高质量Java编程实践 v4

一粟 yisu@taobao.com

代码质量标准

- 可读性
- 可维护性
- 性能

可读性

提高代码可读性

- 命名规范
  - Camel风格：　lastName
  - 匈牙利命名法：sLastName
  - Pascal命名法：　LastName
  - PHP命名法：　last_name

提高代码可读性

- 一致的代码风格
  - 缩进
  - 括号
  - 注释
    - 复杂的逻辑必须注释
    - 明显的逻辑不必注释

提高代码可读性

- 代码需要避免
  - 大函数
  - 大循环
  - 深嵌套
  - 重复代码

可维护性

可维护性

- 软件生命周期
- 可重用，可扩展
- KISS原则 -- Keep it Simple and Stupid
- Best Practises
- 设计模式
- 开发框架
- 公共库
- 单元测试

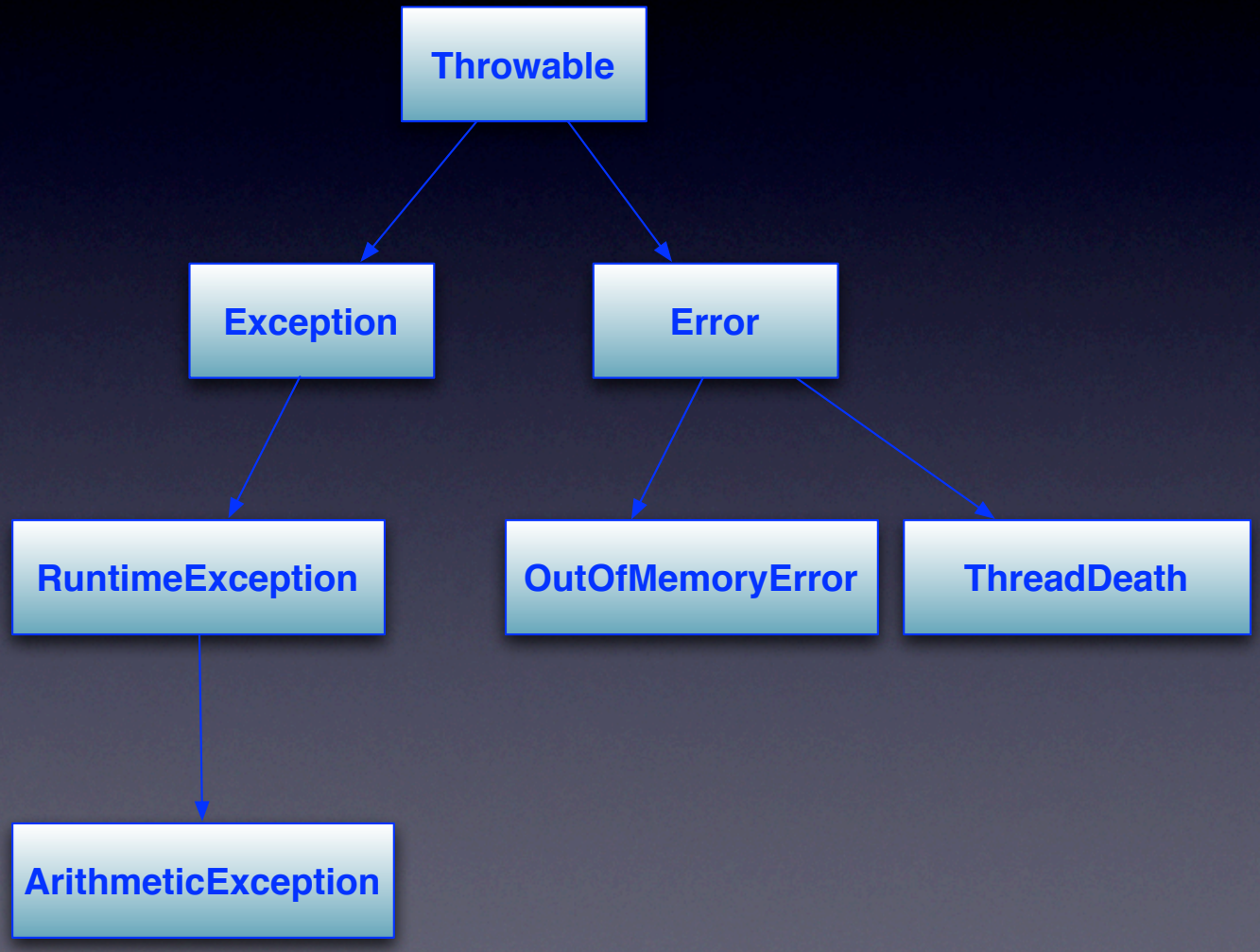
案例

```
1 String s = "0";
2 if (s.equals(0)) {
3     System.out.println("yes");
4 }
5 else {
6     System.out.println("no");
7 }
```

异常处理

```
1 try {
2     ...
3 }
4 catch (IOException e) {
5     ...
6 }
7 catch (SQLException e2) {
8     ...
9 }
10 finally {
11     ...
12 }
```

异常处理


Throwable
Exception
Error
RuntimeException
OutOfMemoryError
ThreadDeath
ArithmeticException

异常处理

- 避免
  - 丢弃异常或不处理又抛出
  - 异常处理忘记回收资源
  - 不指定具体异常
  - 过于庞大的try块
  - 不处理自己能处理的异常!

异常处理

- 提倡
  - 提供完整的异常信息
  - 完善带异常日志可帮助分析问题
    - 日志分类与级别
    - java.util.logging, log4j or slf4j
  - 尽量不往外抛异常

案例

```
1 public class ExceptionTest {
2     public static void main(String[] args) {
3         try {
4             System.out.println(1/0);
5         }
6         finally {
7             return;
8         }
9     }
10 }
11
```

提高方法

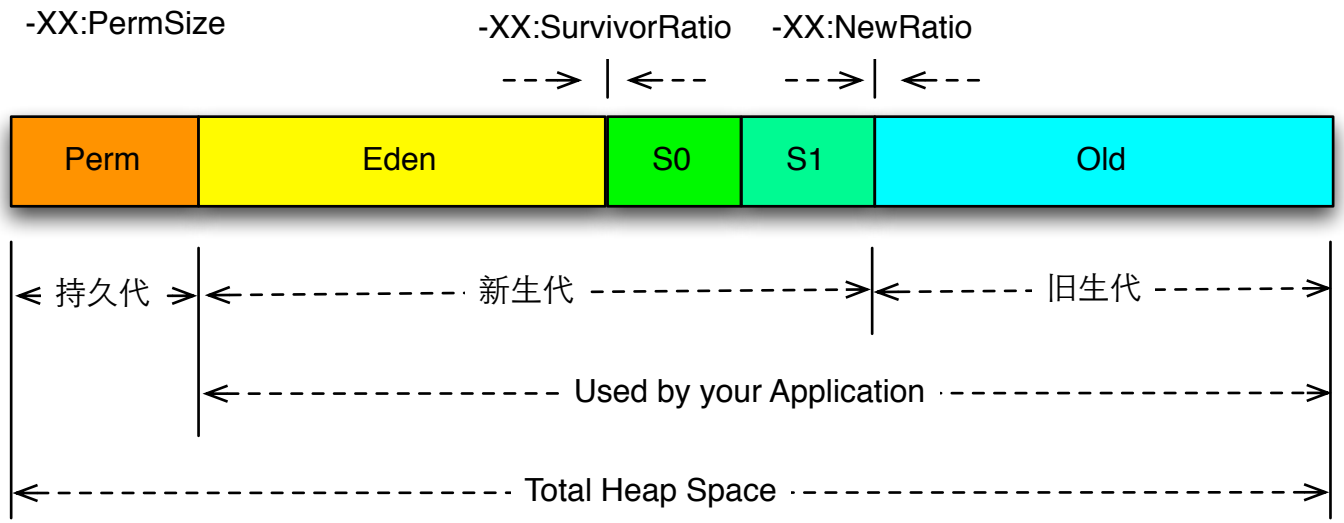
- 《Thinking in Java》
- 《Effective Java》
- Code Review
- 代码重构
- 阅读JDK源代码
- 阅读开源软件代码

性能

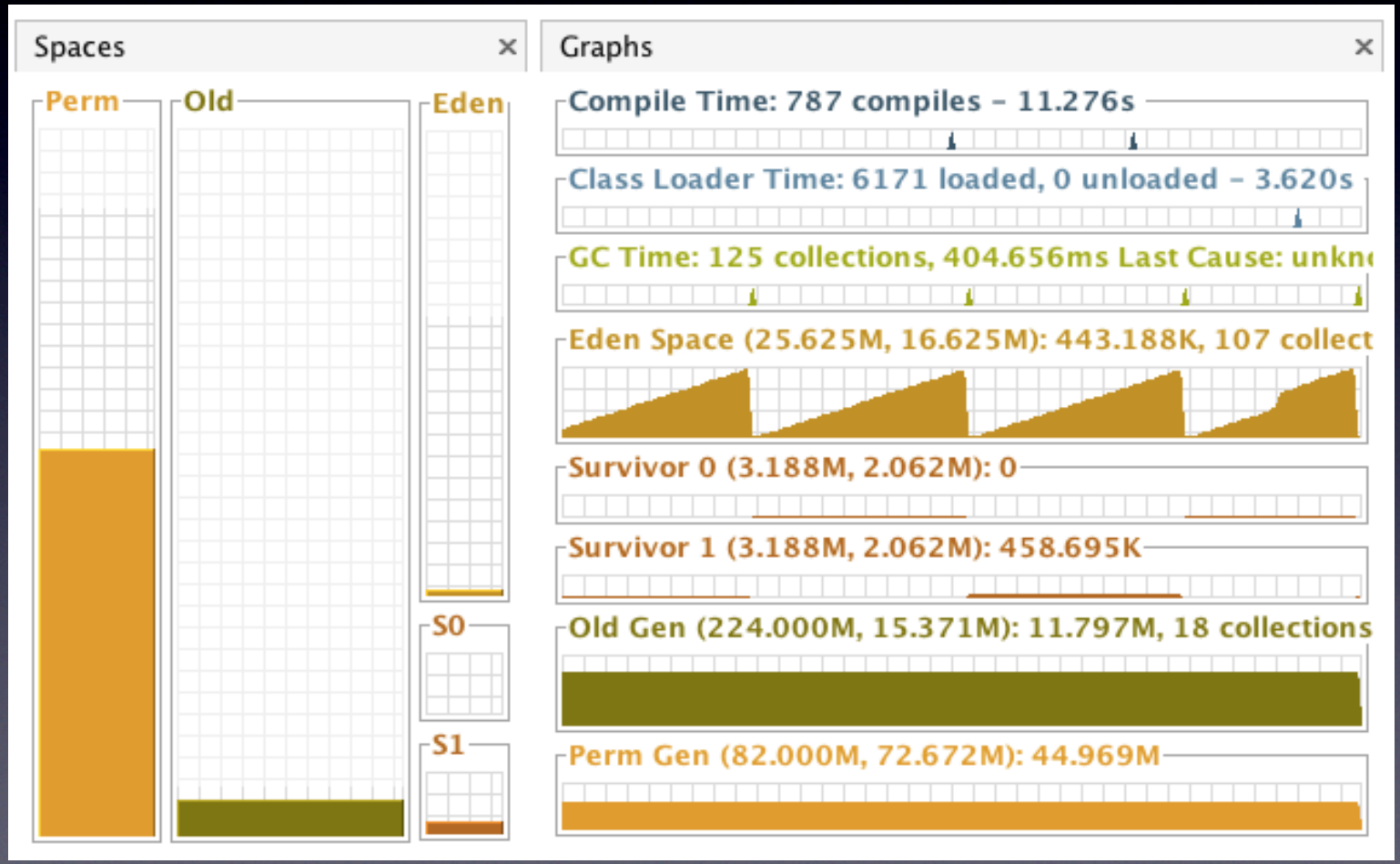
性能

- Memory
- CPU
- I/O
  - Disk
  - Network

内存管理


-XX:PermSize
-XX:SurvivorRatio
-XX:NewRatio
Perm
Eden
S0
S1
Old
持久代
新生代
旧生代
Used by your Application
Total Heap Space

Visual GC


Spaces
Graphs
Perm
Old
Eden
S0
S1
Compile Time: 787 compiles - 11.276s
Class Loader Time: 6171 loaded, 0 unloaded - 3.620s
GC Time: 125 collections, 404.656ms Last Cause: unkn
Eden Space (25.625M, 16.625M): 443.188K, 107 collect
Survivor 0 (3.188M, 2.062M): 0
Survivor 1 (3.188M, 2.062M): 458.695K
Old Gen (224.000M, 15.371M): 11.797M, 18 collections
Perm Gen (82.000M, 72.672M): 44.969M

32位 or 64位 JVM

- 32位 JVM
  - 堆的大小限制: 3G
- 64位 JVM
  - 每个对象会占用更大的内存
  - 压缩普通对象指针优化: -XX:+UseCompressedOops

内存管理

- OutOfMemoryError
- StackOverflowException
- 尽量节省内存开销
  - Object Header and Padding
- 尽量减少Full GC
  - GC参数调节
- 尽量避免大块数据在内存里移动

内存泄漏

- class loader内存泄漏
- 创建数组不检查大小
- ThreadLocal变量不释放
- 静态hash做缓存只增不减
- 资源（数据库连接）不关闭
- 将对象引用设为null是否对GC有帮助?

案例

```
1 final static int SIZE = 1024*1024*10;
```

```
1 int[] arr = new int[SIZE];
2 for (int i=0;i<SIZE;i++) {
3     arr[i] = i;
4 }
```

```
1 Integer[] arr = new Integer[SIZE];
2 for (int i=0;i<SIZE;i++) {
3     arr[i] = i;
4 }
```

JVM缓存

- static constants
- WeakReference
- ThreadLocal
- DirectByteBuffer & MappedByteBuffer
- JNI / GC invisible Heap *

性能衡量指标

- 内存开销
- 响应时间
- 吞吐量

性能测试方法

- 预热
  - -XX:+PrintCompilation
- 消除GC的影响
- 正确的统计方法
- 计时方法
- 注意测试代码本身的影响

性能优化方法

- 建立测试基准指标
- 确定优化方向，固定其它指标测试
- 工具
  - git版本控制
  - Java Profiler: VisualVM, JProfiler, YourKit
  - http_load / ab
  - vmstat

CPU

- 多核CPU （多进程 vs 多线程）
- JIT Compiler
- 线程切换开销
- 线程池大小最佳值
  - 解析XML/数据压缩或解压
  - 文件日志
  - 网络调用

线程安全

有状态(Stateful)	无状态(Stateless)
共享数据可变	共享数据不变
复杂	简单
变量可能线程不安全	变量线程安全
不容易水平扩展	容易水平扩展

并发控制方法

- 同步 synchronization
- wait(), notify() and notifyAll()
- volatile
- 原子性 Atomic
- 锁 Lock
- CAS (compare-and-swap)
- java.util.concurrent
- STM (Software Transactional Memory)

多线程误区

- 在构造函数中启动线程?
- 两个方法A和B都是线程安全，组合调用A()和B()是否还是线程安全?
- 遍历Map时候是否可以删除某个键?
- 线程被中断的异常处理?

案例

```
1  public class CounterServlet implements Servlet {
2      private int count = 0;
3
4      public int getCount() { return count; }
5
6      public void service(ServletRequest req,
                           ServletResponse res) {
7          ++count;
8      }
9  }
10
```

案例

```
1  public class CounterServlet implements Servlet {
2      private int count = 0;
3
4      public synchronized int getCount() { return count; }
5
6      public synchronized void service(ServletRequest req,
                                        ServletResponse res) {
7          ++count;
8      }
9  }
10
```

案例

```
1  public class CounterServlet implements Servlet {
2      private volatile int count = 0;
3
4      public int getCount() { return count; }
5
6      public synchronized void service(ServletRequest req,
                                        ServletResponse res) {
7          ++count;
8      }
9  }
10
```

案例

```
1 public class CounterServlet implements Servlet {
2     private AtomicInteger count = new AtomicInteger(0);
3
4     public int getCount() { return count.get(); }
5
6     public void service(ServletRequest req,
                          ServletResponse res) {
7         count.incrementAndGet();
8     }
9 }
10
```

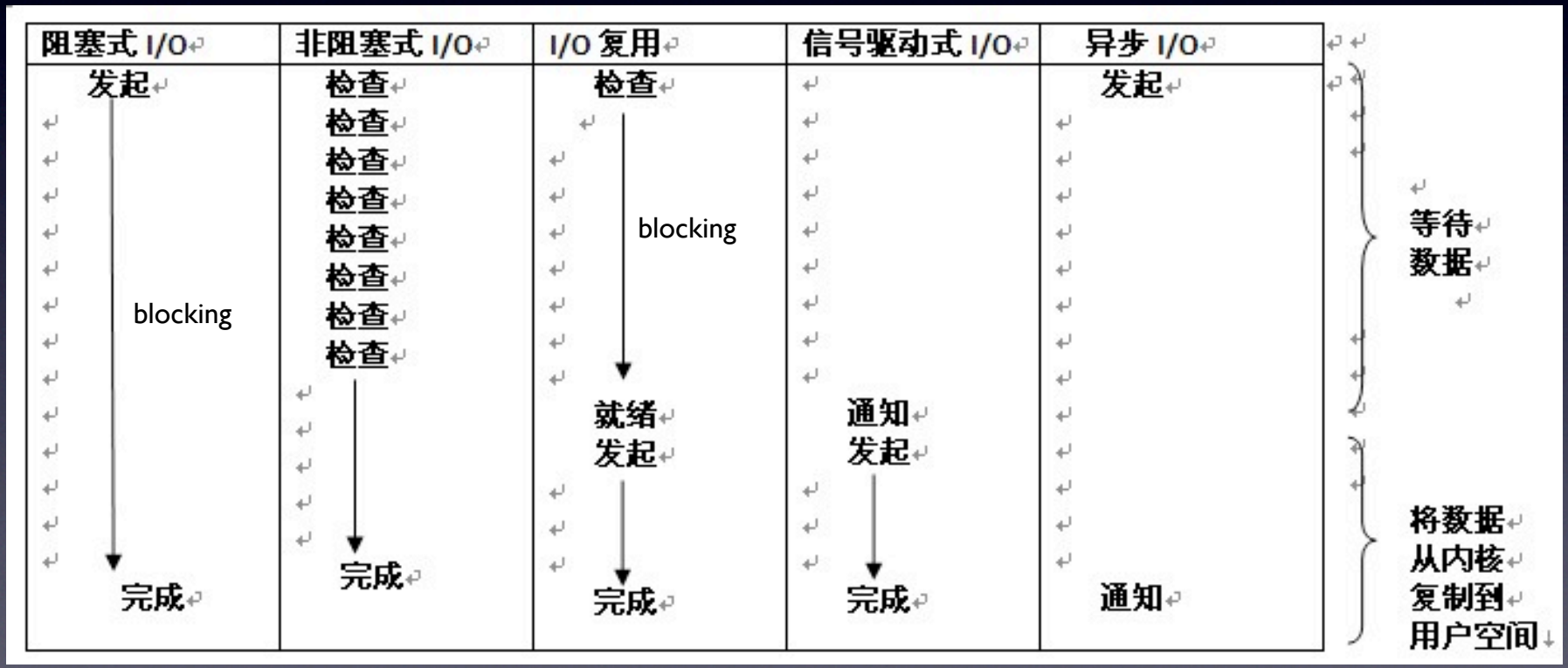
如何提高

- java.util.concurrent.*
- 《Java Concurrency in Practice》

I/O

- CPU 耗时 vs. I/O耗时
  - 计算密集型和I/O密集型应用
- 网络应用一般都带有I/O, 多数情况下，减少I/O可显著提高性能

各种I/O模型


阻塞式 I/O
发起
blocking
完成
非阻塞式 I/O
检查
检查
检查
检查
检查
检查
检查
检查
完成
I/O 复用
检查
blocking
就绪
发起
完成
信号驱动式 I/O
通知
发起
完成
异步 I/O
发起
通知
等待
数据
将数据
从内核
复制到
用户空间

I/O

- 同步（Synchronous) / 异步 (Asynchronous)
- 阻塞(Blocking) / 非阻塞(Non-Blocking)

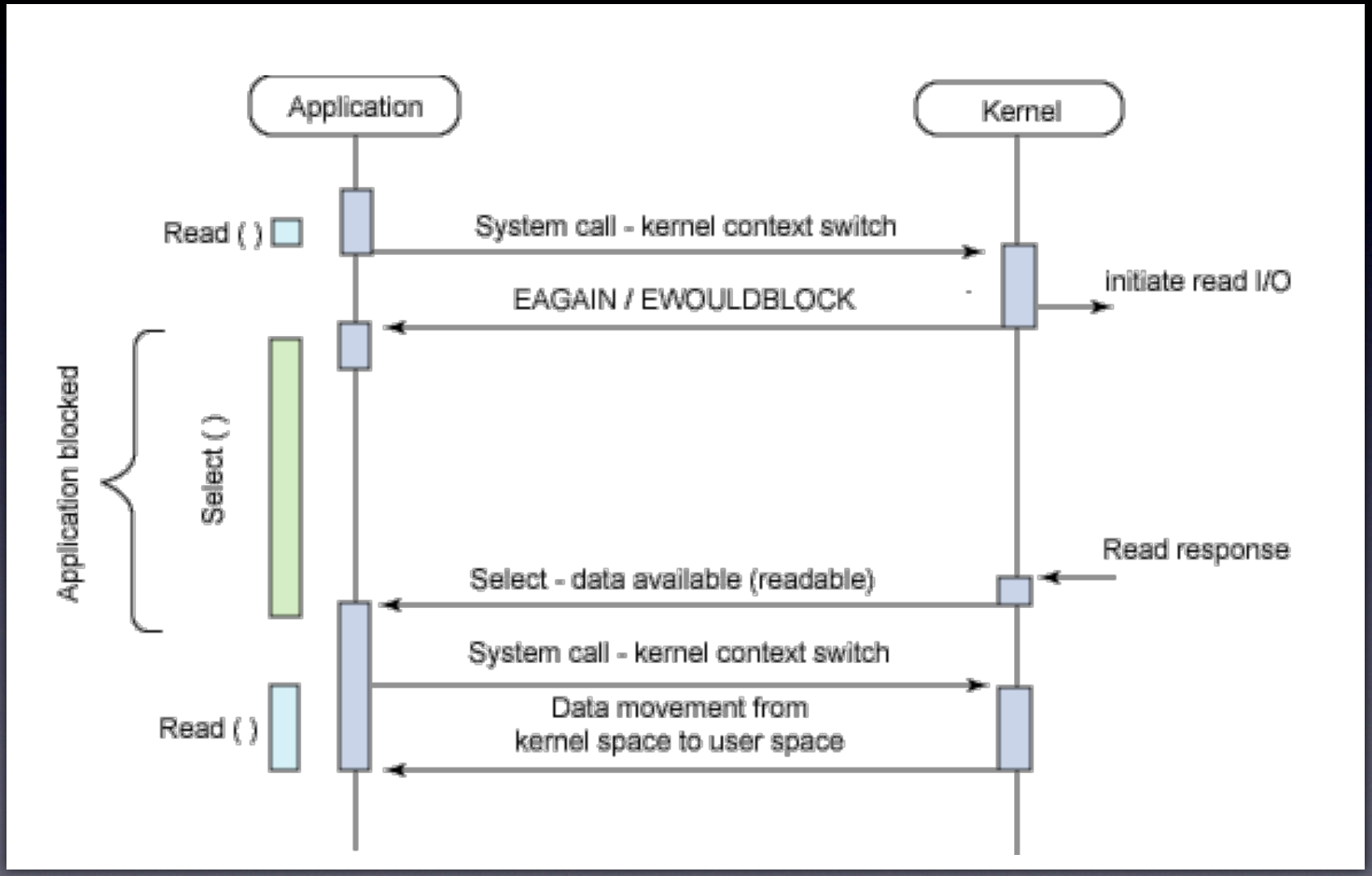
I/O

- java.io.*
  - InputStream/OutputStream
- java.nio.*
  - Buffer/Channel/Selector

Java NIO

- 特点
  - non-blocking socket (异步I/O)
  - select/poll/epoll
  - 一个或少量线程轮询多个连接是否有数据
  - 连接数多的时候性能好
  - 连接数少的时候延迟不如阻塞I/O好

Java NIO


Application
Kernel
Read ( )
System call - kernel context switch
initiate read I/O
EAGAIN / EWOULDBLOCK
Application blocked
Select ( )
Read response
Select - data available (readable)
System call - kernel context switch
Read ( )
Data movement from
kernel space to user space

提高I/O性能

- 尽量批量写数据
- 磁盘I/O
  - 利用好操作系统磁盘缓冲
- 网络I/O
  - 连接管理：长连接或短连接 单个连接或多个连接
  - 大消息可以压缩
  - 消息协议优化

拷贝文件

```java
public static void copyFile(File source, File dest) throws IOException {
    if(!dest.exists()) {
        dest.createNewFile();
    }
    InputStream in = null;
    OutputStream out = null;
    try {
        in = new new FileInputStream(source);
        out = new FileOutputStream(dest);

        // Transfer bytes from in to out
        byte[] buf = new byte[1024];
        int n;
        while ((n = in.read(buf)) > 0) {
            out.write(buf, 0, n);
        }
    }
    finally {
        if(in != null) {
            in.close();
        }
        if(out != null) {
            out.close();
        }
    }
}
```

拷贝文件

```
1 public static void copyFile(File sourceFile, File destFile) throws IOException {
2     if(!destFile.exists()) {
3         destFile.createNewFile();
4     }
5
6     FileChannel source = null;
7     FileChannel destination = null;
8     try {
9         source = new FileInputStream(sourceFile).getChannel();
10        destination = new FileOutputStream(destFile).getChannel();
11        destination.transferFrom(source, 0, source.size());
12    }
13    finally {
14        if(source != null) {
15            source.close();
16        }
17        if(destination != null) {
18            destination.close();
19        }
20    }
21
22 }
```

如何提高

- 生产者/消费者模型实现
- HTTP 协议，简单服务器客户端实现
- O'Reilly《Java NIO》
- Mina / Netty / TBRemoting 网络服务器/客户端框架

总结

总结

- 养成良好的代码编写及自测习惯
- 收集自己的最佳实践并灵活运用
- 熟悉类库的使用及其实现
- 理解了操作系统底层才可以写出最高性能的代码

提问?