2.HadoopCommon包分析

序列化与压缩

序列化

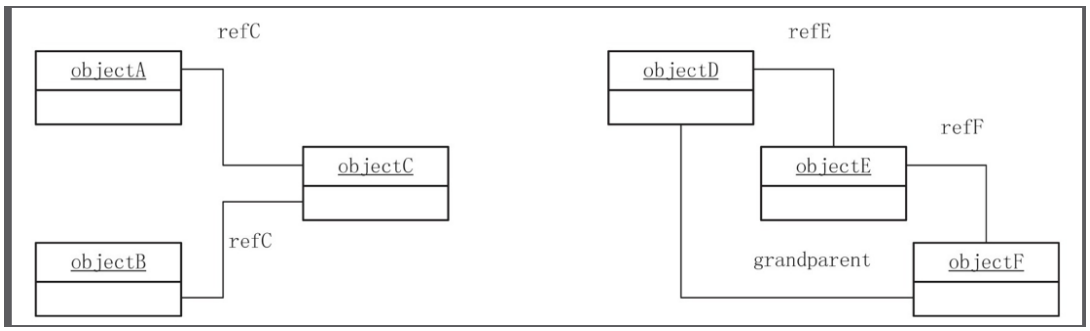
序列化的用途

1. 作为一种持久化，一个对象被持久化后，它的编码可以被存储到磁盘上，供以后反序列化使用。
2. 作为一种通信数据格式，序列化结果可以从一个正在运行的虚拟机，通过网络被传递到另一个虚拟机上
3. 作为一种拷贝、克隆机制:将对象序列化到内存的缓存区，然后通过反序列化，可以得到一个对已存对象进行深拷贝的新对象。

分布式数据处理中，主要使用数据持久化和通信数据格式。

Java的序列化机制

- 对象的类、类签名、类的所有非暂态和非静态成员的值，以及它所有的父类都要被写入。



Java序列化带来的问题

- Java序列化所需的数据量过大，以Block类说，它包含3个长整数，但是它的序列化有112字节，而BlockMetaDataInfo比他多一个long成员变量，但是序列化后的结果已经到190字节了。
- Java序列化Block首先前两个字节是魔法数然后是版本号，然后是类的描述信息包括类的版本ID、是否实现writeObject和readObject方法等信息，对于拥有超类的类，超类信息也会被递归保存。这些信息都写到OutputStream对象，然后才会写对象的数据。因此Hadoop不适用Java序列化需要一种新的序列化方式。

Hadoop序列化机制

- 和Java不同Hadoop的序列化机制通过调用对象的write方法（传递一个DataOutput对象），将对象序列化到流中。反序列化的过程也是相同，通过readFields从流中读取数据。
- Java的序列化机制，反序列化过程会不断创建新的对象，Hadoop的机制中反序列化过程中，用户可以复用对象。

Hadoop序列化机制的特征

- 紧凑:由于带宽是Hadoop集群中最稀缺的资源，一个紧凑的序列化机制可以节省很大的带宽消耗。

- 快速:在进程间通信(包括MR过程中涉及的数据交互)时会大量使用序列化机制，因此，必须尽量减少序列化和反序列化的开销。
- 可扩展:随着系统的发展，系统间通信的协议升级，类的定义会变化，序列化机制可以兼容。
- 互操作:可以支持与不同语言间通信。

可以说java的序列化机制不符合如上要求，首先java序列化机制携带过多附带信息导致序列化数据量过大，而且java序列化不支持跨语言并且java序列化会在反序列化的时候不断创建新的对象给Jvm带来过大的压力。

Hadoop Writable机制

特点

- Writable机制紧凑、快速 (不容易扩展到Java以外的语言)。

RowComparator

```
# 提供了基于二进制字节的比较，可以在不经过反序列化创建对象就可以对两个对象进行比较
public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2);
```

- WritableComparator

```
RowComparator的一个通用实现，主要包含了二进制字节的比较通用实现。
```

常用Writable分析

Java基本类型

表 3-1　Java 基本类型对应的 Writable 封装[⊖]

Java 基本类型	Writable	序列化后长度
布尔型（boolean）	BooleanWritable	1
字节型（byte）	ByteWritable	1
整型（int）	IntWritable VIntWritable	4 1~5
浮点型（float）	FloatWritable	4
长整型（long）	LongWritable VLongWritable	8 1~9
双精度浮点型（double）	DoubleWritable	8

- ObjectWritable实现
    - 针对Java基本类型、字符串、枚举、空值、Writable的其他子类，ObjectWritable提供了一个封装，适用于字段需要使用多种类型。可用于Hadoop的Rpc过程中参数的序列化和反序列化；
    - 并且支持序列化不同类型的对象到某一个字段，如在一个SequenceFile的值中保存不同类型的对象(如LongWritable值或Text值)时
    - 属性包含对象实例instance、对象运行时类的Class对象和Configuration对象

Hadoop序列化框架

- Hadoop自带的Writable方式
- Avro:一直数据序列化系统，用于支持大批量数据交换的应用。
    - 支持二进制序列化方式
    - 便捷快速地处理大量数据
    - 动态语言友好，提供的机制使动态语言可以方便地处理Avro数据

- Thrift:可伸缩的、跨语言的服务开发框架
  - 基于Thrift的跨平台能力封装的Hadoop文件系统Thrift API，提供不同开发的系统访问HDFS的能力。
- Protocol Buffer
  - 轻便高效的结构化数据存储格式，支持多种语言。
- hadoop实现的Serialization接口
  - 使用抽象工厂的涉及模式，提供了一系列和序列化相关依赖对象的接口。

压缩

压缩解压缩格式

表 3-2　Hadoop 支持的压缩格式

压缩格式	UNIX 工具	算法	文件扩展名	支持多文件	可分割
DEFLATE	无	DEFLATE	.deflate	否	否
gzip	gzip	DEFLATE	.gz	否	否
zip	zip	DEFLATE	.zip	是	是
bzip	bzip2	bzip2	.bz2	否	是
LZO	lzop	LZO	.lzo	否	否

表 3-3　压缩算法及其编码 / 解码器

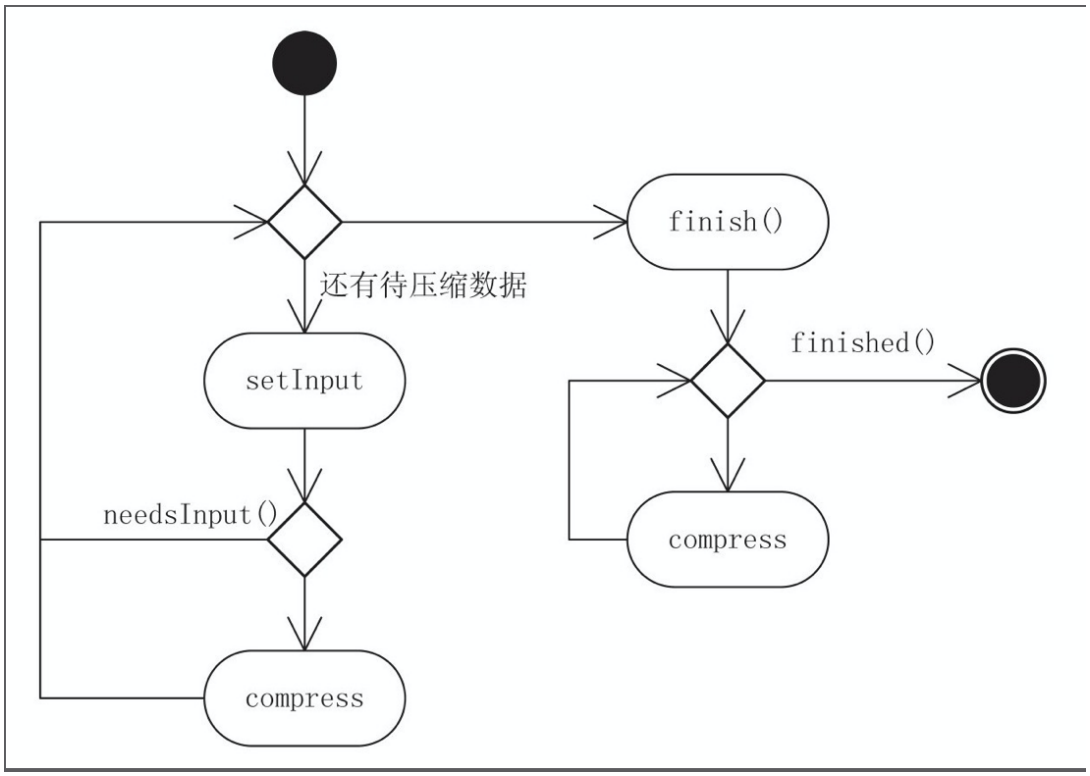
压缩格式	对应的编码 / 解码器
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip	org.apache.hadoop.io.compress.BZip2Codec
Snappy	org.apache.hadoop.io.compress.SnappyCodec ⊖

Compressor

- setInput()方法接收数据懂啊内部缓冲区，可以多次调用，内部缓冲区会被写满。
  - 用户调用完setInput()方法后，压缩器并不会立马去压缩，如果全部数据已经写入必须通过finish()方法。
  - 做一番输入数据的合法性检查后，先将finished标志位置为false，并尝试将输入数据复制到内部缓冲区中。如果内部缓冲区剩余空间不足，那么压缩器将借用输入数据对应的缓冲区，利用userBuf、userBufOff和userBufLen记录输入的数据。否则直接将数据复制到uncompressedDirectbuf中即可
- needsInput()方法判断内部缓冲区是否写满，如果为false表示内部缓冲区已经写满。
- compress()获取压缩后的数据，释放缓冲区空间。
- finished()返回false表明压缩器中还有未读取的压缩数据，可以继续通过compress()方法读取

压缩流和解压缩流

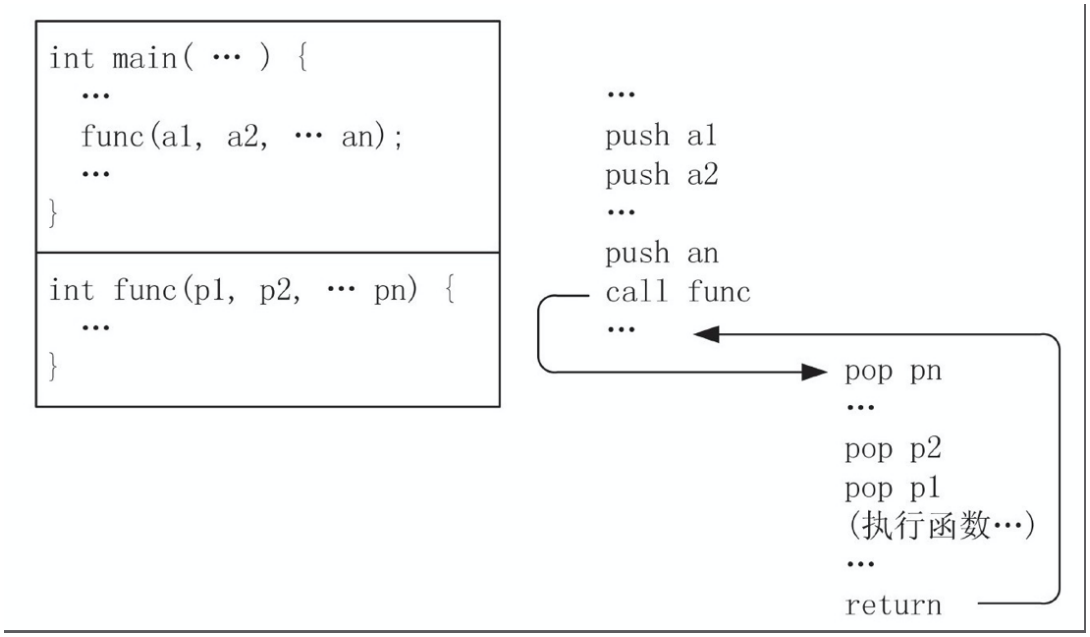
- CompressionOutputStream压缩流
- COmpressionInputStream解压缩流



Hadoop远程过程调用

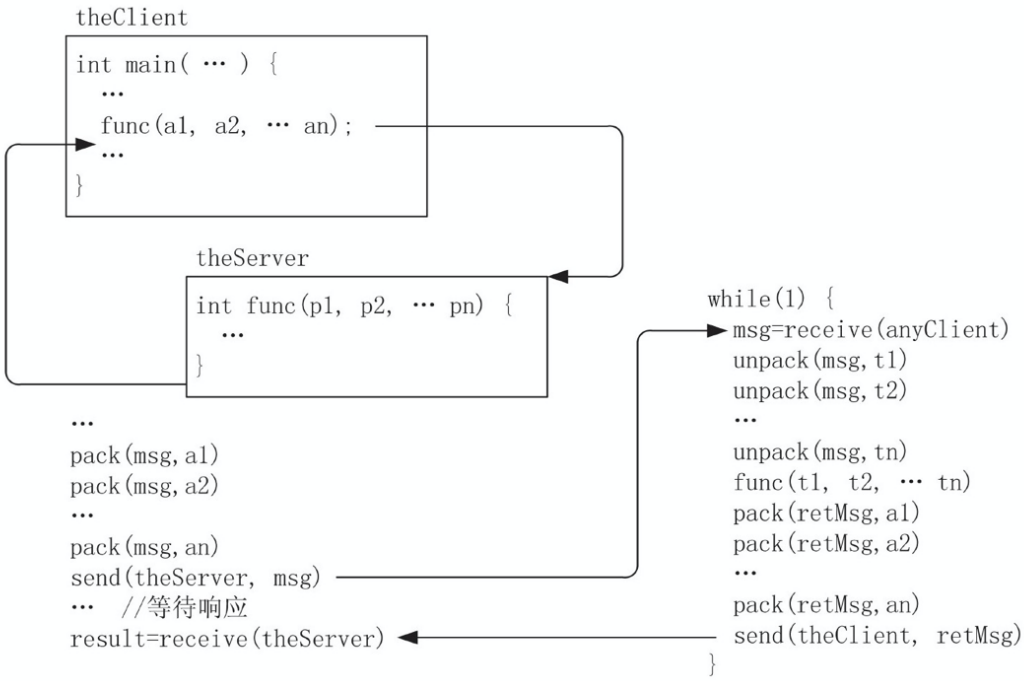
传统调用

- 主程序将参数压入栈内并调用过程，这时候主程序停止执行并开始执行相应的过程。被调用的过程从栈中获取参数，然后执行过程函数；执行完毕后，将返回参数入栈(或保存在寄存器里)，并将控制权交还给调用方。调用方获取返回参数，并继续执行。

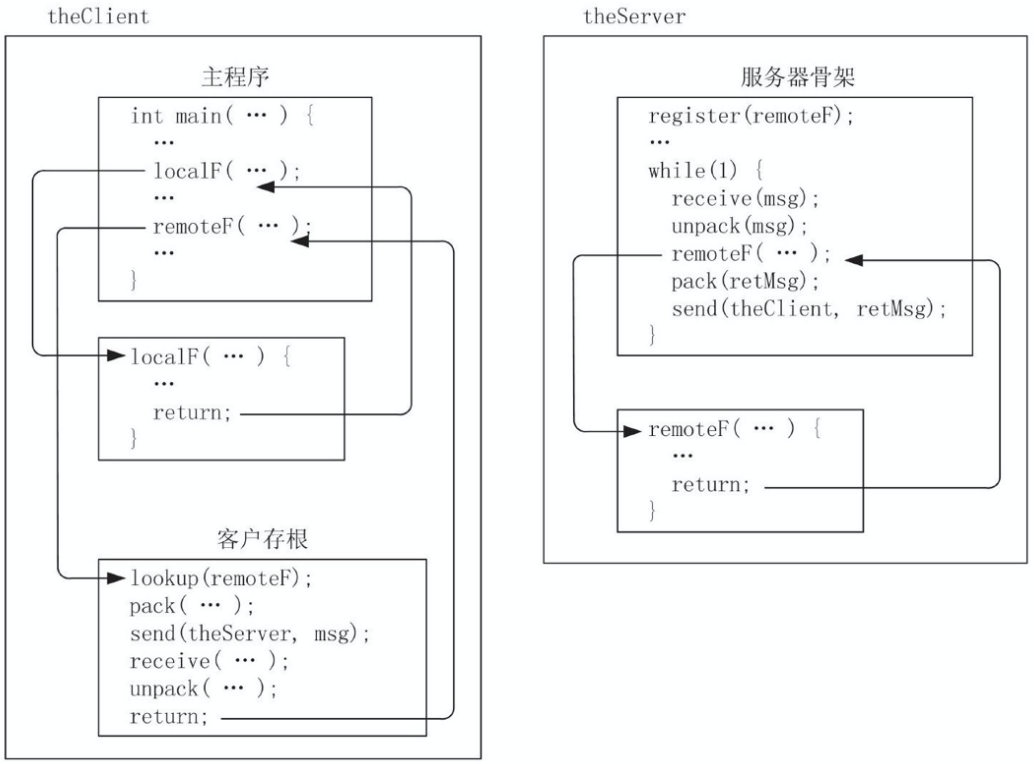


RPC过程

- Rpc的Sever运行时会阻塞在接收消息的调用上，当接到客户端的请求后，它会解包以获取请求参数，类似传统过程调用，被调用函数从栈中接收参数，然后确定调用过程的名字并调用相应过程。调用结束后，返回值通过主程序打包并发送回客户端，通知客户端调用结束。



RPC机制的实现



Java远程方法调用(RMI)

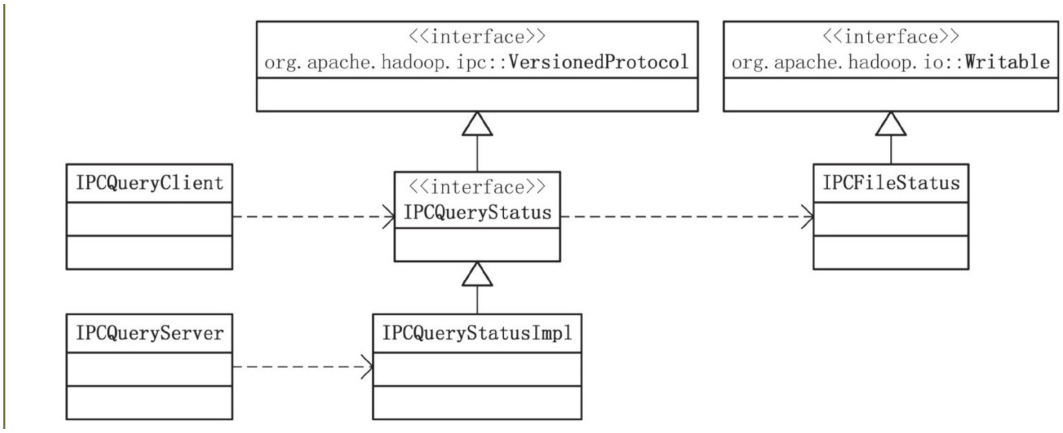
- Java RMI是Java的一个核型API和类库，运行一个JVM运行的Java程序调用在不同JVM虚拟机上运行的对象中的方法，即使这两个虚拟机运行于物理隔离的不同主机上。

Hadoop远程过程调用

- Hadoop实现了自己的一套远程过程调用，Hadoop的IPC提供了进程间连接、超时、缓冲等通信细节。

Hadoop IPC构建简单的分布式系统

- 实现Hadoop IPC必须实现VersionedProtocol接口与Java RMI类似(需要实现Remote接口)。



Java NIO

Java基本套接字

```
基于流的基本套接字(socket)实现，socket是两台主机间的一个连接
```

- 连接远程机器
- 发送数据
- 接收数据
- 关闭连接
- 绑定端口
- 监听入站数据
- 在所绑定端口上接受来自远程机器的连接

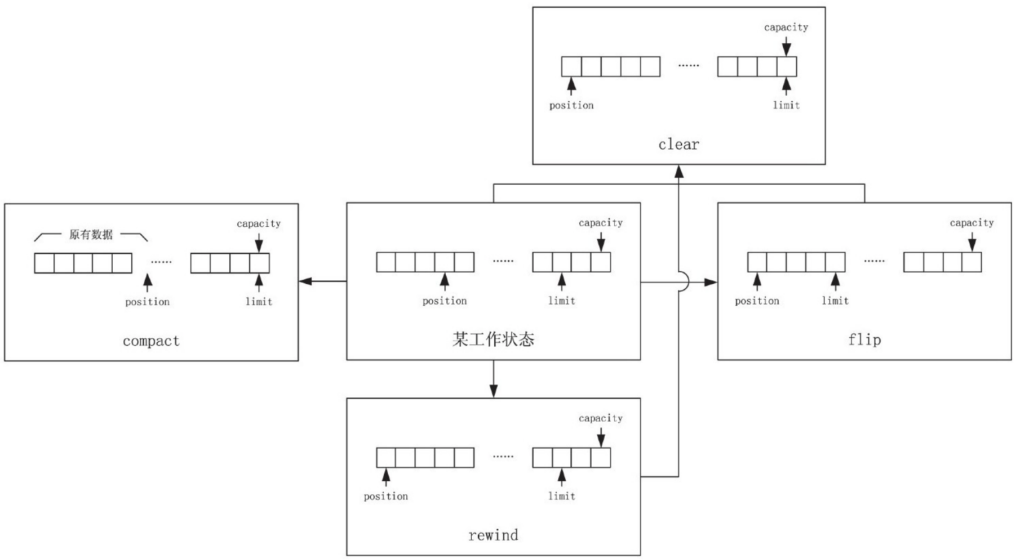
缓冲区

- Nio的主要特性Buffer，缓冲区提供一个比流抽象的、更高效的和可预测的I/O。Buffer代表流一个有限容量的容器-其本质是一个数组，通过Channel使用Buffer来传递数据。
- 流和通道的差别在于流基于字节，通道基于块(Buffer)。

Buffer的结构

- capacity:缓冲区的元素总数(容量)，通过Buffer.capacity()可以获取该索引。缓冲区元素总数是不可修改的。
- position:缓冲区的位置,是下一个要读取或写入的元素的索引该位置可以有position()方法和position(int)方法获取和设置。
- limit:缓冲区的限制，即第一个不应该读取或写入的元素的索引，limit()方法和limit(int)方法可用于获取和设置缓冲区的限制。
- mark:缓冲区的位置设置标记，通过mark()方法设置一个位置，然后利用reset()方法可以将position重制为mark()方法设置的位置。

```
不变式:0<=mark <= position <= limit <= capacity
```



- compact()方法将position和limit间的数据复制到缓冲区的开始位置，为后续的put()/read()调用让出空间。调用结束后，poistion的值被设置为数据的长度，也就是原来的limit减去position的值，而limit则设置为capacity。

通道

- 通道(Channel)是Nio的另一个概念。Channel类似于普通的套接字，对于TCP来提供了ServerSocketChannel和Socketchannel分别对应IO中的ServerSocket和Socket。

数据的读写

- Channel基于Buffer来做数据的读写，用Buffer来作为读写数据的载体。
- 和普通的Socket一样，结合Buffer使用阻塞式Channel几乎没什么有点。需要把Channel设置为非阻塞式才能提高吞吐量和性能。

```
# 设置阻塞非阻塞
public final SelectableChannel configureBlocking(boolean block)
# 判断是否阻塞
public boolean isBlocking()
```

- 非阻塞式ScoketChannel和阻塞时Socket的不同是，非阻塞式的connect方法会立刻返回，用户必须通过isConnect来确定是否建立了连接或者通过finishConnect()方法在非阻塞套接字上阻塞等待连接成功。非阻塞read(),在Socket上没有数据的时候，立刻返回(返回值为0)，不会等待；非阻塞accpt()如果没有等地啊的连接就返回null。
- ServerSocketChannel/Socketchannel和基本套接字的最后一个差别是它们能够和选择器配合工作，避免非阻塞式IO操作很浪费资源的忙等方法。如HDFS中，同时有成千上百个DataNode连接到NameNode服务器，但在任何时刻，服务器都只有少量的(甚至没有)请求需要处理，这就需要一种方法阻塞等待，直到至少有一个Channel可以进行IO操作，并指出是哪个通道。NIO选择器就是为此设计，一个选择器可以同时检查(等待)一组通道的IO状态。

选择器

- 通过静态的工厂方法创建Selector实例，通过Channel的注册方法，将Selector实例注册到想要监控的Channel实例上，最后调用选择器的select()方法。该方法会阻塞等待，知道一个或多个通道准备好IO操作或超时。

SelectionKey支持的事件类型

- OP_READ(通道上有数据可读)

- OP_WRITE(通道已经可写)
- OP_CONNECT(通道连接已建立)
- OP_ACCEPT(通道上有连接请求)

```
这些类型都是SelectionKey类中定义的常量，并且通过位图(bitmap)保存在SelectionKey实例
```