高性能网络通讯

协议、传输、线程

阿里巴巴-B2B-平台技术部-应用框架
刘超 梁飞@Dubbo
2012 4 11

提纲

- Simple RPC & Problem
- Data Protocol
  - Header & Codec
  - Serialization
- IO Model & NIO
  - Java NIO
  - TCP & linux Kernal
- Threading

简单远程调用

客户端

代码

```
HelloService helloService=getProxy(HelloService.class,”127.0.0.1”,”2580);
System.out.println(helloService.sayHello(“hi, charles”));
Pu blic  <T>  T getProxy(Class<T> interfaceClass, String host, int port){
return (T) Proxy.newProxyInstance(interfaceClass.getClassLoader(), new
        Class<?>[] {interfaceClass},
        new InvocationHandler() {
        public Object invoke(Object proxy, Method method, Object[]
        arguments) throws Throwable {
        Socket socket = new Socket(host, port);
        ObjectOutputStream output = new
        ObjectOutputStream(socket.getOutputStream());
        output.writeUTF(method.getName());
        output.writeObject(method.getParameterTypes());
        output.writeObject(arguments);
        ObjectInputStream input = new
        ObjectInputStream(socket.getInputStream());
        return input.readObject();}
    });
}
```

输出：

```
hi,Charles ,wellcome.
```

服务端

代码：

```
HelloServiceImple:
Public String sysHello(String input){return input+” ,wellcome.”}
Public void static Main(String[] args){
ServerSocket server = new ServerSocket(port);
for(;;) {
        final Socket socket = server.accept();
        new Thread(new Runnable() {
        ObjectInputStream input = new
        ObjectInputStream(socket.getInputStream());
        String methodName = input.readUTF();
        Class<?>[] parameterTypes = (Class<?>[])input.readObject();
        Object[] arguments = (Object[])input.readObject();
        ObjectOutputStream output = new
        ObjectOutputStream(socket.getOutputStream());
        Method method = service.getClass().getMethod(methodName,
                parameterTypes);
        Object result = method.invoke(service, arguments);
        output.writeObject(result);
        } ).start();
}}
```

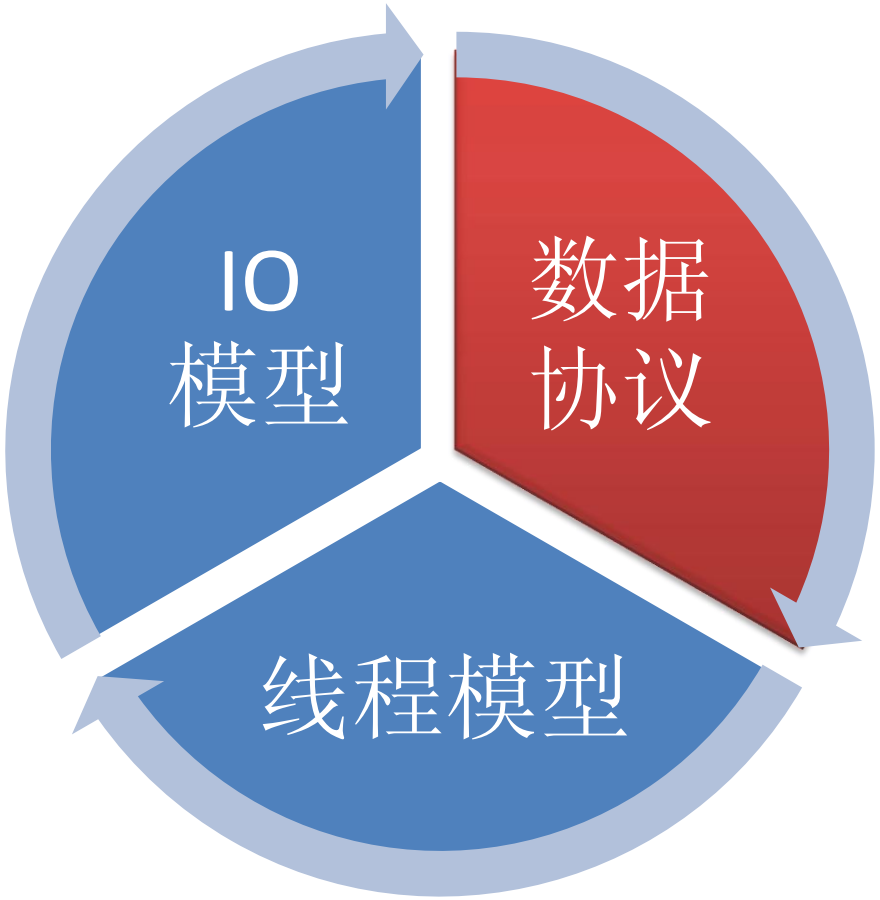
问题在哪里

- 网络传输方式：BIO
- 序列化方式：Java
- 线程模型：每连接每线程
- JDK代理

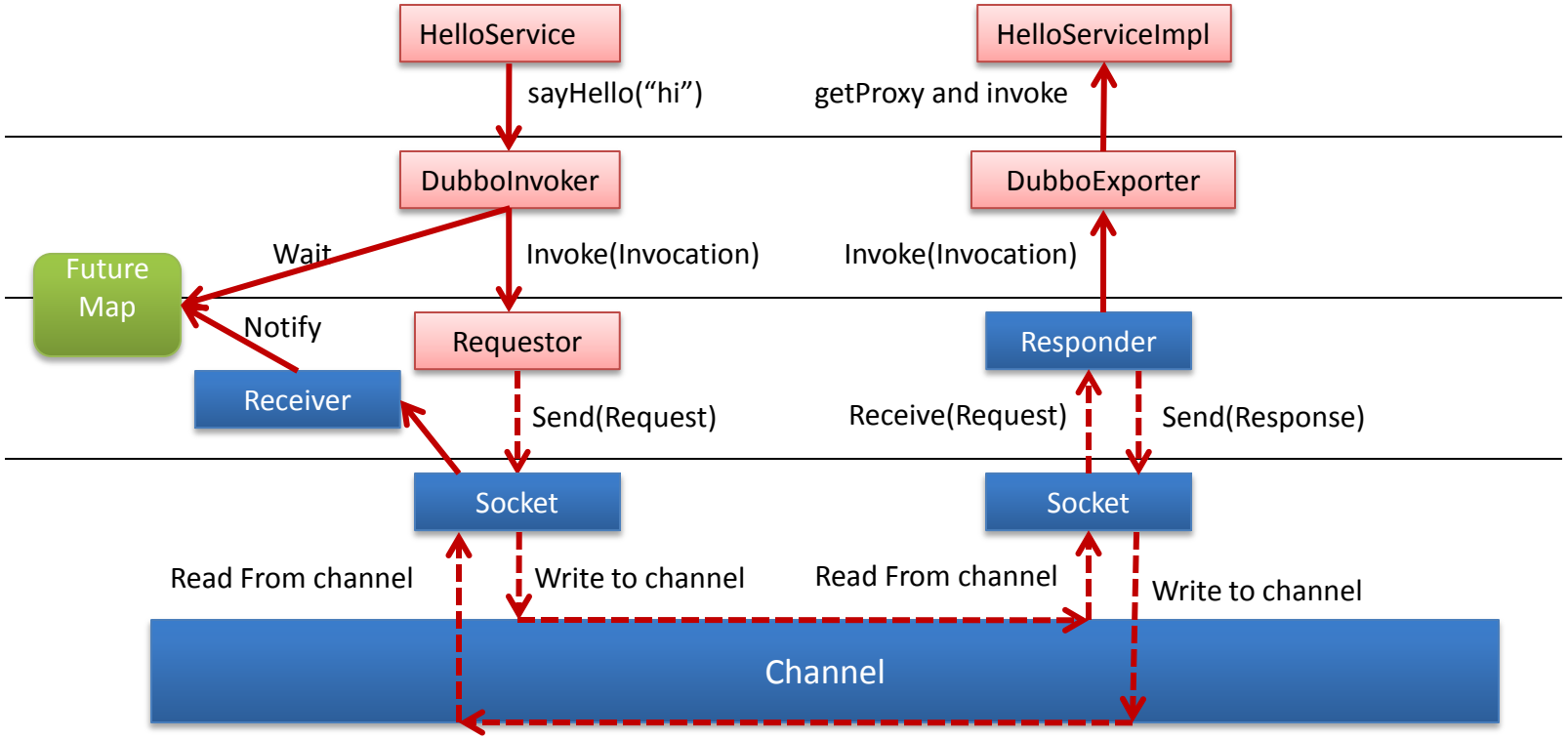
三个主题

- 协议
  - 用什么数据格式进行传输，双方的约定
- 传输
  - 用什么样的通道将数据发送给对方
- 线程
  - 当接收到数据时，如何分发数据进行处理

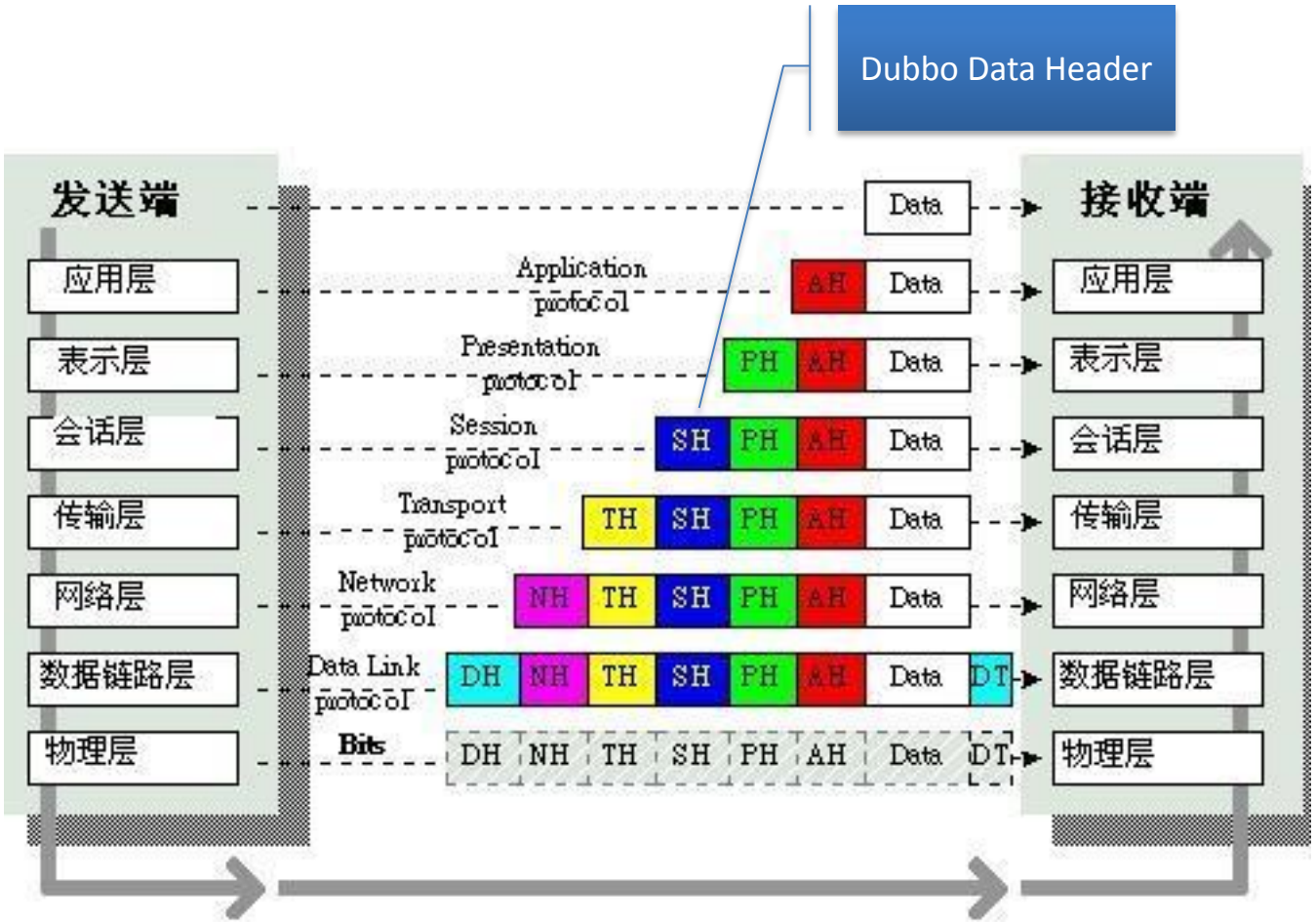
Data Protocol



Data Flow



Append Data Header



此图来自网络

Data Header

IP

Bit offset	0 - 3	4 - 7	8 - 13	14 - 15	16 - 18	19 - 31
0	Version	Header Length	Differentiated Services Code Point	Explicit Congestion Notification	Total Length	
32	Identification				Flags	Fragment Offset
64	Time to Live		Protocol		Header Checksum	
96	Source IP Address					
128	Destination IP Address					
160	Options (if Header Length > 5)					

TCP

Bit offset	0 - 3	4 - 7	8	9	10	11	12	13	14	15	16 - 23	24 - 31
0	Source port										Destination port	
32	Sequence number											
64	Acknowledgment number (if ACK set)											
96	Data offset	Reserved	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size	
128	Checksum										Urgent pointer (if URG set)	
160	Options (if Data Offset > 5)											padding

Dubbo

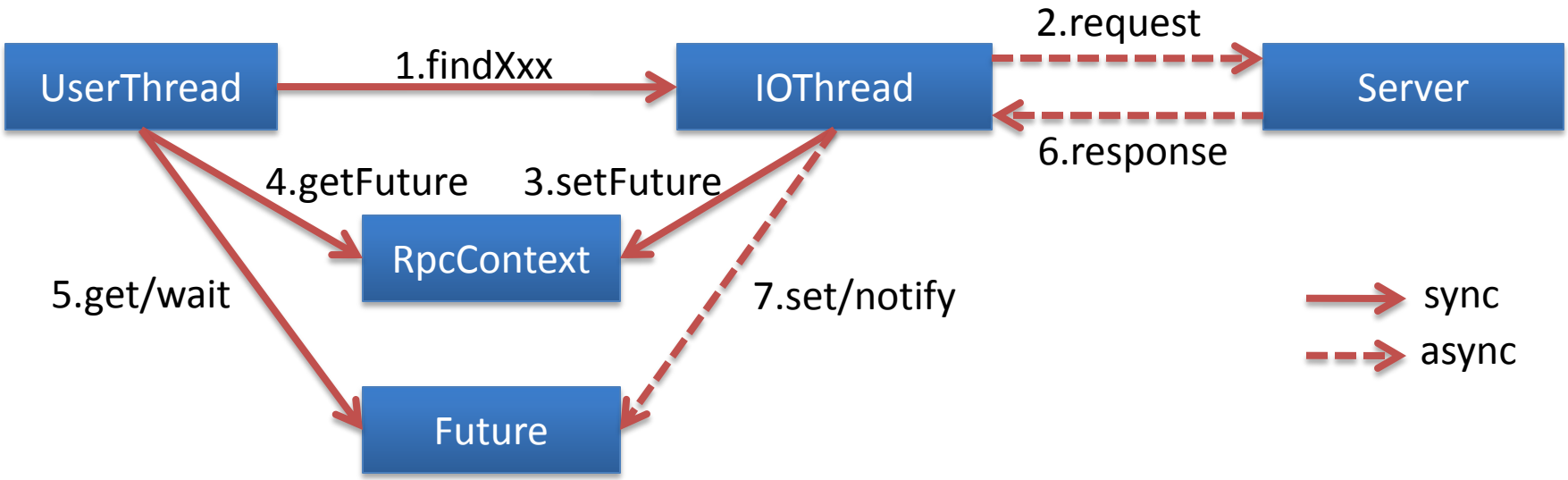
Bit offset	0 - 7	8-15	16-20	21	22	23	24-31
0	Magic High	Magic Low	Serialization id	event	Two way	Req /res	status
32 - 95	Id (long)						
96-127	data length						

Data Header

- Port Unification - Magic Code
  - 多协议支持
  - Telnet
  - 兼容
- Long Id VS Id 轮转
  - 同步转异步
  - 过期策略 + id轮转
  - 扩展header
    - Response Header中增加服务器状态
    - 拥塞控制

Data Header – 同步转异步

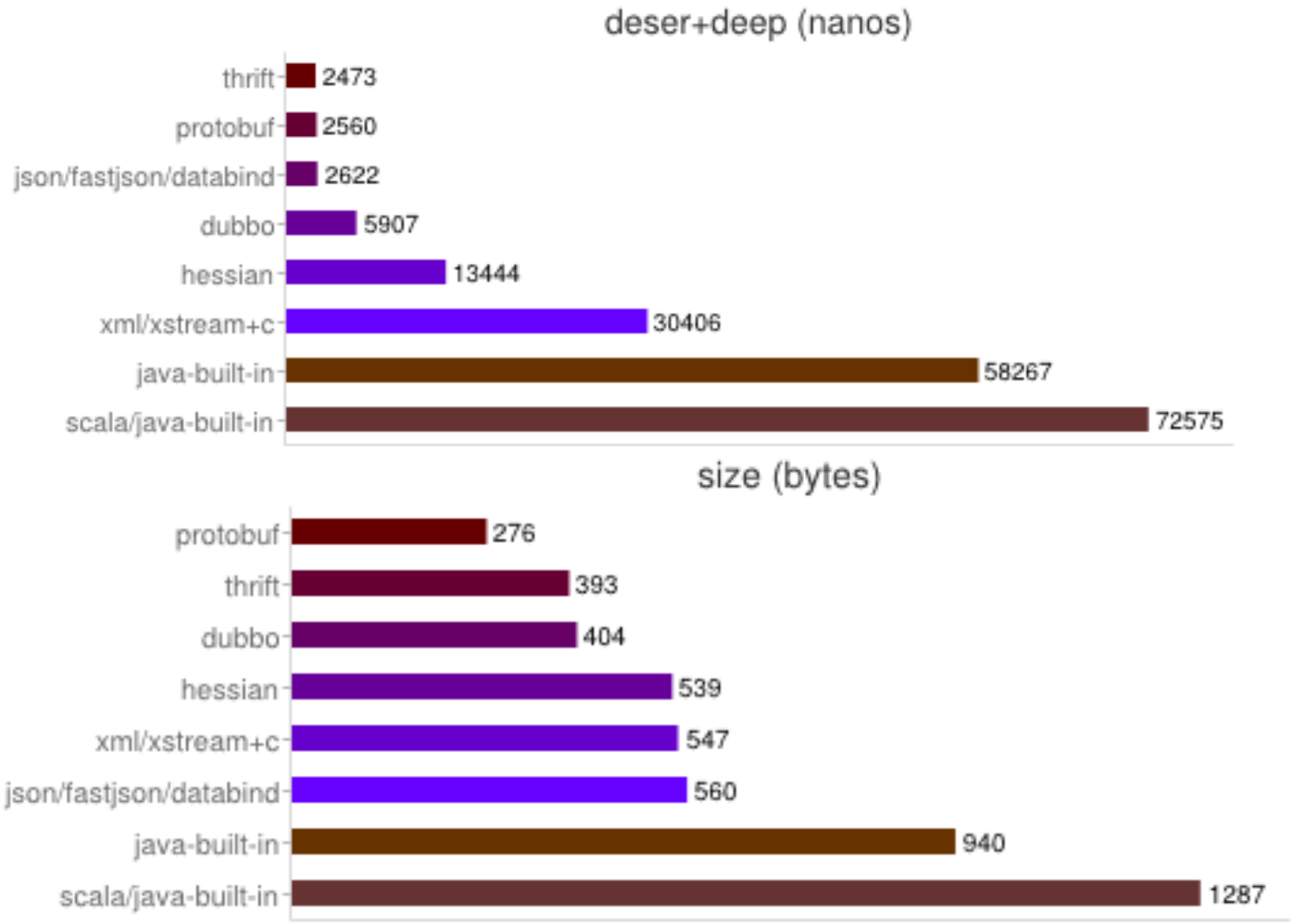
- 并行发起多个请求，但只使用一个线程
  - xxxService.findXxx();
  - Future<Xxx> xxxFuture = RpcContext.getFuture();



Body Serialization

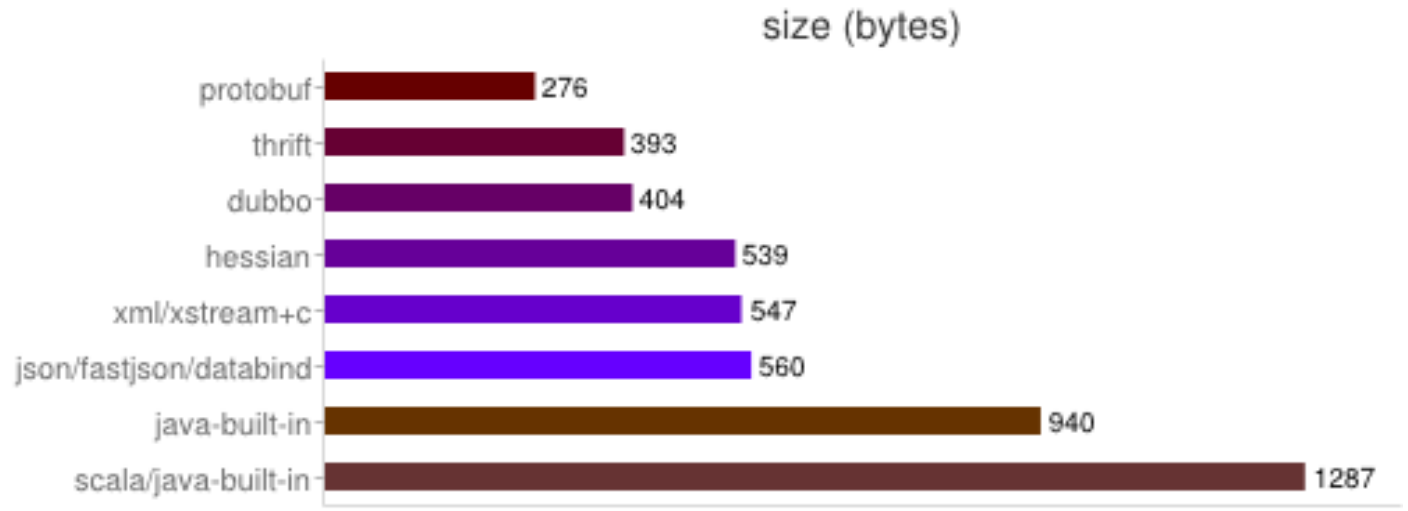
- 几个关键因素
  - 数据大小（传输速度）
  - 序列化&反序列化速度（CPU资源）
  - 兼容性&易用性

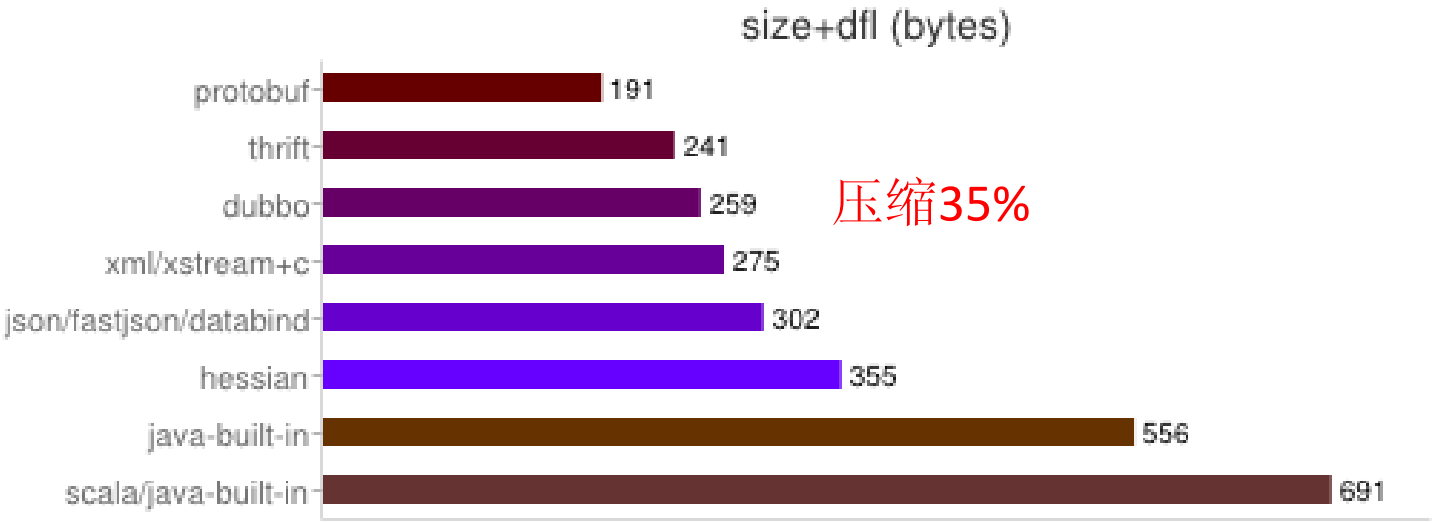
Serialization Performance



Tool：https://github.com/eishay/jvm-serializers/wiki

Serialization Performance





压缩35%

Body Serialization

- 远程服务调用时间主要消耗
  - 网络开销 -> 数据包大小
  - 生产环境某应用容量测试：
    - 9台服务器压力转到1台后（Cpu 3% 网络 90%）
- Dubbo序列化主要优化目标
  - 减少数据包大小（主要）
  - 提高序列化反序列化性能

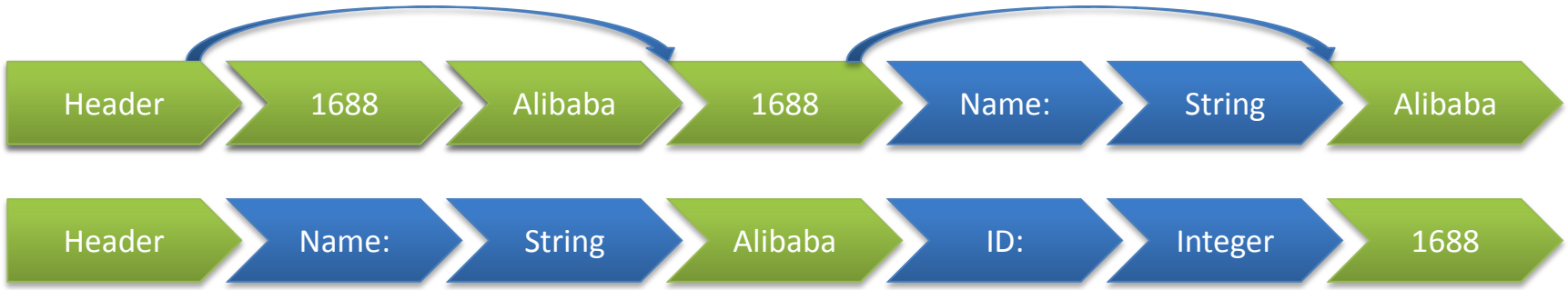
Serialization Optimization

- Special basic type
- Multiplex flag
- Chunk stream??
- Ordered vs. Named
- Contract vs. Self-description
- Poison vs. Exception

Serialization

- Named
- Self-description

- Ordered
- Contract



IO模型

IO模型

- IO Model
- NIO
- TCP选项
- IRQ

IO模型

IO请求划分两个阶段：

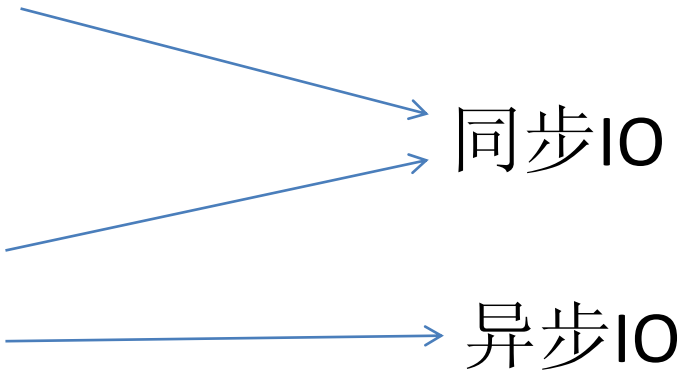
- 等待数据就绪
- 从内核缓冲区拷贝到进程缓冲区

按照请求是否阻塞

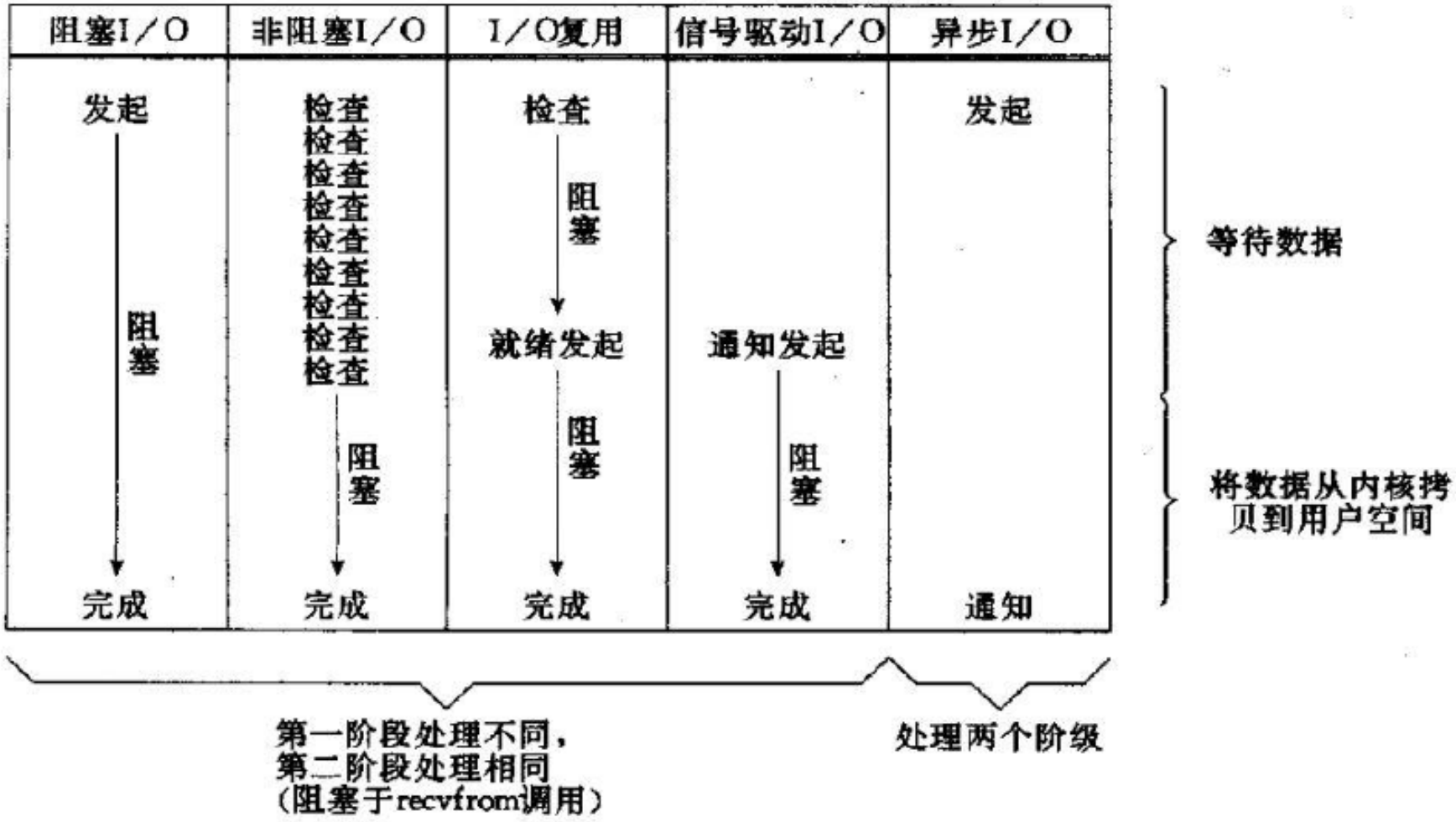
- 同步IO
- 异步IO

Unix的5种IO模型

- Blocking I/O
- Non-Blocking I/O
- I/O Multiplexing
- Signal-Driven I/O
- Asynchronous I/O



IO模型之间的区别



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

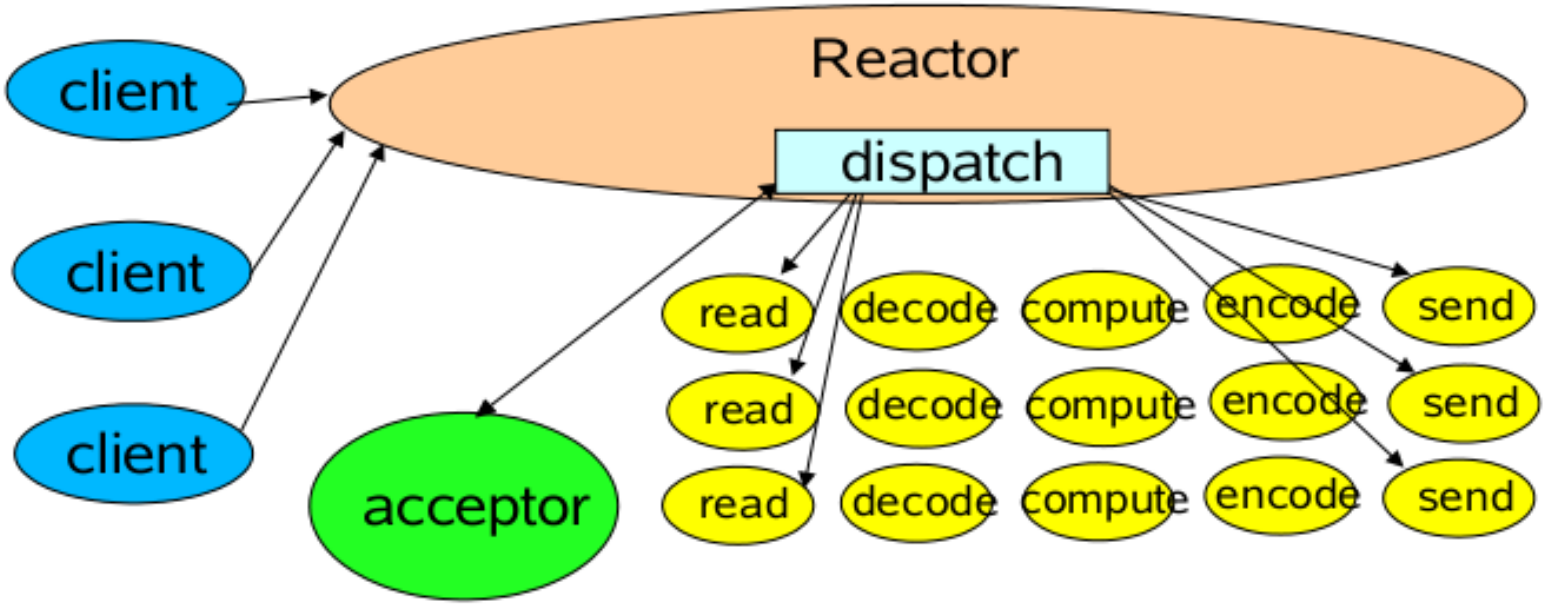
更高级的IO函数，zero-copy

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 - 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 Handle
    - 事件处理,业务代码
- Mina Netty 都是此模式的实现

Reactor示意图



线程模型中详细讲解

NIO优化 - TCP选项

- 合理设置TCP/IP在某些时候可以起到显著的效果

SO_RCVBUF和SO_SNDBUF

- Socket缓冲区大小至少应该是连接的MSS的四倍，

  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
- 在以太网上，4k通常是不够的，增加到16K，吞估量增加了40%。
- 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。
- 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

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= 带宽x RTT

Nagle算法：SO_TCPNODELAY

- NAGLE算法通过将缓冲区内的小封包自动相连，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
- Socket. setTcpNoDelay(true);
  实时性要求较高的应用(telnet 、服务调用),需要关闭该算法，否则响应时间会受影响。

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

TCP心跳间隔是全局设置，建议在应用层做心跳

中断

```
#cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
256: 2278111556          0          0          0          0          0          0          0    Dynamic-irq  timer0
257:    1593006          0          0          0          0          0          0          0    Dynamic-irq  resched0
258:         30          0          0          0          0          0          0          0    Dynamic-irq  callfunc0
259:        680          0          0          0          0          0          0          0    Dynamic-irq  xenbus
260:          0     776074          0          0          0          0          0          0    Dynamic-irq  resched1
261:          0         90          0          0          0          0          0          0    Dynamic-irq  callfunc1
262:          0  460787536          0          0          0          0          0          0    Dynamic-irq  timer1
263:          0          0     784517          0          0          0          0          0    Dynamic-irq  resched2
264:          0          0         92          0          0          0          0          0    Dynamic-irq  callfunc2
265:          0          0  610961913          0          0          0          0          0    Dynamic-irq  timer2
266:          0          0          0     945497          0          0          0          0    Dynamic-irq  resched3
267:          0          0          0         92          0          0          0          0    Dynamic-irq  callfunc3
268:          0          0          0  705281800          0          0          0          0    Dynamic-irq  timer3
269:          0          0          0          0     729973          0          0          0    Dynamic-irq  resched4
270:          0          0          0          0         92          0          0          0    Dynamic-irq  callfunc4
271:          0          0          0          0  612159865          0          0          0    Dynamic-irq  timer4
272:          0          0          0          0          0     747793          0          0    Dynamic-irq  resched5
273:          0          0          0          0          0         91          0          0    Dynamic-irq  callfunc5
274:          0          0          0          0          0  739413487          0          0    Dynamic-irq  timer5
275:          0          0          0          0          0          0     705970          0    Dynamic-irq  resched6
276:          0          0          0          0          0          0         90          0    Dynamic-irq  callfunc6
277:          0          0          0          0          0          0  543901390          0    Dynamic-irq  timer6
278:          0          0          0          0          0          0          0     707045    Dynamic-irq  resched7
279:          0          0          0          0          0          0          0         67    Dynamic-irq  callfunc7
280:          0          0          0          0          0          0          0  499288362    Dynamic-irq  timer7
281:         35          0          0          0          0          0          0          0    Dynamic-irq  xencons
282:       3138          0          0          0          0          0          0          0    Dynamic-irq  xenfb
283:          0          0          0          0          0          0          0          0    Dynamic-irq  xenkbd
284:    2601367        606          0       2611          0          0          0          0    Dynamic-irq  blkif
285:     112133       4153          0          0         60          0          0          0    Dynamic-irq  blkif
286:         19          0          0          0          0          0          0          0    Dynamic-irq  blkif
287:   14259441          0          0          0          0          0          0          0    Dynamic-irq  eth0


#cat /proc/irq/287/smp_affinity
00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000001
```

软中断

- 软中断此时都是运行在在硬件中断相应的cpu上。
- 如果始终是cpu0相应网卡的硬件中断，那么始终都是cpu0在处理软中断，而此时cpu1就被浪费了，因为无法并行的执行多个软中断

Linux内核优化- RPS

- 网卡数据包处理优化
  - Linux 内核开启 RPS，小包处理提升15+%

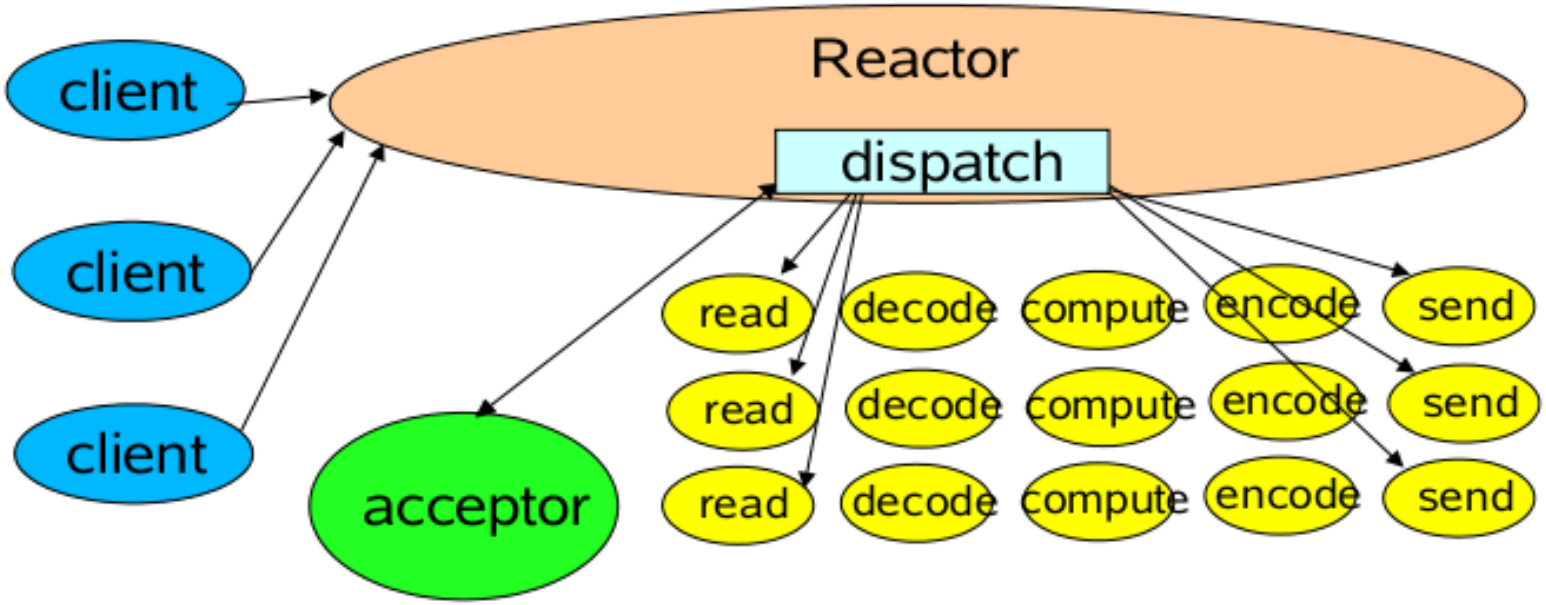
    RPS 这项技术将流入的数据包分布给所有可用的 CPU 去处理。
  - 使用Linux Kernel 2.6.35开启该功能Dubbo的TPS提高3倍
- RPS (Receive Packet Steering) 基本原理
  - 根据数据包的源地址，目的地址以及目的和源端口，计算出一个hash值，然后根据这个hash值来选择软中断运行的cpu，
  - 从上层来看，也就是说将每个连接和cpu绑定，并通过这个hash值，来均衡软中断在多个cpu上

线程模型

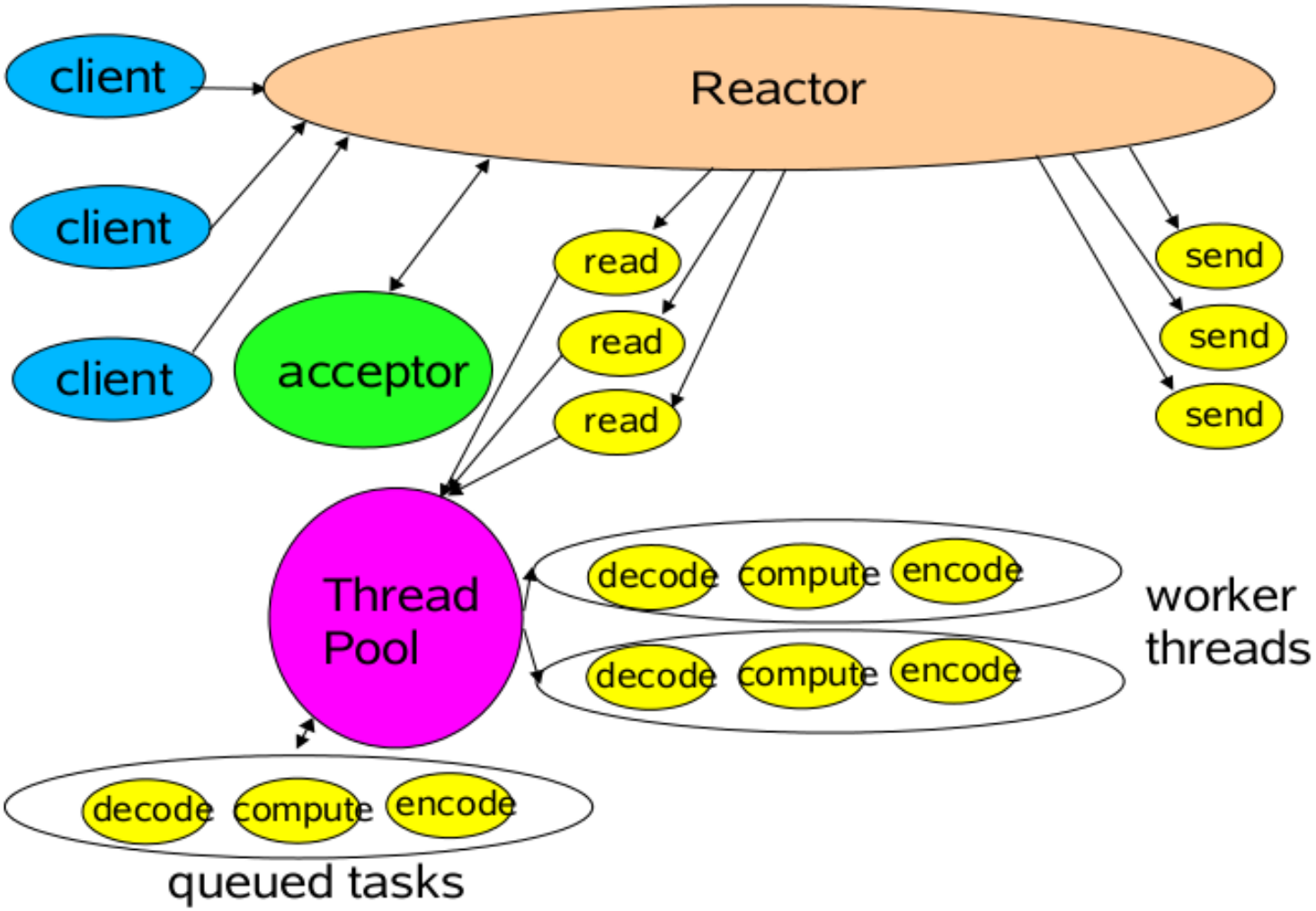
线程模型内容

- Reactor线程模型
- 序列化线程
- 业务线程派发策略

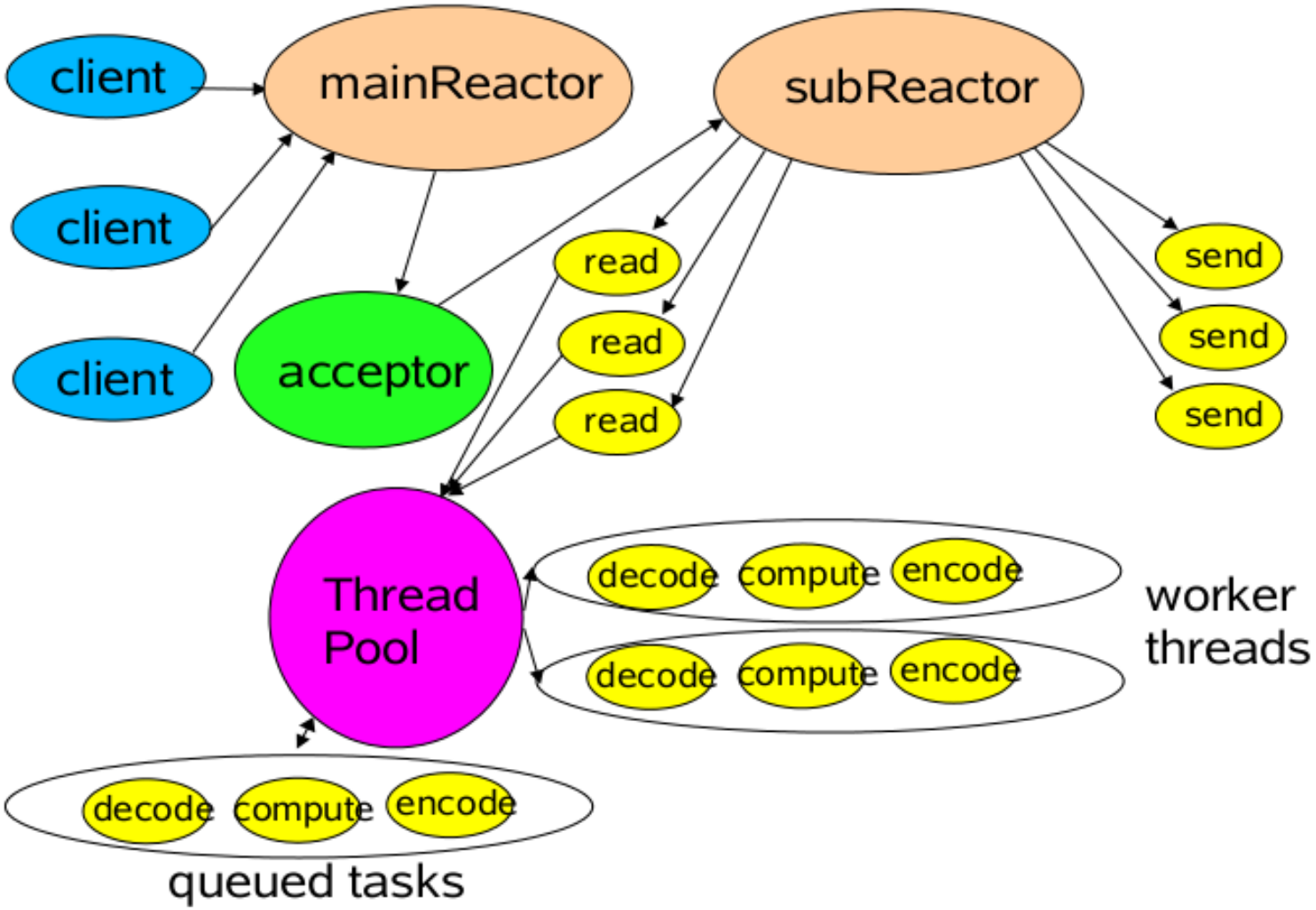
Reactor单线程模型



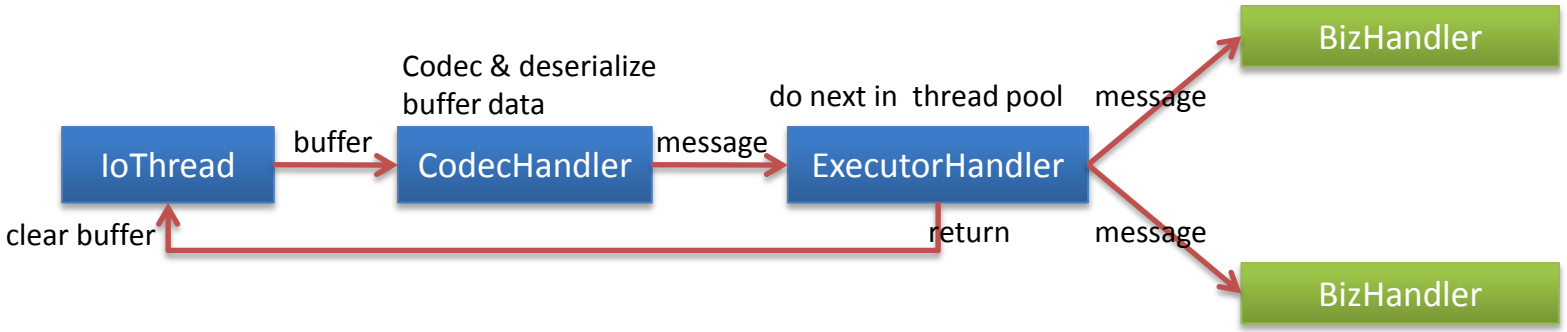
Reactor多线程模型



Reactor多线程模型

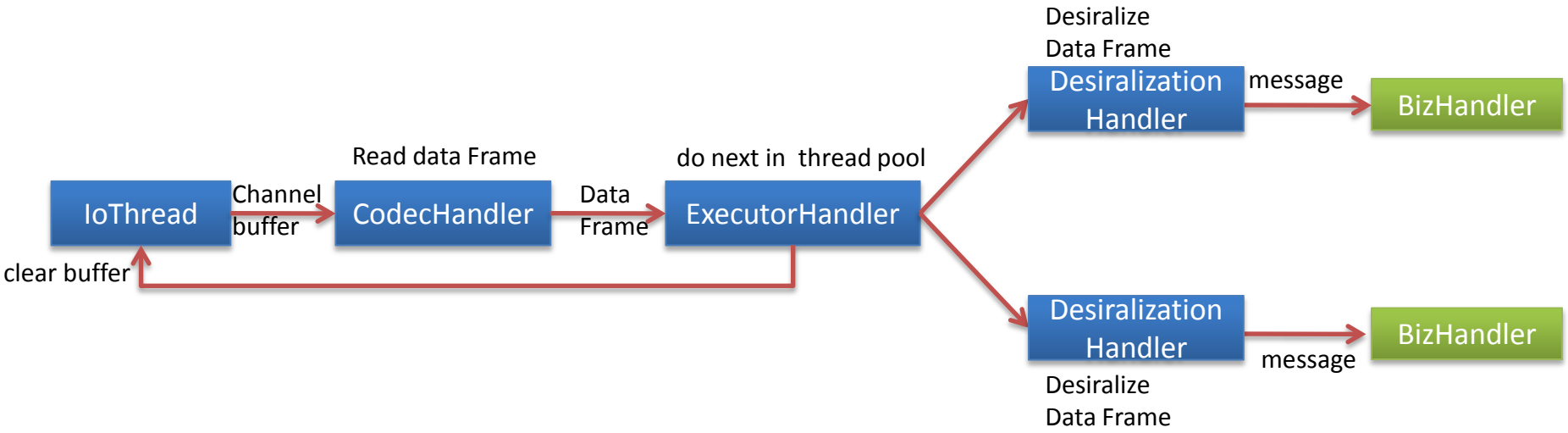


IO线程处理序列化



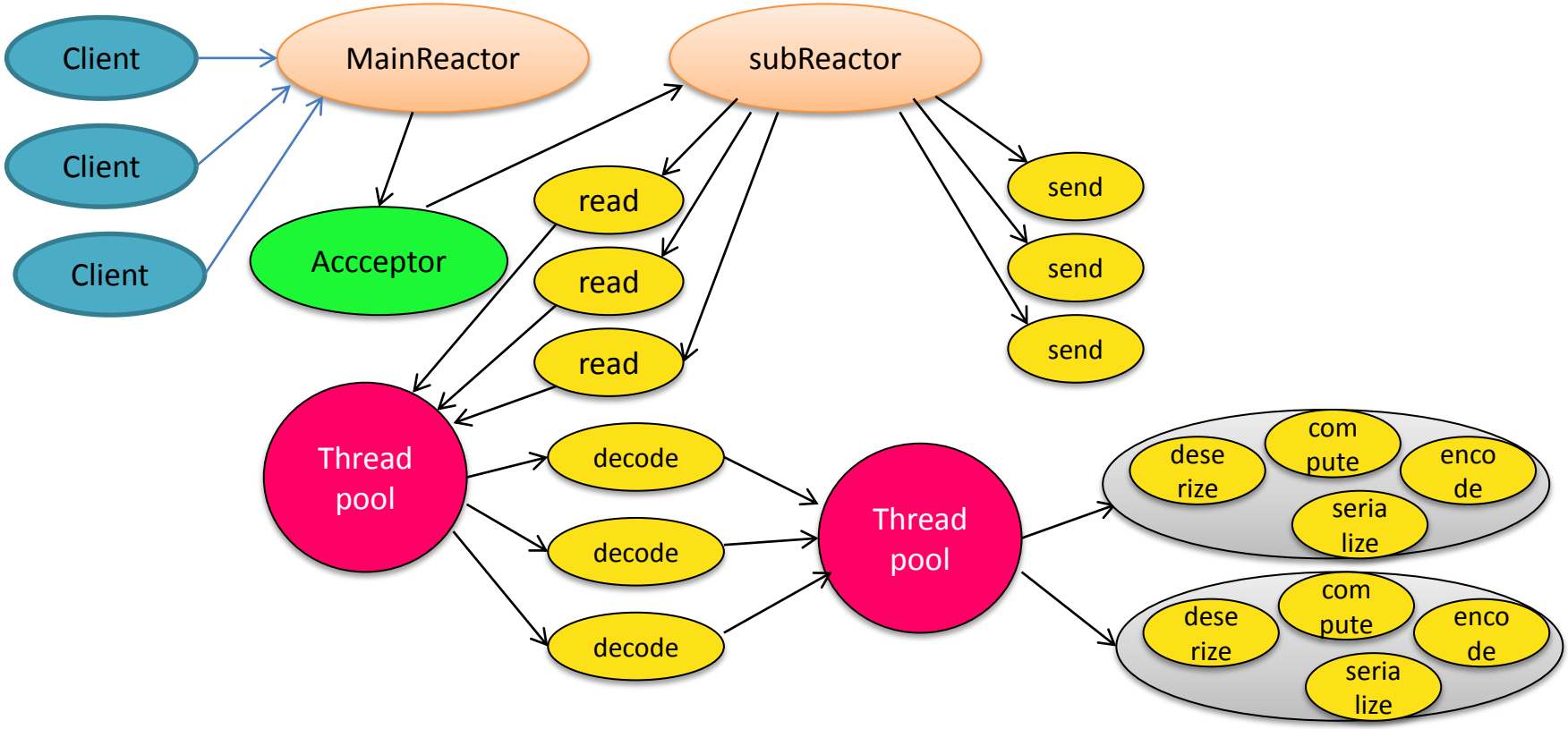
- 在IO线程处理反序列化问题是?
- 可以直接在业务线程处理么？
- 可以改进么？

业务线程处理序列化



Codec 可以放到业务线程池处理么？有必要么？

Reactor多线程模型



线程派发策略

- 5个事件
  - Connect (连接建立)
  - Disconnect (连接断开）
  - MessageReceived (消息已接受)
  - Sent(消息已发送)
  - Exception Caught(异常)
- 派发策略
  - 5个事件共享同一个线程池
  - Connect disconnect 使用独立线程池(size为1)
  - 全部不派发线程池，IO线程处理

Thread Pool Size

- 设置多少线程合适？

有这样一个模块

cpu计算时间18ms（running）

查询数据库，网络io时间80ms（waiting）

解析结果2ms如果服务器2CPU，大家看看这里多少线程合适

充分利用cpu资源：
线程数量=100/20*2=10

Thread Pool Size

- 从CPU角度而言

  线程数量=（cpu时间+cpu等待时间）/cpu时间*cpu数量

- 线程数量的设置就是由CPU决定的？

  有这样一个模块：

  线程同步锁(数据库事务锁)50ms

  cpu时间18ms

  查询数据库，网络io时间80ms

  解析结果2ms 如果服务器有2个CPU，这个模块线程多少合适？

Thread Pool Size

- CPU计算为瓶颈，计算线程数量

  线程数=(18 + 2 + 50 + 80) / 20 * 2 = 15
- 以线程同步锁为瓶颈，计算线程数

  线程数=(50 + 18 + 2 + 80) / 50 * 1/1 = 3

Thread Pool Size

公式一：

线程数量=（线程总时间/瓶颈资源时间）* 瓶颈资源的线程并行数

准确的讲

瓶颈资源的线程并行数=瓶颈资源的总份数/单次请求占用瓶颈资源的份数

约束：

在计算的时候，对同一类资源的消耗时间进行合并

公式二：

QPS=1000/线程总时间*线程数

注意：

如果线程数不够，则QPS减少。

线程本身也要消耗资源，如果线程太多，同样QPS会下降。

其他优化

- 其他优化
  - Lock free data structure
  - Buffer copy (Zero Copy)
  - JVM GC tuning
  - Context Switch
  - 同步转异步
  - JavassistProxy改进JDK proxy
- 注意性能的短板效应，避免过度优化
  - 优化的代价，通常是牺牲未来的可能性。

总结

- 数据协议
  - 同步转异步
  - 高效序列化
- NIO
  - TCP属性设置
  - Linux内核开启RPS
- 线程模型
  - IO线程池与业务线程池的隔离
  - 线程池计算方法
- 其他优化

参考

- 《Linux高级网络编程》
- 《NIO trcik and trip》-伯岩
- 《淘宝前台系统优化实践》- 蒋江伟
- 《 Receive packet steering patch详解》

Q & A

Q & A

End

Thanks