# 1 I/O专题

## 1.1 I/O基础知识

所谓IO，就是计算机的输入输出系统，大多IO都会和硬件打交道，比如内存、硬盘、光驱等。由于要操作硬件，IO一般都比较耗时，所以操作系统都支持同步和异步两种IO方式，同步IO就是要等到操作完成才继续执行，而异步IO不需要等待。

* 同步IO操作：IO操作将导致请求进程阻塞，直到IO操作完成。
* 异步IO操作：不会因为IO操作阻塞，IO操作全部完成才通知请求进程

## 1.2 I/O模型

通常操作系统的I/O模式可分为：同步（synchronous） IO和异步（asynchronous） IO。

**同步（synchronous） IO**

阻塞（blocking） IO、非阻塞（non-blocking）IO、多路复用（multiplexing）IO以及信号驱动式IO模型都属于同步IO.

**异步（asynchronous） IO**

windows的iocp是真正的异步IO

## 1.3 Linux环境下的network IO

通常IO发生时涉及的对象和步骤，对于一个network IO (这里我们以read举例)，它会涉及到两个系统对象，一个是调用这个IO的process (or thread)，另一个就是系统内核(kernel)。当一个read操作发生时，它会经历两个阶段：

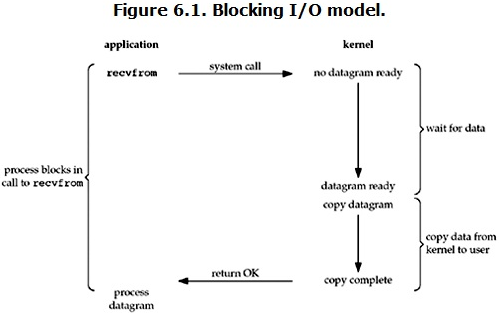
1. 等待数据准备 (Waiting for the data to be ready)

2. 将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)

记住这两点很重要，因为这些I/O Model的区别就是在两个阶段上各有不同的情况。

### 1.3.1阻塞（blocking） IO

在Linux中，默认情况下所有的socket都是blocking，一个典型的读操作流程大概是这样：

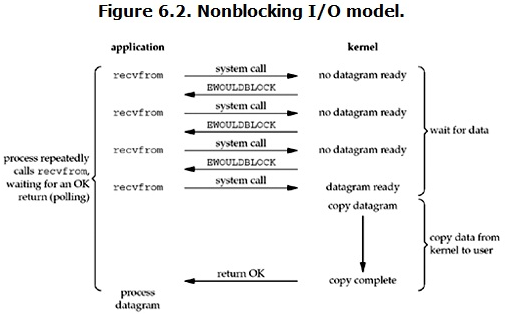


当用户进程调用了recvfrom这个系统调用，kernel就开始了IO的第一个阶段：准备数据。对于network io来说，很多时候数据在一开始还没有到达（比如，还没有收到一个完整的UDP包），这个时候kernel就要等待足够的数据到来。而在用户进程这边，整个进程会被阻塞。当kernel一直等到数据准备好了，它就会将数据从kernel中拷贝到用户内存，然后kernel返回结果，用户进程才解除block的状态，重新运行起来。

所以，blocking IO的特点就是在IO执行的两个阶段都被block了。

### 1.3.2非阻塞（non-blocking）IO

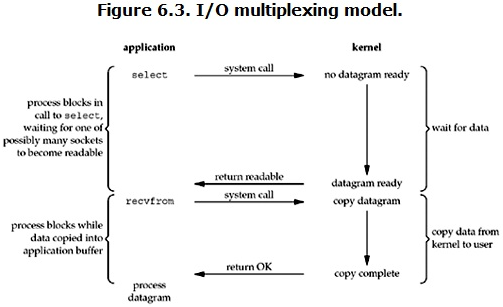
Linux下，可以通过设置socket使其变为non-blocking。当对一个non-blocking socket执行读操作时，流程是这个样子：



从图中可以看出，当用户进程发出read操作时，如果kernel中的数据还没有准备好，那么它并不会block用户进程，而是立刻返回一个error。从用户进程角度讲 ，它发起一个read操作后，并不需要等待，而是马上就得到了一个结果。用户进程判断结果是一个error时，它就知道数据还没有准备好，于是它可以再次发送read操作。一旦kernel中的数据准备好了，并且又再次收到了用户进程的system call，那么它马上就将数据拷贝到了用户内存，然后返回。所以，用户进程其实是需要不断的主动询问kernel数据好了没有。

### 1.3.3多路复用（multiplexing）IO

IO multiplexing这个词可能有点陌生，但是如果我说select，epoll，大概就都能明白了。有些地方也称这种IO方式为event driven IO。我们都知道，select/epoll的好处就在于单个process就可以同时处理多个网络连接的IO。它的基本原理就是select/epoll这个function会不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。它的流程如图：



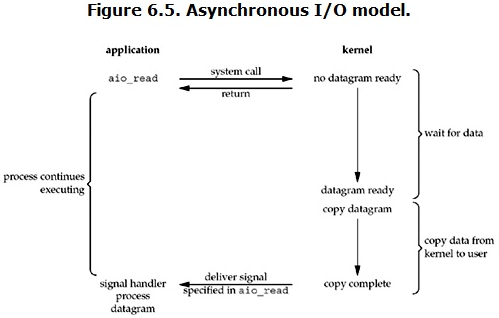
当用户进程调用了select，那么整个进程会被block，而同时，kernel会“监视”所有select负责的socket，当任何一个socket中的数据准备好了，select就会返回。这个时候用户进程再调用read操作，将数据从kernel拷贝到用户进程。

这个图和blocking IO的图其实并没有太大的不同，事实上，还更差一些。因为这里需要使用两个system call (select 和 recvfrom)，而blocking IO只调用了一个system call (recvfrom)。但是，用select的优势在于它可以同时处理多个connection。（多说一句。所以，如果处理的连接数不是很高的话，使用select/epoll的web server不一定比使用multi-threading + blocking IO的web server性能更好，可能延迟还更大。select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）

在IO multiplexing Model中，实际中，对于每一个socket，一般都设置成为non-blocking，但是，如上图所示，整个用户的process其实是一直被block的。只不过process是被select这个函数block，而不是被socket IO给block。

### 1.3.4异步Asynchronous I/O

Linux下的asynchronous IO其实用得很少。先看一下它的流程：



用户进程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从kernel的角度，当它受到一个asynchronous read之后，首先它会立刻返回，所以不会对用户进程产生任何block。然后，kernel会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel会给用户进程发送一个signal，告诉它read操作完成了。

最后，再举几个不是很恰当的例子来说明这四个IO Model:

有A，B，C，D四个人在钓鱼：

A用的是最老式的鱼竿，所以呢，得一直守着，等到鱼上钩了再拉杆；

B的鱼竿有个功能，能够显示是否有鱼上钩，所以呢，B就和旁边的MM聊天，隔会再看看有没有鱼上钩，有的话就迅速拉杆；

C用的鱼竿和B差不多，但他想了一个好办法，就是同时放好几根鱼竿，然后守在旁边，一旦有显示说鱼上钩了，它就将对应的鱼竿拉起来；

D是个有钱人，干脆雇了一个人帮他钓鱼，一旦那个人把鱼钓上来了，就给D发个短信。

## 1.4 Java I/O

I/O 指的是计算机与外部世界或者一个程序与计算机的其余部分的之间的接口。

### 1.4.1 Java I/O

#### 1.4.1.1Java 的 I/O 类库的基本架构

I/O 问题是任何编程语言都无法回避的问题，可以说 I/O 问题是整个人机交互的核心问题，因为 I/O 是机器获取和交换信息的主要渠道。

Java 的 I/O 操作类在包 java.io 下，大概有将近 80 个类，但是这些类大概可以分成四组，分别是：

* 基于字节操作的 I/O 接口：InputStream 和 OutputStream
* 基于字符操作的 I/O 接口：Writer 和 Reader
* 基于磁盘操作的 I/O 接口：File
* 基于网络操作的 I/O 接口：Socket

前两组主要是根据传输数据的数据格式，后两组主要是根据传输数据的方式，虽然 Socket 类并不在 java.io 包下，因为我个人认为 I/O 的核心问题要么是数据格式影响 I/O 操作，要么是传输方式影响 I/O 操作。

### 1.4.2 Java NIO

#### 1.4.2.1 NIO简介

新的输入/ 输出 (NIO) 库是在 JDK 1.4 中引入的。NIO 弥补了原来的 I/O 的不足，它在标准 Java 代码中提供了高速的、面向块的 I/O。

**NIO与I/O比较：**

|  |  |  |
| --- | --- | --- |
| **ITEM** | **NIO** | **IO** |
| 包 | java.nio.\* | java.io.\* |
| 版本 | JDK1.4 | JDK1.0 |
| 数据处理方式 | 通过定义包含数据的类，以及通过以块的形式处理这些数据，NIO 不用使用本机代码就可以利用低级优化，每一个操作都在一步中产生或者消费一个数据块。 | 使用流的方式完成，所有I/O都被视为单个字节的移动，通过Stream的对象一次移动一个字节。流I/O用于与外部世界接触，内部使用则将对象转换为字节，然后再转为对象。(需序列化) |
| 优点 | 1. NIO 将最耗时的 I/O 操作  (即填充和提取缓冲区) 转移回操作系统，因而可以极大地提高速度。  2.按块处理数据比按( 流式的) 字节处理数据要快得多。 | 为流式数据创建过滤器非常容易。链接几个过滤器，以便每个过滤器只负责单个复杂处理机制的一部分，这样也是相对简单的 |
| 缺点 | 面向块的 I/O 缺少一些面向流的 I/O 所具有的优雅性和简单性。 | 面向流的 I/O 通常相当慢。 |
| 集成I/O |  |  |

**集成的 I/O**

在JDK 1.4中原来的 I/O 包和 NIO 已经很好地集成了。 java.io.\* 已经以 NIO 为基础重新实现了，所以现在它可以利用 NIO 的一些特性。例如， java.io.\* 包中的一些类包含以块的形式读写数据的方法，这使得即使在更面向流的系统中，处理速度也会更快。

也可以用 NIO 库实现标准 I/O 功能。例如，可以容易地使用块 I/O 一次一个字节地移动数据。但是正如您会看到的，NIO 还提供了原 I/O 包中所没有的许多好处。

#### 1.4.2.2 通道和缓冲区

通道和缓冲区是 NIO 中的核心对象，几乎在每一个 I/O 操作中都要使用它们。 通道是对原 I/O 包中的流的模拟。到任何目的地( 或来自任何地方) 的所有数据都必须通过一个 Channel 对象。一个Buffer 实质上是一个容器对象。发送给一个通道的所有对象都必须首先放到缓冲区中；同样地，从通道中读取的任何数据都要读到缓冲区中。

**缓冲区**

Buffer 是一个对象，它包含一些要写入或者刚读出的数据。在 NIO 中加入Buffer 对象，体现了新I/O库与原 I/O 的一个重要区别。在面向流的 I/O 中，您将数据直接写入或者将数据直接读到Stream 对象中。

在 NIO 库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的。在写入数据时，它是写入到缓冲区中的。任何时候访问 NIO 中的数据，您都是将它放到缓冲区中。

缓冲区实质上是一个数组。通常它是一个字节数组，但是也可以使用其他种类的数组。但是一个缓冲区不仅仅是一个数组。缓冲区提供了对数据的结构化访问，而且还可以跟踪系统的读/ 写进程。

缓冲区类型：每一种基本 Java 类型都有一种与之对应的缓冲区类型，如下：

* ByteBuffer
* CharBuffer
* ShortBuffer
* IntBuffer
* LongBuffer
* FloatBuffer
* DoubleBuffer

**缓冲区内部细节**

状态变量，可以用三个值指定缓冲区在任意时刻的状态：

* position：缓冲区实际上就是美化了的数组。在从通道读取时，您将所读取的数据放到底层的数组中。 position变量指定了下一个字节将放到数组的哪一个元素中。因此，如果您从通道中读三个字节到缓冲区中，那么缓冲区的 position 将会设置为 3，指向数组中第四个元素。 同样，在写入通道时，您是从缓冲区中获取数据。 position值跟踪从缓冲区中获取了多少数据。更准确地说，它指定下一个字节来自数组的哪一个元素。因此如果从缓冲区写了 5 个字节到通道中，那么缓冲区的 position 将被设置为 5，指向数组的第六个元素。
* limit：变量表明还有多少数据需要取出(在从缓冲区写写入通道时)，或者还有多多少空间可以放入数据(在在从通道读入缓缓冲区时时)。

position 总是是小于或者等于于 limit。

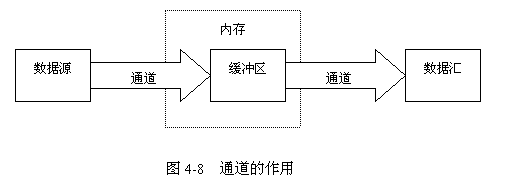
* capacity：缓冲冲区的 capaccity 表明可可以储存在缓冲冲区中的最大大数据容量。实际上，它指指定了底层数组组的大小 ― 或者至少是是指定了准准许我们使用的的底层数组的的容量。

limmit 决不能大大于 capacity。

这三个变量一起可以跟踪缓冲区的状态和它所包含的数据。

**通道**

Channel是一个对象，可以通过它读取和写入数据。拿 NIO 与原来的 I/O 做个比较，通道就像是流。正如前面提到的，所有数据都通过 Buffer对象来处理。您永远不会将字节直接写入通道中，相反，您是将数据写入包含一个或者多个字节的缓冲区。同样，您不会直接从通道中读取字节，而是将数据从通道读入缓冲区，再从缓冲区获取这个字节



通道类型

通道与流的不同之处在于通道是双向的。而流只是在一个方向上移动( 一个流必须是InputStream 或者 OutputStream 的子类)， 而通道可以用于读、写或者同时用于读写。因为它们是双向的，所以通道可以比流更好地反映底层操作系统的真实情况。特别是在 UNIX 模型中，底层操作系统通

道是双向的。

#### 1.4.2.3 NIO的读和写

读和写是 I/O 的基本过程。从一个通道中读取很简单：只需创建一个缓冲区，然后让通道将数据读到这个缓冲区中。写入也相当简单：创建一个缓冲区，用数据填充它，然后让通道用这些数据来执行写入操作。

在 NIO系统中，任何时候执行一个读操作，您都是从通道中读取，但是您不是 直接 从通道读取。因为所有数据最终都驻留在缓冲区中，所以您是从通道读到缓冲区中。

**get() 方法**

ByteBuffer 类中有四个 get() 方法：

* byte get();
* ByteBuffer get( byte dst[] );
* ByteBuffer get( byte dst[], int offset, int length );
* byte get( int index );

第一个方法获取单个字节。第二和第三个方法将一组字节读到一个数组中。第四个方法从缓冲区中的特定位置获取字节。 那些返回 ByteBuffer 的方法只是返回调用它们的缓冲区的 this 值。

此外，我们认为前三个 get() 方法是相对的，而最后一个方法是绝对的。 相对 意味着 get() 操作服从 limit 和 position 值 ― 更明确地说，字节是从当前 position 读取的，而 position 在 get 之后会增加。另一方面，一个绝对方法会忽略 limit 和 position 值，也不会影响它们。事实上，它完全绕过了缓冲区的统计方法。

上面列出的方法对应于 ByteBuffer 类。其他类有等价的 get() 方法，这些方法除了不是处理字节外，其它方面是是完全一样的，它们处理的是与该缓冲区类相适应的类型。

**put()方法**

ByteBuffer 类中有五个 put() 方法：

* ByteBuffer put( byte b );
* ByteBuffer put( byte src[] );
* ByteBuffer put( byte src[], int offset, int length );
* ByteBuffer put( ByteBuffer src );
* ByteBuffer put( int index, byte b );

第一个方法 写入（put） 单个字节。第二和第三个方法写入来自一个数组的一组字节。第四个方法将数据从一个给定的源 ByteBuffer 写入这个 ByteBuffer。第五个方法将字节写入缓冲区中特定的 位置 。那些返回 ByteBuffer 的方法只是返回调用它们的缓冲区的 this 值。

与 get() 方法一样，我们将把 put() 方法划分为相对或者绝对的。前四个方法是相对的，而第五个方法是绝对的。

上面显示的方法对应于 ByteBuffer 类。其他类有等价的 put() 方法，这些方法除了不是处理字节之外，其它方面是完全一样的。它们处理的是与该缓冲区类相适应的类型。

### 1.4.3 Java AIO

# 2 Java Socket

无论何种方式实现，Socket服务端共同的结构如下：

STEP1：Read Request -- 接受请求

STEP2：Decode Request -- 请求值解码(读)

STEP3：Process Service -- 请求处理

STEP4：Encode Reply -- 响应值编码(写)

STEP5：Send Reply -- 发送响应

## 2.1 IO

传统的并发型服务器则是采用多线程的模式响应用户请求的；

*//阻塞*

*while(一直等待){*

*if(有客户连接)*

*启动新线程，与客户的通信; //可能会阻塞*

*}*

## 2.2 NIO

Java NIO(new IO)是JDK1.4引入的非阻塞IO机制（详细参考IO专题 ）

Java NIO引入了两个新的概念：通道Channel和选择器Selector；

**通道**是服务端和客户端进行通信的接口-----原来是直接的IO操作，客户端发信息给服务端，服务端从OutputStream中读取，然后向InputStream中写数据；现在则直接从Channel中读取或写入数据；

**选择器**是一个多路复用器：所有的通道向它注册事件，因此它管理了所有的通道信息，并轮询各个通道的状态，一旦某个通道某事件发生（比如有数据读或可以写入数据），则通知该管道对应事件的处理器去处理它；

如下图所示：



客户端连接上服务端后，首先每个客户端都要与服务端建立一个通道（SocketChannel）；然后每个通道向选择器(Selector)注册事件，注册器就会轮询查看每个通道是否有事件发生，一旦某通道有事件发生，比如Client1的SocketChannel有数据了，就触发了读就绪事件，可以进行读取的操作啦。

## 2.3 Mina2

### 2.3.1 Mina2介绍

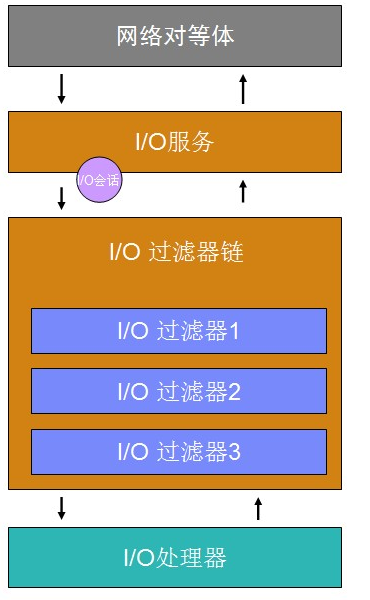
Apache Mina 2 是一个开发高性能和高可伸缩性网络应用程序的网络应用框架。它提供了一个抽象的事件驱动的异步 API，可以使用 TCP/IP、UDP/IP、串口和虚拟机内部的管道等传输方式。Apache Mina 2 可以作为开发网络应用程序的一个良好基础。

基于 Apache MINA 开发的网络应用，有着相似的架构。下图中给出了架构的示意图。基于 Apache MINA 的网络应用有三个层次，分别是 I/O 服务、I/O 过滤器和 I/O 处理器：

I/O 服务：I/O 服务用来执行实际的 I/O 操作。Apache MINA 已经提供了一系列支持不同协议的 I/O 服务，如 TCP/IP、UDP/IP、串口和虚拟机内部的管道等。开发人员也可以实现自己的 I/O 服务。

I/O 过滤器：I/O 服务能够传输的是字节流，而上层应用需要的是特定的对象与数据结构。I/O 过滤器用来完成这两者之间的转换。I/O 过滤器的另外一个重要作用是对输入输出的数据进行处理，满足横切的需求。多个 I/O 过滤器串联起来，形成 I/O 过滤器链。

I/O 处理器：I/O 处理器用来执行具体的业务逻辑。对接收到的消息执行特定的处理。



创建一个完整的基于 Apache MINA 的网络应用，需要分别构建这三个层次。Apache MINA 已经为 I/O 服务和 I/O 过滤器提供了不少的实现，因此这两个层次在大多数情况下可以使用已有的实现。I/O 处理器由于是与具体的业务相关的，一般来说都是需要自己来实现的。

### 2.3.2 Mina主要接口

**I/O 服务**

I/O 服务用来执行真正的 I/O 操作，以及管理 I/O 会话。根据所使用的数据传输方式的不同，有不同的 I/O 服务的实现。由于 I/O 服务执行的是输入和输出两种操作，实际上有两种具体的子类型。一种称为“I/O 接受器（I/O acceptor）”，用来接受连接，一般用在服务器的实现中；另外一种称为“I/O 连接器（I/O connector）”，用来发起连接，一般用在客户端的实现中。

**I/O 接受器**

I/O 接受器用来接受连接，与对等体（客户端）进行通讯，并发出相应的 I/O 事件交给 I/O 处理器来处理。使用 I/O 接受器的时候，只需要调用 bind方法并指定要监听的套接字地址。当不再接受连接的时候，调用 unbind停止监听即可。

**I/O 连接器**

I/O 连接器用来发起连接，与对等体（服务器）进行通讯，并发出相应的 I/O 事件交给 I/O 处理器来处理。使用 I/O 连接器的时候，只需要调用 connect方法连接指定的套接字地址。另外可以通过 setConnectTimeoutMillis设置连接超时时间（毫秒数）。

**I/O 会话**

I/O 会话表示一个活动的网络连接，与所使用的传输方式无关。I/O 会话可以用来存储用户自定义的与应用相关的属性。这些属性通常用来保存应用的状态信息，还可以用来在 I/O 过滤器和 I/O 处理器之间交换数据。 I/O 会话在作用上类似于 Servlet 规范中的 HTTP 会话。Apache MINA 中 I/O 会话实现的接口是 org.apache.mina.core.session.IoSession。

## 2.4 线程池

使用线程池时需要遵循以下原则：

* 如果任务A在执行过程中需要同步等待任务B的执行结果，那么任务A不适合加入到线程池的工作队列中。
* 如果执行某个任务时可能会阻塞，并且是长时间的阻塞，则应该设定超时时间，避免工作线程永久的阻塞下去而导致线程泄漏。
* 根据任务的特点，对任务进行分类，然后把不同类型的任务分别加入到不同线程池的工作队列中，这样可以根据任务的特点，分别调整每个线程池。
* 调整线程池的大小。线程池的最佳大小主要取决于系统的可用CPU的数目以及工作队列中任务的特点。
* 避免任务过载。

# 3 持久化

持久化是将程序数据在持久状态和瞬时状态间转换的机制。

## 3.1定义

持久化（Persistence），即把数据（如内存中的对象）保存到可永久保存的存储设备中（如磁盘）。

持久化的主要应用是将内存中的对象存储在关系型的数据库中，当然也可以存储在磁盘文件中、XML数据文件中等等。

**举例**

* JDBC就是一种持久化机制。
* 文件IO也是一种持久化机制。
* 将鲜肉冷藏，吃的时候再解冻的方法也是。
* 将水果做成罐头的方法也是。
* 将人的脏器迅速冷冻，运输，然后解冻给人移植的技术也是。

## 3.2 持久化特点

在一定周期内保持不变就是持久化，持久化是针对时间来说的。简单的理解持久化可以在二个层面：应用层和系统层

**应用层**

如果关闭(shutdown)你的应用然后重新启动则先前的数据依然存在。

**系统层**

如果关闭(shutdown)你的系统（电脑）然后重新启动则先前的数据依然存在。

为什么需要持久化服务呢？那是由于内存本身的缺陷引起的：

内存掉电后数据会丢失，但有一些对象是无论如何都不能丢失的，比如银行账号，遗憾的是，人们还无法保证内存永不掉电。

内存过于昂贵，与硬盘、磁带、光盘等外存相比，内存的价格要高2~3个数量级，而且维持成本也高，至少需要一直供电吧。所以即使对象不需要永久保存，也会因为内存的容量限制不能一直呆在内存中，需要持久化来缓存到外存。

既然持久化服务在看得到的未来还有市场，我们就来看看如何构建一个好的持久化框架，框架是否真的好在于如何在扩展性、缩放性、重用性上取得良好的平衡：

扩展性，如果一个持久性框架不能支持用户定义的类型，显然不是一个好的框架。

缩放性，保存和取回对象都需要耗费cpu、带宽、时间资源，哪一个消耗太多都不能接受。

重用性是我们建立框架的初衷，就是通过框架能够减少一些编码和测试的工作量。

这几个需求往往是互相冲突的，所以关键是平衡。

## 3.3 持久化实现

**Hibernate**

hibernate为应用程序提供了高效的O/R关系映射和查询服务，为面向对象的领域模型到传统的关系型数据库的映射，提供了一个使用方便的框架。

**JPA**

JPA（Java Persistense API）是EJB3.0的一部分，为其提供了一套O/R关系映射的API，但不仅限于EJB中使用，它也可以在web应用或者应用程序客户端中被使用，甚至在Java桌面程序中被使用。

# 4 序列化

## 4.1 定义及简介

**序列化**

序列化是将对象状态转换为可保持或传输的格式的过程。与序列化相对的是反序列化，它将流转换为对象。

在序列化期间，对象将其当前状态写入到临时或持久性存储区。以后，可以通过从存储区中读取或反序列化对象的状态，重新创建该对象。

**注意**

对于任何可能包含重要的安全性数据的对象，如果可能，应该使该对象不可序列化。如果它必须为可序列化的，请尝试生成特定字段来保存不可序列化的重要数据。如果无法实现这一点，则应注意该数据会被公开给任何拥有序列化权限的代码，并确保不让任何恶意代码获得该权限。

## 4.2 目的及技术

**目的**

1.以某种存储形式使自定义对象持久化；

2.将对象从一个地方传递到另一个地方。

**技术**

 二进制序列化保持类型保真度，这对于在应用程序的不同调用之间保留对象的状态很有用。例如，通过将对象序列化到剪贴板，可在不同的应用程序之间共享对象。您可以将对象序列化到流、磁盘、内存和网络等等。远程处理使用序列化“通过值”在计算机或应用程序域之间传递对象。

 XML 序列化仅序列化公共属性和字段，且不保持类型保真度。当您要提供或使用数据而不限制使用该数据的应用程序时，这一点是很有用的。由于 XML 是一个开放式标准，因此，对于通过 Web 共享数据而言，这是一个很好的选择。SOAP 同样是一个开放式标准，这使它也成为一个颇具吸引力的选择。

## 4.3 JAVA序列化

序列化是为了解决在对对象流进行读写操作时所引发的问题。

### 4.3.1 实现Java序列化

**序列化的实现**

将需要被序列化的Java类通过实现 java.io.Serializable 接口以启用其序列化功能。未实现此接口的类将无法使其任何状态序列化或反序列化。可序列化类的所有子类型本身都是可序列化的。序列化接口没有方法或字段，仅用于标识可序列化的语义。

**关于序列化ID**

如果可序列化类未显式声明 serialVersionUID，则序列化运行时将基于该类的各个方面计算该类的默认 serialVersionUID 值，如“Java(TM) 对象序列化规范”中所述。不过，强烈建议 所有可序列化类都显式声明 serialVersionUID 值，原因是计算默认的 serialVersionUID 对类的详细信息具有较高的敏感性，根据编译器实现的不同可能千差万别，这样在反序列化过程中可能会导致意外的 InvalidClassException。因此，为保证 serialVersionUID 值跨不同 java 编译器实现的一致性，序列化类必须声明一个明确的 serialVersionUID 值。还强烈建议使用 private 修饰符显示声明 serialVersionUID（如果可能），原因是这种声明仅应用于直接声明类 -- serialVersionUID 字段作为继承成员没有用处。

**注意**

* 如果某个类能够被序列化，其子类也可以被序列化。
* 声明为static和transient类型的成员数据不能被序列化。因为static代表类的状态， transient代表对象的临时数据。

### Java序列化顺序

**Java的序列化顺序**

Step1：将对象实例相关的类元数据输出。

Step2：递归地输出类的超类描述直到不再有超类。

Step3：类元数据完了以后，开始从最顶层的超类开始输出对象实例的实际数据值。

Step4：从上至下递归输出实例的数据

**Java中序列化算法顺序**

Step1：子类的类信息描述

Step2：子类的字段信息描述（如果遇到类对象的属性，暂时用 string 的指针表示）

Step3：父类的类信息描述

Step4：父类的字段信息描述

Step5：对父类的字段信息进行赋值

Step6：对子类的字段信息进行赋值

Step7：发现子类的字段为类对象时，描述该类对象，并查找该类对象的父类，以以上方式描述，然后赋值。

### 4.3.3什么时候使用序列化

Java 中，一切都是对象，在分布式环境中经常需要将 Object 从这一端网络或设备传递到另一端。 这就需要有一种可以在两端传输数据的协议。Java 序列化机制就是为了解决这个问题而产生。

* 对象序列化可以实现分布式对象。主要应用例如：RMI要利用对象序列化运行远程主机上的服务，就像在本地机上运行对象时一样。
* Java对象序列化不仅保留一个对象的数据，而且递归保存对象引用的每个对象的数据。可以将整个对象层次写入字节流中，可以保存在文件中或在网络连接上传递。利用对象序列化可以进行对象的"深复制"，即复制对象本身及引用的对象本身。序列化一个对象可能得到整个对象序列。
* 序列化后的对象用于网络传输。

# 5 设计模式

参考阎宏的《java与模式》

# 6 OO设计

## 6.1 OO基础

抽象

封装

多态

继承

## 6.2 设计原则

* 针对接口编程，不针对实现编程。
* 对扩展开放，对修改关闭。
* 依赖倒置原则：要依赖抽象，不依赖具体类。
* 多用组合，少用继承
* 为交互对象之间的松耦合设计而努力
* 一个类，一个责任

## 6.3 设计方法

# 8 Java Web安全

url重写

rest

# 9 Java线程

## 9.1线程基础

### 9.1.1什么是线程

几乎每种操作系统都支持进程的概念 ―― 进程就是在某种程度上相互隔离的、独立运行的程序。

Java 是第一个在语言本身中显式地包含线程的主流编程语言，它没有把线程化看作是底层操作系统的工具。

有时候，线程也称作轻量级进程。就象进程一样，线程在程序中是独立的、并发的执行路径，每个线程有它自己的堆栈、自己的程序计数器和自己的局部变量。但是，与分隔的进程相比，进程中的线程之间的隔离程度要小。它们共享内存、文件句柄和其它每个进程应有的状态。

进程可以支持多个线程，它们看似同时执行，但互相之间并不同步。一个进程中的多个线程共享相同的内存地址空间，这就意味着它们可以访问相同的变量和对象，而且它们从同一堆中分配对象。尽管这让线程之间共享信息变得更容易，但您必须小心，确保它们不会妨碍同一进程里的其它线程。

Java 线程工具和 API 看似简单。但是，编写有效使用线程的复杂程序并不十分容易。因为有多个线程共存在相同的内存空间中并共享相同的变量，所以您必须小心，确保您的线程不会互相干扰。

### 9.1.2 使用线程的好处

* 使 UI 响应更快
* 利用多处理器系统
* 简化建模
* 执行异步或后台处理

**响应更快的 UI**

事件驱动的 UI 工具箱（如 AWT 和 Swing）有一个事件线程，它处理 UI 事件，如击键或鼠标点击。