# Executor Future FutureTask

## Future RunnableFuture FutureTask

**Future是一个接口，用来取异步计算的结果V，可以终止执行**

public interface **Future<V>** {

boolean cancel(boolean mayInterruptIfRunning);

boolean isDone();

V get() throws InterruptedException, ExecutionException;

V get(long timeout, TimeUnit unit)

throws InterruptedException, ExecutionException, TimeoutException;

}

**RunnableFuture也是接口，继承了Runnable和Future接口**

public interface **RunnableFuture<V>** extends Runnable, Future<V> {

void run();

}

**FutureTask是实现类，实现了Future和Runnable接口中的方法**

public class **FutureTask<V>** implements RunnableFuture<V> {

private Callable<V> callable;

private volatile Thread runner;

// 使用Runnable构造

public FutureTask(Runnable runnable, V result) {

this.callable = Executors.callable(runnable, result);

this.state = NEW; // ensure visibility of callable

}

// 使用Callable构造

public FutureTask(Callable<V> callable) {

if (callable == null)

throw new NullPointerException();

this.callable = callable;

this.state = NEW; // ensure visibility of callable

}

// 执行Thread的interruput

public boolean cancel(boolean mayInterruptIfRunning) {

Thread t = runner;

t.interrupt();

}

// 执行Callable的call，并得到结果

public void run() {

Callable<V> c = callable;

result = c.call();

}

}

## Executor ExecutorService AbstractExecutorService ThreadPoolExecutor

Executor可以直接调用execute执行一个runnable

ExecutorService可以调用submit执行，返回一个Future，用来取执行的结果

**Executor是一个接口，包含唯一的execute方法**

public interface Executor {

void execute(Runnable command);

}

**ExecutorService也是接口，新增了shutdown，submit**

public interface ExecutorService extends Executor {

void shutdown();

List<Runnable> shutdownNow();

boolean isShutdown();

boolean isTerminated();

<T> Future<T> submit(Callable<T> task);

<T> Future<T> submit(Runnable task, T result);

Future<?> submit(Runnable task);

invokeAll(…)

…

}

**AbstractExecutorService是抽象的类，其功能是把submit的Runnable转换成FutureTask后执行**

public abstract class **AbstractExecutorService** implements ExecutorService {

public <T> Future<T> submit(Runnable task, T result) {

if (task == null) throw new NullPointerException();

RunnableFuture<T> ftask = new FutureTask<T>( task, result);

execute(ftask);

return ftask;

}

}

**ThreadPoolExecutor是具体实现类，使用线程池实现了execute和shutdown等**

public class **ThreadPoolExecutor** extends AbstractExecutorService {

public void execute(Runnable command) {

}

public void shutdown() {

}

}

// 使用示例

mWorkQueue = new PriorityBlockingQueue<Runnable>(Const.WORK\_QUEUE\_MAX\_COUNT);

mExecutor = new ThreadPoolExecutor(Const.THREAD\_DEF\_WORDER\_COUNT,

Const.THREAD\_MAX\_WORKER\_COUNT, Const.THREAD\_KEEP\_ALIVE\_TIME,

TimeUnit.SECONDS, mWorkQueue, new ThreadFactory() {

private final AtomicInteger mCount = new AtomicInteger(1);

public Thread newThread(Runnable r) {

return new Thread(r, "HttpHandlerImpl ConnectTask #"

+ mCount.getAndIncrement());

}

}, new RejectedExecutionHandler() {

public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

if (!e.isShutdown()) {

Runnable runnable = e.getQueue().poll();

e.execute(r);

}

}

}); // 或者new ThreadPoolExecutor.DiscardOldestPolicy()

# Syncronized ReentrantLock

# LinkedHashMap Iterator

next

remove

# InputStream OutputStream

## OutputStream

**public** **abstract** **class** OutputStream **implements** Closeable, Flushable {

**public** OutputStream() {

}

**public** **void** close() **throws** IOException {

}

**public** **void** flush() **throws** IOException {

}

**public** **void** write(**byte**[] buffer) **throws** IOException {

write(buffer, 0, buffer.length);

}

**// 从buffer的offset开始写入count个字节**

**public** **void** write(**byte**[] buffer, **int offset**, **int** count) **throws** IOException {

Arrays.checkOffsetAndCount(buffer.length, offset, count);

**for** (**int** i = offset; i < offset + count; i++) {

write(buffer[i]);

}

}

**public** **abstract** **void** write(**int** oneByte) **throws** IOException;

**boolean** checkError() {

**return** **false**;

}

}

## ByteArrayOutputStream

**public** **class** ByteArrayOutputStream **extends** OutputStream {

**protected** **byte**[] buf;

**protected** **int** count;

**public** ByteArrayOutputStream() {

buf = **new** **byte**[32];

}

**public** ByteArrayOutputStream(**int** size) {

**if** (size >= 0) {

buf = **new** **byte**[size];

} **else** {

**throw** **new** IllegalArgumentException("size < 0");

}

}

@Override

**public** **void** close() **throws** IOException {

**super**.close();

}

**private** **void** expand(**int** i) {

**if** (count + i <= buf.length) {

**return**;

}

**// 以写入的2倍扩充buffer**

**byte**[] newbuf = **new** **byte**[(count + i) \* 2];

System.*arraycopy*(buf, 0, newbuf, 0, count);

buf = newbuf;

}

**public** **synchronized** **void** reset() {

count = 0;

}

**public** **int** size() {

**return** count;

}

**// 拷贝并产生一个新的数组**

**public** **synchronized** **byte**[] **toByteArray**() {

**byte**[] newArray = **new** **byte**[count];

System.*arraycopy*(buf, 0, newArray, 0, count);

**return** newArray;

}

@Override

**public** String toString() {

… …

}

@Deprecated

**public** String ~~toString~~(**int** hibyte) {

… …

}

**public** String toString(String charsetName) **throws** UnsupportedEncodingException {

**return** **new** String(buf, 0, count, charsetName);

}

@Override

**public** **synchronized** **void** write(**byte**[] buffer, **int** offset, **int** len) {

Arrays.checkOffsetAndCount(buffer.length, offset, len);

**if** (len == 0) {

**return**;

}

expand(len);

System.*arraycopy*(buffer, offset, buf, **this**.count, len);

**this**.count += len;

}

@Override

**public** **synchronized** **void** write(**int** oneByte) {

**if** (count == buf.length) {

expand(1);

}

buf[count++] = (**byte**) oneByte;

}

**public** **synchronized** **void** writeTo(OutputStream out) **throws** IOException {

out.write(buf, 0, count);

}

}

## FilterOutputStream

## BufferedOutputStream

**public** **class** BufferedOutputStream **extends** FilterOutputStream {

**protected** **byte**[] buf;

**protected** **int** count;

**public** BufferedOutputStream(OutputStream out) {

**this**(out, 8192);

}

**public** BufferedOutputStream(OutputStream out, **int** size) {

**super**(out);

**if** (size <= 0) {

**throw** **new** IllegalArgumentException("size <= 0");

}

buf = **new** **byte**[size];

}

@Override

**public** **synchronized** **void** flush() **throws** IOException {

… …

}

**private** **void** checkNotClosed() **throws** IOException {

… …

}

@Override

**public** **synchronized** **void** write(**byte**[] buffer, **int** offset, **int** length) **throws** IOException {

checkNotClosed();

**if** (buffer == **null**) {

**throw** **new** NullPointerException("buffer == null");

}

**byte**[] internalBuffer = buf;

**if** (length >= internalBuffer.length) {

flushInternal();

out.write(buffer, offset, length);

**return**;

}

Arrays.checkOffsetAndCount(buffer.length, offset, length);

// flush the internal buffer first if we have not enough space left

**if** (length > (internalBuffer.length - count)) {

flushInternal();

}

System.*arraycopy*(buffer, offset, internalBuffer, count, length);

count += length;

}

@Override **public** **synchronized** **void** close() **throws** IOException {

… …

}

@Override

**public** **synchronized** **void** write(**int** oneByte) **throws** IOException {

checkNotClosed();

**if** (count == buf.length) {

out.write(buf, 0, count);

count = 0;

}

buf[count++] = (**byte**) oneByte;

}

**private** **void** flushInternal() **throws** IOException {

**if** (count > 0) {

out.write(buf, 0, count);

count = 0;

}

}

}

## InputStream

**public** **abstract** **class** InputStream **extends** Object **implements** Closeable {

**public** InputStream() {

/\* empty \*/

}

**public** **int** available() **throws** IOException {

**return** 0;

}

**public** **void** close() **throws** IOException {

/\* empty \*/

}

**public** **void** mark(**int** readlimit) {

/\* empty \*/

}

**public** **boolean** markSupported() {

**return** **false**;

}

**public** **abstract** **int** read() **throws** IOException;

**public** **int** read(**byte**[] buffer) **throws** IOException {

**return** read(buffer, 0, buffer.length);

}

**public** **int** read(**byte**[] buffer, **int** byteOffset, **int** byteCount) **throws** IOException {

Arrays.checkOffsetAndCount(buffer.length, byteOffset, byteCount);

**for** (**int** i = 0; i < byteCount; ++i) {

**int** c;

**try** {

**if** ((c = read()) == -1) {

**return** i == 0 ? -1 : i;

}

} **catch** (IOException e) {

**if** (i != 0) {

**return** i;

}

**throw** e;

}

buffer[byteOffset + i] = (**byte**) c;

}

**return** byteCount;

}

**public** **synchronized** **void** reset() **throws** IOException {

**throw** **new** IOException();

}

**public** **long** skip(**long** byteCount) **throws** IOException {

**return** Streams.skipByReading(**this**, byteCount);

}

}

## ByteArrayOutputStream

**public** **class** ByteArrayInputStream **extends** InputStream {

**protected** **byte**[] buf;

**protected** **int** pos;

**protected** **int** mark;

**protected** **int** count;

**public** ByteArrayInputStream(**byte**[] buf) {

**this**.mark = 0;

**this**.buf = buf;

**this**.count = buf.length;

}

**public** ByteArrayInputStream(**byte**[] buf, **int** offset, **int** length) {

**this**.buf = buf;

pos = offset;

mark = offset;

count = offset + length > buf.length ? buf.length : offset + length;

}

@Override

**public** **synchronized** **int** available() {

**return** count - pos;

}

@Override

**public** **void** close() **throws** IOException {

// Do nothing on close, this matches JDK behavior.

}

@Override

**public** **synchronized** **void** mark(**int** readlimit) {

mark = pos;

}

@Override

**public** **boolean** markSupported() {

**return** **true**;

}

@Override

**public** **synchronized** **int** read() {

**return** pos < count ? buf[pos++] & 0xFF : -1;

}

@Override **public** **synchronized** **int** read(**byte**[] buffer, **int** byteOffset, **int** byteCount) {

Arrays.checkOffsetAndCount(buffer.length, byteOffset, byteCount);

// Are there any bytes available?

**if** (**this**.pos >= **this**.count) {

**return** -1;

}

**if** (byteCount == 0) {

**return** 0;

}

**int** copylen = **this**.count - pos < byteCount ? **this**.count - pos : byteCount;

System.*arraycopy*(**this**.buf, pos, buffer, byteOffset, copylen);

pos += copylen;

**return** copylen;

}

@Override

**public** **synchronized** **void** reset() {

pos = mark;

}

@Override

**public** **synchronized** **long** skip(**long** byteCount) {

**if** (byteCount <= 0) {

**return** 0;

}

**int** temp = pos;

pos = **this**.count - pos < byteCount ? **this**.count : (**int**) (pos + byteCount);

**return** pos - temp;

}

}

## DataInputStream

**public** **class** DataInputStream **extends** FilterInputStream **implements** DataInput {

**private** **final** **byte**[] scratch = **new** **byte**[8];

**public** DataInputStream(InputStream in) {

**super**(in);

}

@Override **public** **final** **int** read(**byte**[] buffer) **throws** IOException {

**return** **super**.read(buffer);

}

@Override **public** **final** **int** read(**byte**[] buffer, **int** byteOffset, **int** byteCount) **throws** IOException {

**return** in.read(buffer, byteOffset, byteCount);

}

**public** **final** **boolean** readBoolean() **throws** IOException {

**int** temp = in.read();

**if** (temp < 0) {

**throw** **new** EOFException();

}

**return** temp != 0;

}

**public** **final** **byte** readByte() **throws** IOException {

**int** temp = in.read();

**if** (temp < 0) {

**throw** **new** EOFException();

}

**return** (**byte**) temp;

}

**public** **final** **char** readChar() **throws** IOException {

**return** (**char**) readShort();

}

**public** **final** **double** readDouble() **throws** IOException {

**return** Double.*longBitsToDouble*(readLong());

}

**public** **final** **float** readFloat() **throws** IOException {

**return** Float.*intBitsToFloat*(readInt());

}

**public** **final** **void** readFully(**byte**[] dst) **throws** IOException {

readFully(dst, 0, dst.length);

}

**public** **final** **void** readFully(**byte**[] dst, **int** offset, **int** byteCount) **throws** IOException {

Streams.readFully(in, dst, offset, byteCount);

}

**public** **final** **int** readInt() **throws** IOException {

Streams.readFully(in, scratch, 0, SizeOf.INT);

**return** Memory.peekInt(scratch, 0, ByteOrder.*BIG\_ENDIAN*);

}

@Deprecated

**public** **final** String ~~readLine~~() **throws** IOException {

StringBuilder line = **new** StringBuilder(80); // Typical line length

**boolean** foundTerminator = **false**;

**while** (**true**) {

**int** nextByte = in.read();

**switch** (nextByte) {

**case** -1:

**if** (line.length() == 0 && !foundTerminator) {

**return** **null**;

}

**return** line.toString();

**case** (**byte**) '\r':

**if** (foundTerminator) {

((PushbackInputStream) in).unread(nextByte);

**return** line.toString();

}

foundTerminator = **true**;

/\* Have to be able to peek ahead one byte \*/

**if** (!(in.getClass() == PushbackInputStream.**class**)) {

in = **new** PushbackInputStream(in);

}

**break**;

**case** (**byte**) '\n':

**return** line.toString();

**default**:

**if** (foundTerminator) {

((PushbackInputStream) in).unread(nextByte);

**return** line.toString();

}

line.append((**char**) nextByte);

}

}

}

**public** **final** **long** readLong() **throws** IOException {

Streams.readFully(in, scratch, 0, SizeOf.LONG);

**return** Memory.peekLong(scratch, 0, ByteOrder.*BIG\_ENDIAN*);

}

**public** **final** **short** readShort() **throws** IOException {

Streams.readFully(in, scratch, 0, SizeOf.SHORT);

**return** Memory.peekShort(scratch, 0, ByteOrder.*BIG\_ENDIAN*);

}

**public** **final** **int** readUnsignedByte() **throws** IOException {

**int** temp = in.read();

**if** (temp < 0) {

**throw** **new** EOFException();

}

**return** temp;

}

**public** **final** **int** readUnsignedShort() **throws** IOException {

**return** ((**int**) readShort()) & 0xffff;

}

**public** **final** String readUTF() **throws** IOException {

**return** decodeUTF(readUnsignedShort());

}

String decodeUTF(**int** utfSize) **throws** IOException {

**return** *decodeUTF*(utfSize, **this**);

}

**private** **static** String decodeUTF(**int** utfSize, DataInput in) **throws** IOException {

**byte**[] buf = **new** **byte**[utfSize];

in.readFully(buf, 0, utfSize);

**return** ModifiedUtf8.decode(buf, **new** **char**[utfSize], 0, utfSize);

}

**public** **static** **final** String readUTF(DataInput in) **throws** IOException {

**return** *decodeUTF*(in.readUnsignedShort(), in);

}

**public** **final** **int** skipBytes(**int** count) **throws** IOException {

**int** skipped = 0;

**long** skip;

**while** (skipped < count && (skip = in.skip(count - skipped)) != 0) {

skipped += skip;

}

**return** skipped;

}

}

# Timer

**private** TimerTask mTask;

**private** Timer mTimer;

mTask = **new** TimerTask() {

@Override

**public** **void** run() {

mMainHandler.sendEmptyMessage(*REFRESH\_MESSAGEf*);

}

};

mTimer = **new** Timer("Weibo friendship refresh timer");

mTimer.schedule(mTask, *WEIBO\_REFRESH\_DELAY*, *WEIBO\_REFRESH\_INTERVAL*);