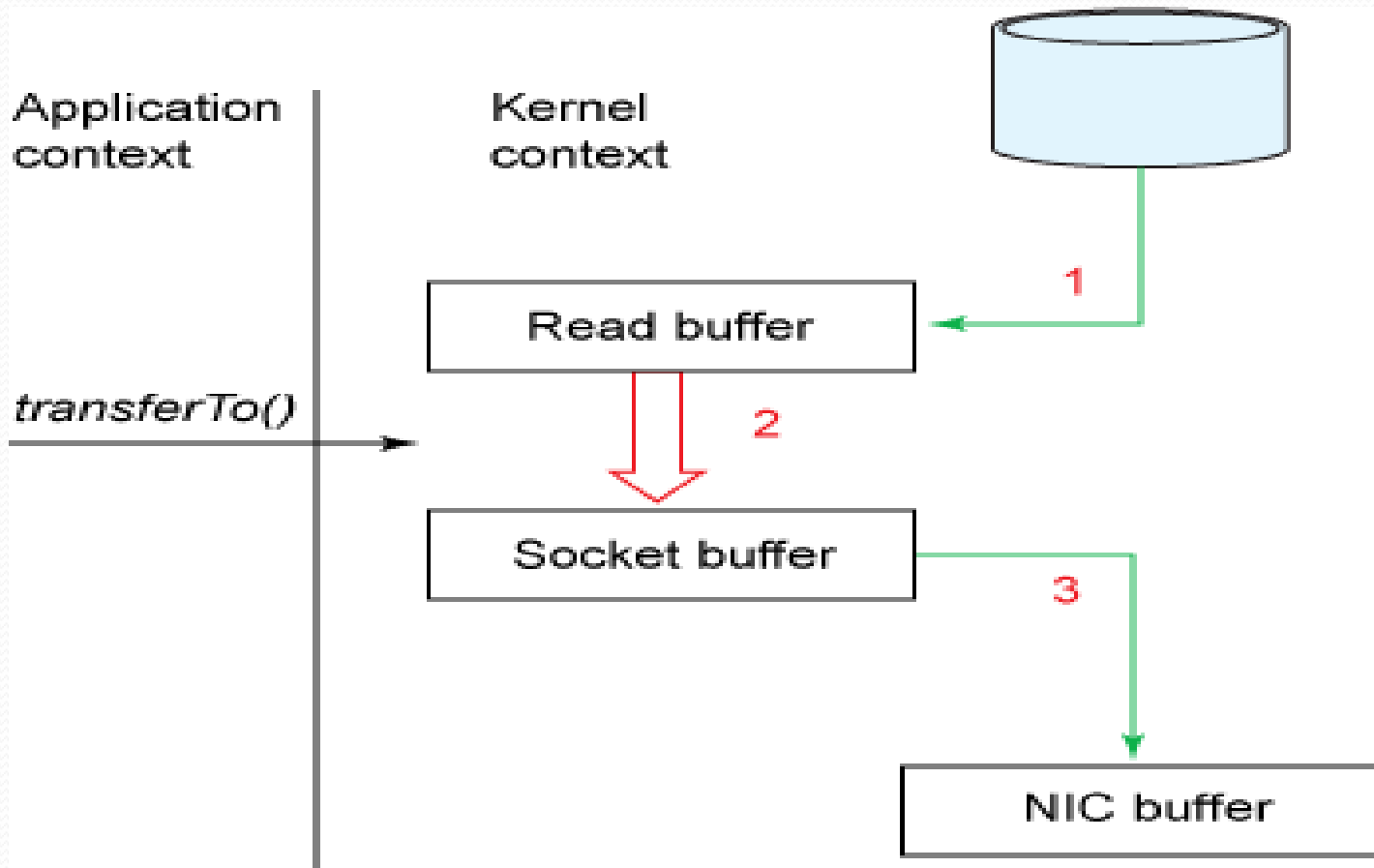
# 传输文件

| 传统方式                                                                                       | FileChannel                                                                                                                                         |
|--------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>byte []buf=new byte[8192];<br/>while(in.read(buf)&gt;0)<br/>    out.write(buf);</pre> | <pre>FileChannel in=...<br/>WritableByteChannel out=...<br/>in.transferTo(0, fsize, out)</pre>                                                      |
| 四次拷贝, 四次用户态和内核态的切换                                                                         | 两次拷贝, 两次切换, 拷贝在kernel进行, *nix系统使用sendfile系统调用。                                                                                                      |
| 久经考验的共产主义战士                                                                                | 性能有60%左右的提升<br>在JDK 6u18之前有致命BUG<br><br><a href="http://bugs.sun.com/view_bug.do?bug_id=5103988">http://bugs.sun.com/view_bug.do?bug_id=5103988</a> |

# 传统方式



# FileChannel.transferTo



# FileChannel.map

- 将文件映射为内存区域——MappedByteBuffer
- 提供快速的文件随机读写能力
- 平台相关
- 适合大文件、只读型操作，如大文件的MD5校验等
- 没有unmap方法，什么时候被回收取决于GC

# 减少上下文切换

- 时间缓存
- Selector.wakeup
- 提高IO读写效率
- 线程模型

# 时间缓存

- 网络服务器通常需要频繁地获取系统时间：定时器、协议时间戳、缓存过期等等。
- `System.currentTimeMillis`
  - Linux调用`gettimeofday`，需要切换到内核态
  - 在我的机器上，1000万次调用需要12秒，平均一次1.3毫秒
  - 大部分应用并不需要特别高的精度
- `SystemTimer.currentTimeMillis`
  - 独立线程定期更新时间缓存
  - `currentTimeMillis`直接返回缓存值
  - 精度取决于定期间隔
  - 1000万次调用降低到59毫秒

# SystemTimer

```
public class SystemTimer {
 private final static ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();

 private static final long tickUnit = Long.parseLong(System.getProperty("notify.systimer.tick", "50"));
 private static volatile long time = System.currentTimeMillis();

 private static class TimerTicker implements Runnable {
 public void run() {
 time = System.currentTimeMillis();
 }
 }

 public static long currentTimeMillis() {
 return time;
 }

 static {
 executor.scheduleAtFixedRate(new TimerTicker(), tickUnit, tickUnit, TimeUnit.MILLISECONDS);
 Runtime.getRuntime().addShutdownHook(new Thread() {
 @Override
 public void run() {
 executor.shutdown();
 }
 });
 }
}
```

# Selector.wakeup

- 主要作用
  - 解除阻塞在`Selector.select()`/`select(long)`上的线程，立即返回。
  - 两次成功的`select`之间多次调用`wakeup`等价于一次调用。
  - 如果当前没有阻塞在`select`上，则本次`wakeup`调用将作用于下一次`select`——“记忆”作用。
- 为什么要唤醒？
  - 注册了新的`channel`或者事件
  - `channel`关闭，取消注册
  - 优先级更高的事件触发（如定时器事件），希望及时处理



# wakeup的原理

- Linux上利用pipe调用创建一个管道
- Windows上则是一个loopback的tcp连接
  - 这是因为win32的管道无法加入select的fd set
- 将管道或者TCP连接加入select fd set
- wakeup往管道或者连接写入一个字节
- 阻塞的select因为有IO事件就绪，立即返回
- 可见，wakeup的调用开销不可忽视

# Wakeup原理

```
JNIEXPORT void JNICALL
Java_sun_nio_ch_EPollArrayWrapper_interrupt(JNI
Env *env, jobject this, jint fd)
{
 int fakebuf[1];
 fakebuf[0] = 1;
 if (write(fd, fakebuf, 1) < 0) {
 JNU_ThrowIOExceptionWithLastError(env, "
write to interrupt fd failed");
 }
}
```

# 减少wakeup调用

- 仅在有需要的时候才调用
  - 如往连接发送数据，通常是缓存在一个消息队列，当且仅当队列为空的注册OP\_WRITE并wakeup

```
boolean needsWakeup=false;
synchronized(queue) {
 if(queue.isEmpty())
 needsWakeup=true;
 queue.add(session);
}
if(needsWakeup) {
 registerOPWrite();
 selector.wakeup();
}
```

# 减少wakeup调用

- 记录调用状态，避免重复调用
  - Netty的优化

```
AtomicBoolean wakenUp = new AtomicBoolean();
wakenUp.set(false); //select之前设置为false
selector.select(500);
```

```
if (wakenUp.compareAndSet(false, true)) {
 selector.wakeup();
}
```

# 读到或者写入0个字节

- 读到0个字节，或者写入0个字节
  - 不代表连接关闭
  - 高负载或者慢速网络下很常见的情况
  - 通常的处理办法是返回并继续注册OP\_READ/OP\_WRITE等待下次处理。
    - 缺点：系统调用开销，线程切换开销
  - 其他解决办法
    - 循环忙等待或者yield一定次数
      - Netty: writeSpinCount=16
      - Kilim: YIELD\_COUNT=4
      - Mina: WRITE\_SPIN\_COUNT =256
    - 启用Temporary Selector在当前线程注册并poll
      - Girzzy

# 循环一定次数写入

```
//Mina
private static final int WRITE_SPIN_COUNT = 256;

for (int i = WRITE_SPIN_COUNT; i > 0; i--) {
 localWrittenBytes = write(session, buf, length);
 if (localWrittenBytes != 0) {
 break;
 }
}
```

# 使用临时selector

```
Selector readSelector = SelectorFactory.getSelector();
SelectionKey tmpKey = null;
try {
 tmpKey = this.selectableChannel.register(readSelector, 0);
 tmpKey.interestOps(tmpKey.interestOps() | SelectionKey.OP_READ);
 int code = readSelector.select(500);
 tmpKey.interestOps(tmpKey.interestOps() & ~SelectionKey.OP_READ);
 if (code > 0) {
 //try to read from channel
 }
}
finally {
 if (tmpKey != null) {
 tmpKey.cancel();
 tmpKey = null;
 }
 if (readSelector != null) {
 // Cancel the key.
 readSelector.selectNow();
 SelectorFactory.returnSelector(readSelector);
 }
}
```

# 在当前线程写入

- 当发送缓冲队列为空的时候，可以直接往channel写数据，而不是放入缓冲队列，interest了OP\_WRITE等待IO线程写入，一定程度上可以提高发送效率。
  - 优点：减少系统调用和线程切换
  - 缺点：当前线程中断会引起channel关闭

```
if queue.isEmpty
 if writeLock.tryLock &&
 current.compareAndSet(null, msg)
 write to channel
 else
 queue.offer(msg);
 interest OP_WRITE
```



# 线程模型

- Selector的三个主要事件：OP\_READ、OP\_WRITE和OP\_ACCEPT，都可以运行在不同的线程。
- 通常Reactor实现为一个线程
  - 内部维护一个Selector

```
while(true) {
 int sel=selector.select(timeout);
 processRegister();
 if(sel>0)
 processSelected();
}
```

# Reactor数目

- Boss thread + worker thread
  - Boss处理OP\_ACCEPT、OP\_CONNECT, 处理连接接入
  - Worker处理OP\_READ、OP\_WRITE, 处理IO读写
- Reactor线程数目
  - Netty  $1+2*\text{cpu}$
  - Mina  $1+\text{cpu}+1$
  - Grizzly  $1+1$

# 常见线程模型

- OP\_READ和OP\_ACCEPT都运行在reactor线程
- OP\_ACCEPT运行在reactor，OP\_READ运行在单独的线程。
- OP\_READ和OP\_ACCEPT都运行在单独的线程
- OP\_READ运行在reactor线程，而OP\_ACCEPT运行在单独的线程

# 线程模型的选择

- 类echo应用，unmashall和业务处理的开销非常低，选择第一种模型。
  - 创建线程和切换线程的开销
- 第二、第三、第四种模型，从测试来看，OP\_ACCEPT的处理开销很低
  - 从已经完成三路握手的队列移出
- 最佳选择：第二种模型
  - unmashall一般是cpu-bound
  - 业务逻辑代码通常比较耗时，不要在reactor线程处理

# 线程模型的设置

- Mina、Netty设置线程模型的示例代码（略）

# 内存管理

- Java能做的事情有限
  - GC带来的自动内存管理
- 缓冲区的管理
  - 池化
    - ThreadLocal cache
    - 环形缓冲区
  - 扩展
    - putString, getString等高级API
    - 缓冲区自动扩展和收缩, 处理不定长度字节
  - 字节序
    - 跨语言通讯需要注意
    - 网络字节序-BigEnd
    - 默认缓冲区-BigEnd
    - Java的IO库和class文件-BigEnd

# 数据结构的选择

- 使用简单的数据结构
  - 链表、队列、数组、散列表
- 使用j. u. c框架引入的并发集合类
  - lock-free, spin lock
- 任何数据结构都要注意容量限制
  - OutOfMemoryError
- 适当选择数据结构的初始容量
  - 降低GC带来的影响

# 定时器的实现

- 定时器在网络程序中频繁使用
  - 周期性事件的触发
  - 异步超时通知和移除
  - 延迟事件的触发
- 三个时间复杂度
  - 插入定时器
  - 删除定时器
  - PerTickBookkeeping, 一次tick内, 系统需要执行的操作
- Tick的方式
  - `Selector.select(timeout)`
  - `Thread.sleep(timeout);`



# 定时器实现：链表

- 将定时器组织成链表结构
- 插入定时器，加入链表尾部
  - $O(1)$
- 删除定时器
  - $O(1)$
- PerTickBookkeeping，遍历链表查找expire事件
  - $O(n)$

# 定时器实现：排序链表

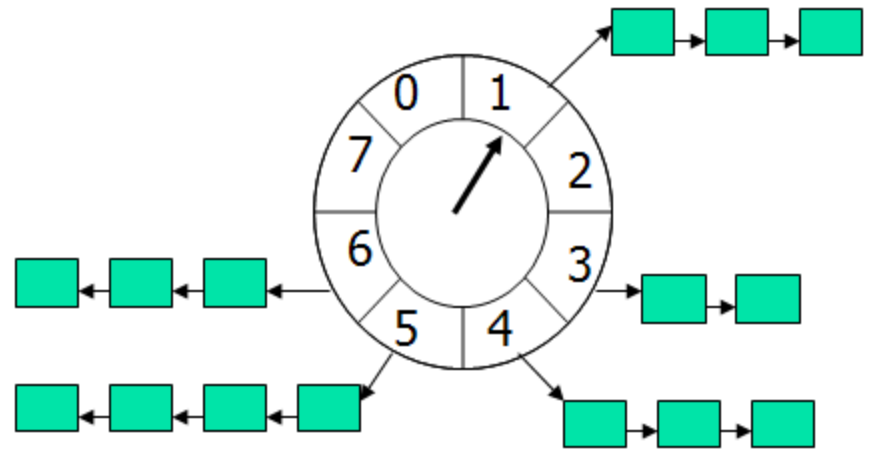
- 将定时器组织成有序链表结构，按照`expire`截止时间升序排序
- 插入定时器，找到合适的位置插入
  - $O(n)$
- 删除定时器
  - $O(1)$
- `PerTickBookkeeping`，直接从表头找起
  - $O(1)$

# 定时器的实现：优先队列

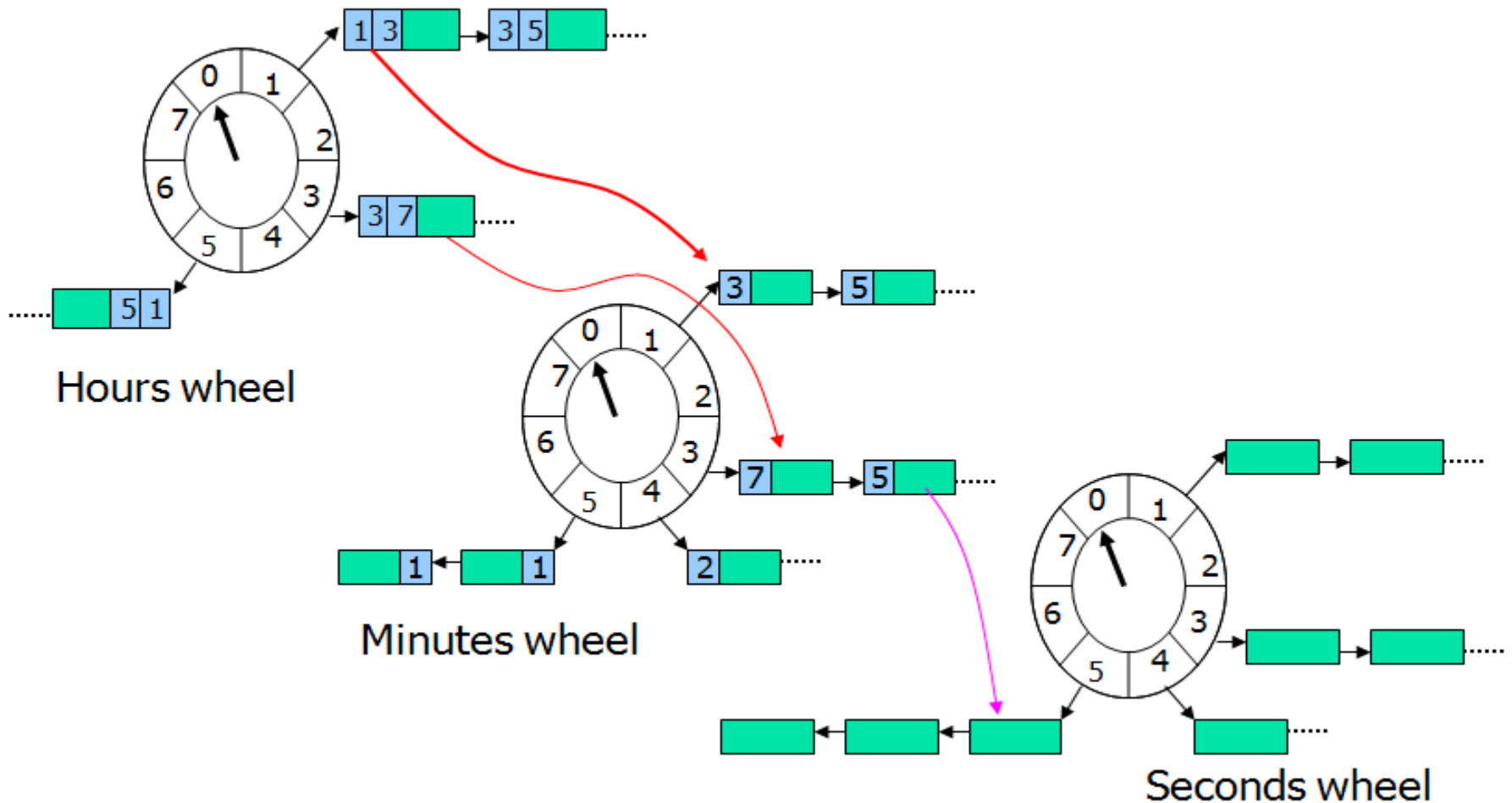
- 将定时器组织成优先队列，按照`expire`截止时间作为优先级，优先队列一般采用最小堆实现
- 插入定时器
  - $O(\lg(n))$
- 删除定时器
  - $O(\lg(n))$  or  $O(1)$
- PerTickBookkeeping，直接取`root`判断
  - $O(1)$

# 定时器实现：Hash Wheel Timer

- 将定时器组织成时间轮
- 指针按照一定周期旋转，一个tick跳动一个槽位
- 定时器根据延时时间和当前指针位置插入到特定槽位
- 插入定时器
  - $O(1)$
- 删除定时器
  - $O(1)$
- PerTickBookKeeping
  - $O(1)$
- 槽位和tick决定了精度和延时



# 定时器实现：Hierarchical Timing Wheel



# 连接IDLE的判断

- 连接处于Idle状态：一段时间内没有IO读写事件发生
  - 读Idle，一段时间内没有IO读
  - 写Idle，一段时间内没有IO写
  - Both，一段时间内没有IO读写
- 实现方式：
  - 每次IO读写都记录IO读和写的时间戳
  - 定时扫描所有连接，判断当前时间与上一次读或者写的时间差是否超过设定阈值，超过即认为连接处于Idle状态，通知业务处理器。
  - 定时的方式：基于select(timeout)或者定时器
    - Mina: select(timeout)
    - Netty: HashWheelTimer

# 合理设置TCP/IP选项

- 合理设置TCP/IP在某些时候可以起到显著的效果。
- 需要根据应用类型、协议设计、网络环境、OS平台等因素做考量，以实际测试结果为准。

# Socket缓冲区：SO\_RCVBUF和SO\_SNDBUF

- `Socket.setReceiveBufferSize/setSendBufferSize`
  - 仅仅是对底层平台的一个提示，是否有效取决于底层平台，因此`get`返回的可能不是你设置的值，也可能不是底层平台实际使用的值。
- 设置原则
  - 在以太网上，4k通常是不够的，增加到16K，吞吐量增加了40%。
  - Socket缓冲区大小至少应该是连接的MSS的三倍， $MSS=MTU+40$ ，一般以太网卡的 $MTU=1500$ 字节。
    - MSS：最大分段大小
    - MTU：最大传输单元
  - send buffer最好与对端的receive buffer一致。
  - 对于一次性发送大量数据的应用，增加发送缓冲区到48K、64K可能是唯一的最有效地提高性能的方式。为了最大化性能，发送缓冲区可能至少要跟BDP（带宽延迟乘积）一样大小。
  - 同样，对于大量接收数据的应用，提高接收缓冲区，能减少发送端的阻塞。
  - 如果应用既发送大量数据，也接收大量数据，recv buffer和send buffer应该同时增加
  - 如果要设置ServerSocket的recv缓冲区超过RFC1323中定义的64k，那么必须在绑定端口前设置，以后accept产生的socket将继承这一设置。
  - 无论缓冲区大小多少，你都应该尽可能地帮助TCP至少以那样的大小的块写入



# 带宽延迟乘积——BDP

- 为了优化 TCP 吞吐量（假设为合理的无差错传输路径），发送端应该发送足够的数据包以填满发送端和接收端之间的逻辑管道。
- 逻辑管道的容量计算
  - $BDP = \text{带宽} \times RTT$

# Nagle算法：SO\_TCPNODELAY

- NAGLE算法通过将缓冲区内的封包自动相连，组成较大的封包，阻止大量小封包的发送阻塞网络，从而提高网络应用效率
  - 默认打开

- 算法描述

```
if there is new data to send
 if the window size >= MSS and available data is >= MSS
 send complete MSS segment now
 else
 if there is unconfirmed data still in the pipe
 enqueue data in the buffer until an acknowledge is received
 else
 send data immediately
 end if
 end if
end if
```

- 对于实时性要求较高的应用(telnet、网游)，可能需要关闭此算法。
  - `Socket. setTcpNoDelay(true);`
  - 注意, true为关闭此算法, false为开启

# SO\_LINGER选项

- `Socket. setSoLinger(boolean linger, int timeout)`
  - 控制socket关闭后的行为
- 默认行为:`linger=false, timeout=-1`
  - 当socket主动close, 调用的线程会马上返回, 不会阻塞, 然后进入CLOSING状态, 残留在缓冲区中的数据将继续发送给对端, 并且与对端进行FIN-ACK协议交换, 最后进入TIME\_WAIT状态
- `Linger=true, timeout>0`
  - 调用close的线程将阻塞, 发生两种可能的情况: 一是剩余的数据继续发送, 进行关闭协议交换, 二就是超时过期, 剩余的数据将被删除, 进行FIN-ACK交换。
- `Linger=true, timeout=0`
  - 进行所谓“hard-close”, 任何剩余的数据都将被丢弃, 并且FIN-ACK交换也不会发生, 替代产生RST, 让对端抛出“connection reset”的SocketException
- 慎重使用该选项, TIME\_WAIT状态的价值
  - 可靠实现TCP连接终止
  - 允许老的分节在网络中流失, 防止发给新的连接。
  - 持续时间=2\*MSL
    - MSL是最大分节生命期, 一般为30秒——2分钟

# SO\_REUSEADDR:重用端口

- `Socket.setReuseAddress(boolean)`
  - 默认一般为false
- 适用场景
  - 当有一个有相同本地地址和端口的socket1处于TIME\_WAIT状态时，而你启动的程序的socket2要占用该地址和端口，你的程序就要用到该选项。
  - SO\_REUSEADDR允许同一port上启动同一服务器的多个实例(多个进程)。但每个实例绑定的IP地址是不能相同的。在有多块网卡或用IP Alias技术的机器可以测试这种情况。
  - SO\_REUSEADDR允许完全相同的地址和端口的重复绑定。但这只用于UDP的多播，不用于TCP。
  - 题外：SO\_REUSEPORT
    - Listen做四元组，多进程同一IP同一端口做accept，适合大量短连接的web server
    - FreeBSD独有，google提交了linux patch
    - <http://pdxplumbers.osuosl.org/2010/ocw/proposals/489>
    - <http://kerneltrap.org/mailarchive/linux-netdev/2010/4/19/6274993>

# 其他选项

- Performance preference, JDK5引入
  - `Socket.setPerformancePreferences(connectionTime, latency, bandwidth)`
  - 设置连接时间、延迟、带宽的相对重要性
- SO\_KEEPALIVE
  - `Socket.setKeepAlive(boolean)`
  - 这是TCP层，而非HTTP协议的keep-alive概念
  - 默认一般为false，用于TCP连接保活，默认间隔2个小时
  - 更建议在应用层做心跳
- 带外数据
  - `Socket.sendUrgentData(data)`

# 奇淫技巧

- 读写公平
  - Mina限制一次写入的字节数不超过最大的读缓冲区大小的1.5倍
- 针对FileChannel.transferTo的bug
  - Mina判断异常，如果是temporarily unavailable的IOException，则认为传输字节数为0
- 发送消息，通常是放入一个缓冲队列注册OP\_WRITE等待IO线程去写
  - 线程切换、系统调用
  - 如果队列为空，直接在当前线程channel.write
    - 隐患：当前线程的中断会引起连接关闭

# 奇淫技巧

- 事件处理优先级：
  - ACE框架推荐: `Accept > Write > Read`
  - Mina和Netty: `Read > Write`
  - 个人倾向于ACE的处理顺序，优先发送
- 处理事件注册的顺序
  - 在select之前
  - 在select之后
    - 处理wakeup竞争条件

# Java socket实现在不同平台上的差异

- 由于各种OS平台的socket实现都不尽相同，这些差异同样影响到Java Socket的实现
- 在做Java网络编程的时候需要考虑这些实现
  - 性能
  - 健壮性
- 下面这些图都引自 《Fundamental NetWorking in java》



TABLE B.1 Platform dependencies affecting Java TCP/IP

| Class.method                                                                               | Issue                                                                                             | Comments                                                                                                                                                                                                                    |
|--------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DatagramSocket<br>.send                                                                    | Maximum size of a datagram: 65507 in the UDP protocol RFC; bounded by the socket send buffer size | Linux: 9216 bytes. <sup>a</sup>                                                                                                                                                                                             |
| ServerSocket<br>.bind                                                                      | Length of default backlog queue; adjustment to application-supplied value                         | Originally 5; seems to be at least 50 on most current platforms; varies between workstation and server versions of Windows.                                                                                                 |
| Socket<br>.close and<br>SocketChannel<br>.close<br>after calling<br>Socket<br>.setSoLinger | If timeout is set and expires, whether unsent data is still sent or the connection is reset       | <p>Unix-based platforms leave unsent data queued for transmission; Windows resets the connection.</p> <p>On Linux, if a positive linger timeout is set, SocketChannel.close blocks even if the channel is non-blocking.</p> |
| Socket<br>.connect                                                                         | Behaviour if the target backlog queue is full                                                     | Unix-based platforms ignore the connect request, so the client times out and retries within connect.                                                                                                                        |



Socket  
.getReceiveBufferSize  
Socket  
.getSendBufferSize

Default size of socket  
buffers

Windows: issues a reset, so  
Windows clients therefore also  
retry within connect on  
receiving a reset. This is not  
the intent of the RFC.

Originally 2k.

Up to 56k on various Unix-  
based platforms:

FreeBSD: send=32k,  
receive=56k.

Linux: send=16k,  
receive=43689 bytes (!).

Solaris 52k.

Windows: 8k.<sup>a</sup>

TABLE B.1 Platform dependencies affecting Java TCP/IP (continued)

| Class.method                                                    | Issue                                                  | Comments                                                                                                                                                                            |
|-----------------------------------------------------------------|--------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Socket<br>.setKeepalive                                         | Whether supported by the platform; keep-alive interval | Can be detected by checking Socket.getKeepalive after calling setKeepalive.<br><br>Keep-alive interval is normally 2 hours globally and if changeable requires privilege to change. |
| Socket<br>.setReceiveBufferSize<br>Socket<br>.setSendBufferSize | Adjustment to application-supplied values              | Will be adjusted to fit the platform's maxima and minima (see below), and may be rounded up or down to suit the platform's buffer-size granularity as well.                         |
| Socket<br>.setReceiveBufferSize                                 | Maximum size of socket buffers                         | FreeBSD: depends on various kernel constants.                                                                                                                                       |

Socket  
.setSendBufferSize

Linux: 131070 bytes.<sup>a</sup>

Windows: 131070 bytes.<sup>a</sup>

Socket  
.setReceiveBufferSize

Minimum size of socket buffers

FreeBSD: 1 byte.

Linux: send=2048,  
receive=256 bytes.

Socket  
.setSendBufferSize

Windows: zero (!).

Socket  
.setSoLinger

Maximum linger timeout value, nominally  $2^{31} - 1$  seconds in Java specification

Some platforms limit it to  $(2^{15} - 1)/100 = 32.767$  seconds, by using an internal 16-bit signed quantity representing hundredths of a second.

Socket  
.shutdownInput


Whether SelectionKey.OP\_READ is selected for such a socket

FreeBSD: yes

Linux: yes

Windows: no.<sup>b</sup>



| Class.method                                                                            | Issue                                                       | Comments                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-----------------------------------------------------------------------------------------|-------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Socket<br>.shutdownInput<br>at receiver<br><br>all write, writeXXX<br>methods at sender | Behaviour as seen by remote sender                          | Most Unix-based platforms accept and ignore the data, so the sender's writes all succeed. <br><br>Windows sends an RST, so the sender incurs a SocketException 'connection reset by peer'<br><br>Linux accepts and buffers the data but cannot transmit it to the local application, so the sender eventually gets blocked in write, or is returned zero from non-blocking writes. |
| Socket<br>.shutdownOutput                                                               | Whether SelectionKey.OP_WRITE is selected for such a socket | FreeBSD: no<br>Linux: no<br>Windows: yes. <sup>b</sup>                                                                                                                                                                                                                                                                                                                                                                                                               |

a. This is too small for modern Ethernets or high-latency links such as DSL or ADSL. Socket buffers should be at least equal to the bandwidth-delay product for the intervening network: at least 16k on a 10Mb LAN, more like 63k on 100Mb LANs or links with high latency, or a multiple of 64k when window-scaling can be used.

b. Note the inconsistency of all platforms as between their behaviour for shutdownInput/OP\_READ and shutdownOutput/OP\_WRITE.

# 死神来了

- NIO编程很容易吗?不容易吗? 很容易吗? 不容易吗? .....
- 有些陷阱你需要知道

# 陷阱1：处理事件忘记移除key

- 在select返回值大于0的情况下，循环处理Selector.selectedKeys集合，每处理一个必须移除
- ```
Iterator<SelectionKey> it=set.iterator();  
While(it.hasNext()) {  
    SelectionKey key=it.next();  
    it.remove(); //切记移除  
    .....处理事件  
}
```
- 不移除的后果是本次的就绪的key集合下次会再次返回，导致无限循环，CPU消耗 100%

陷阱2：Selector返回的key集合 非线程安全

- Selector.selectedKeys/keys 返回的集合都是非线程安全的
 - Selector.selectedKeys返回的可移除
 - Selector.keys 不可变
- 对selected keys的处理必须单线程处理或者适当同步

陷阱3：正确注册Channel和更新interest

- 直接注册不可吗？
 - `channel.register(selector, ops, attachment);`
- 不是不可以, 效率问题
 - 至少加两次锁, 锁竞争激烈
 - Channel本身的`regLock`, 竞争几乎没有
 - Selector内部的key集合, 竞争激烈
- 更好的方式: 加入缓冲队列, 等待注册, reactor单线程处理

```
If(isReactorThread()) {  
    channel.register(selector, ops, attachment);  
}  
else {  
    register.offer(new Event(channel, ops, attachment));  
    selector.wakeup();  
}
```

陷阱3：正确注册Channel和更新interest

- 同样，`SelectionKey.interest(ops)`
 - 在linux上会阻塞，需要获取selector内部锁做同步
 - 在win32上不会阻塞
- 屏蔽平台差异，避免锁的激烈竞争，采用类似注册channel的方式：

```
if (this.isReactorThread()) {  
    key.interestOps(key.interestOps() | SelectionKey.OP_READ);  
}  
else {  
    this.register.offer(new Event(key, SelectionKey.OP_READ));  
    selector.wakeup();  
}
```

陷阱4:正确处理OP_WRITE

- OP_WRITE处理不当很容易导致CPU 100%
- OP_WRITE触发条件
 - 前提:interest了OP_WRITE
 - 触发条件:
 - socket发送缓冲区可写
 - 远端关闭
 - 有错误发生
- 正确的处理方式
 - 仅在已经连接的channel上注册
 - 仅在有数据可写的时候才注册
 - 触发之后立即取消注册, 否则会继续触发导致循环
 - 处理完成后视情况决定是否继续注册
 - 没有完全写入, 继续注册
 - 全部写入, 无需注册

陷阱5：正确取消注册channel

- SelectableChannel一旦注册将一直有效直到明确取消
- 怎么取消注册？
 - `channel.close()`，内部会调用`key.cancel()`
 - `key.cancel()`；
 - 中断 channel的读写所在线程引起的channel关闭
- 但是这样还不够！
 - `key.cancel()` 仅仅是将key加入`cancelledKeys`
 - 直到下一次select才真正处理
 - 并且channel的`socketfd`只有在真正取消注册后才会`close(fd)`

陷阱5：正确取消注册channel

- 后果是什么？
 - 服务端，问题不大，select调用频繁
 - 客户端，通常只有一个连接，关闭channel之后，没有调用select就关闭了selector
 - sockfd没有关闭，停留在CLOSE_WAIT状态
- 正确的处理方式，取消注册也应当作为事件交给reactor处理, 及时wakeup做select
- 适当的时候调用selector.selectNow()
 - Netty 在超过 256 连接关闭的时候主动调用一次selectNow

陷阱5：正确取消注册 Channel——netty的处理

```
static final int CLEANUP_INTERVAL = 256;
private boolean cleanUpCancelledKeys() throws IOException {
    if (cancelledKeys >= CLEANUP_INTERVAL) {
        cancelledKeys = 0;
        selector.selectNow();
        return true;
    }
    return false;
}
```

```
//channel关闭的时候
channel.socket.close();
cancelledKeys ++;
```

陷阱6：同时注册OP_ACCEPT和OP_READ，同时注册OP_CONNECT和OP_WRITE

- 在底层来说，只有两种事件：read和write
- Java NIO还引入了OP_ACCEPT和OP_CONNECT
 - OP_ACCEPT、OP_READ == Read
 - OP_CONNECT、OP_WRITE == Write
- 同时注册OP_ACCEPT和OP_READ，或者同时注册OP_CONNECT和OP_WRITE在不同平台上产生错误的行为，避免这样做！

陷阱7：正确处理connect

- `SocketChannel.connect`方法在非阻塞模式下可能返回false，切记判断返回值
 - 如果是loopback连接，可能直接返回true，表示连接成功
 - 返回false，后续处理
 - 注册channel到selector, 监听OP_CONNECT事件
 - 在OP_CONNECT触发后，调用`SocketChannel.finishConnect`成功后，连接才真正建立
- 陷阱：
 - 没有判断connect返回值
 - 没有调用`finishConnect`
 - 在OP_CONNECT触发后，没有移除OP_CONNECT，导致`SelectionKey`一直处于就绪状态，空耗CPU
 - OP_CONNECT只能在还没有连接的channel上注册

陷阱8：NIO的那些bug

- http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6403933
- http://bugs.sun.com/view_bug.do?bug_id=6693490
- 现象：导致已经关闭的连接一直处于就绪状态，`select(timeout)`不阻塞，CPU消耗100%
- 解决：
 - 升级到jdk 6u4以上版本
 - 代码上进行规避

陷阱8：NIO的那些bug

- 代码如何规避此bug?
 - 简单方案：在每次`channel.close()`之后马上调用`select`
 - Jetty 6的方案：
 - 当`select`阻塞时间远远小于设置值
 - 取消所有`interest`为0的key
 - 重新创建`Selector`并注册有效的key
 - 还不够健壮，`select`可能由于中断或者`wakeup`唤醒，导致误判
 - 更完善：加入`wakeup`判断和中断状态判断
 - Mina有规避此bug的代码
 - Netty3，定期调用`selectNow`

陷阱8：NIO的那些bug

- 无论如何，请尽量使用最新版本JDK

忠告

- 尽量不要尝试实现自己的nio框架，除非有经验丰富的工程师
- 尽量使用经过广泛实践的开源NIO框架Mina、Netty3、xSocket
- 尽量使用最新稳定版JDK
- 遇到问题的时候，也许你可以先看下 java 的 bug database

NIO框架设计

- 责任链
 - Mina: filter chain
 - Netty: pipeline
 - SEDA架构
- 业务处理器
 - 回调方法
 - 业务逻辑碎片化
- 提供更友好的API
 - xSocket, 提供同步API, 流式API
 - 方便codec框架, 包括常用协议的codec实现
- 流量控制
- 统计和监控, 透明性

基于协程的NIO框架

- 协程：协作式的轻量级线程
 - 半对称：yield/resume
 - 需要调度器
 - 全对称：transform
- NIO + 协程 = 优雅的异步代码
 - 以同步的方式写异步代码
 - 实例：kilim框架

```
Task task = new Task() {  
    @Override  
    public void execute() throws Pausable, Exception {  
        HttpClient client = new HttpClient();  
        HttpResponse resp = client.get( "http://www.taobao.com" );  
        String body=resp.content()  
    }  
};  
task.start();
```

基于Kilim的HttpClient

- 1、HttpClient.get不会阻塞，而是会挂起当前task，在数据达到后resume此task
- 2、Task是轻量级的线程

NIO 2.0

- 更易用的文件API
 - FileSystem
 - Path
 - Directory
 - FileVisitor
 - Attribute
- 文件变更通知
 - WatchService
- <http://java.sun.com/developer/technicalArticles/javase/nio/>
- Java AIO
 - Windows: IOCP
 - Linux: epoll模拟
- 未来NIO框架的变化
 - 风格偏向AIO的方式
 - 在win32上预计性能会有比较大的提升
 - 与协程结合的可能性
 - 闭包带来的影响



QA



Thank you