

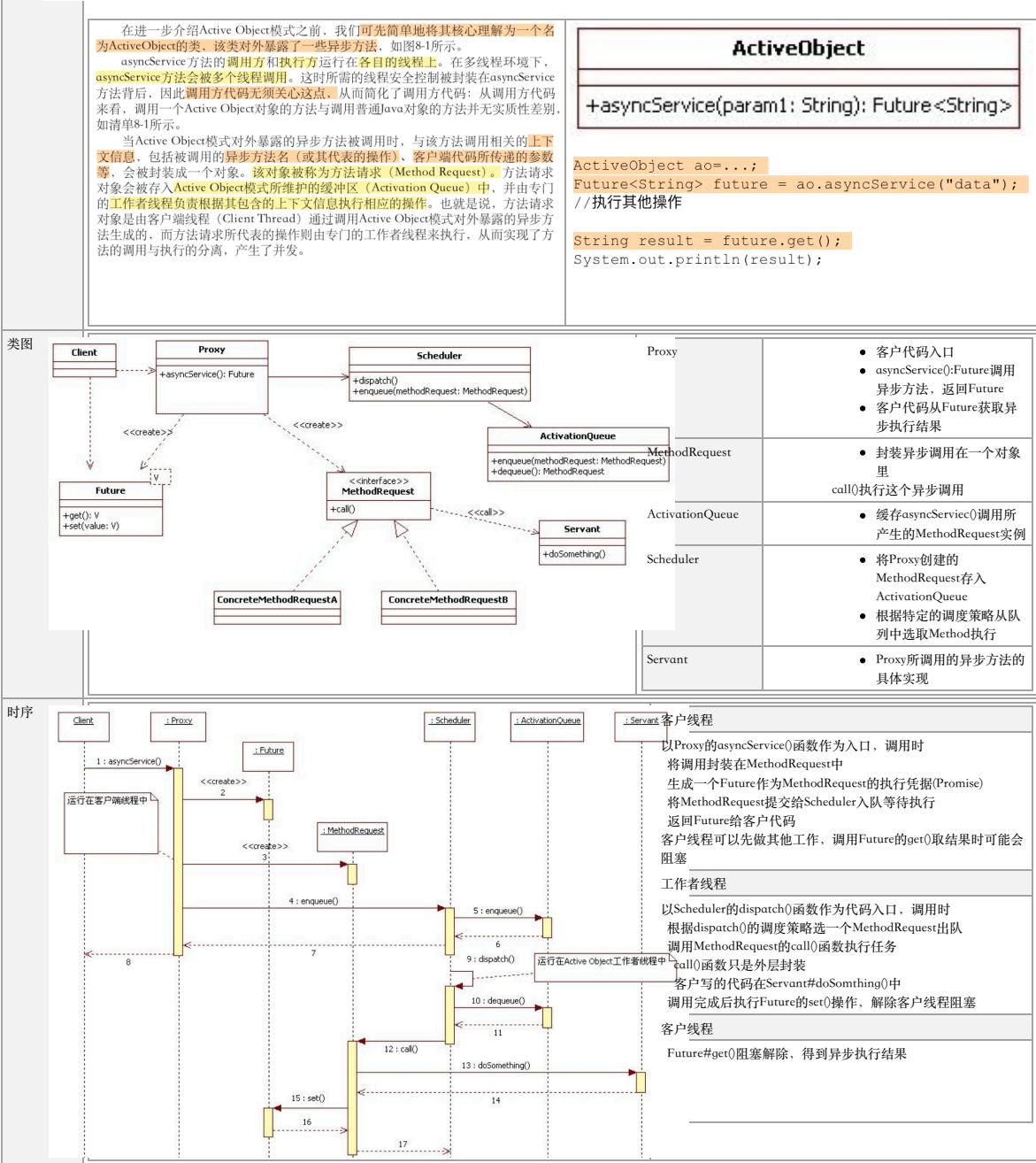
发件人：方堃 fangkun119@icloud.com
主题：CH08 Active Object
日期：2017年5月2日 下午12:00
收件人：

方

CH08 Active Object

2017年4月26日 星期三
下午7:59

用途 Active Object模式是一种异步编程模式。它通过对方法的调用（Method Invocation）与方法的执行（Method Execution）进行解耦（Decoupling）来提高并发性。若以任务的概念来说，Active Object模式的核心则是它允许任务的提交（相当于对异步方法的调用）和任务的执行（相当于异步方法的真正执行）分离。这有点类似于System.gc()这个方法：客户端代码调用完gc()后，一个进行垃圾回收的任务被提交，但此时JVM并不一定进行了垃圾回收，而可能是在gc()方法调用返回后的某段时间才开始执行任务——回收垃圾。我们知道，System.gc()的调用方代码是运行在自己的线程上（通常是main线程派生的子线程），而JVM的垃圾回收这个动作则由专门的工作者线程（垃圾回收线程）来执行。换而言之，System.gc()这个方法所代表的动作（其定义的功能）的调用方和执行方是运行在不同的线程中的，从而提高了并发性。



Active Object模式还有个好处是它可以将任务（MethodRequest）的提交（调用异步方法）和任务的执行策略（Execution Policy）分离。任务的执行策略被封装在Scheduler的实现类之内，因此它对外是“不可见”的，一旦需要变动也不会影响其他代码，从而降低了系统的耦合性。任务的执行策略可以反映以下一些问题。

- 采用什么顺序去执行任务，如FIFO、LIFO，或者基于任务中包含的信息所定的优先级？
- 多少个任务可以并发执行？
- 多少个任务可以被排队等待执行？
- 如果有任务由于系统过载被拒绝，此时哪个任务该被选中作为牺牲品，应用程序该如何被通知到？
- 任务执行前、执行后需要执行哪些操作？

这意味着，任务的执行顺序可以和任务的提交顺序不同，可以采用单线程也可以采用多线程去执行任务等。

当然，好处的背后总是隐藏着代价，Active Object模式实现异步编程也有其代价。该模式的参与者有6个之多，其实现过程也包含了不少中间的处理：MethodRequest对象的生

成、MethodRequest对象的移动（进出缓冲区）、MethodRequest对象的运行调度和线程上下文切换等。这些处理都有其空间和时间的代价。因此，Active Object模式适合于分解一个比较耗时的任务（如涉及I/O操作的任务）：将任务的发起和执行进行分离，以减少不必要的等待时间。

实现要领	错误隔离	<table border="1"> <tr> <td>复用JDK的ThreadPoolExecutor</td><td>已经实现了错误隔离</td></tr> <tr> <td>自己写Scheduler</td><td>必须特别注意run()方法的异常处理，以做到错误隔离</td></tr> </table>	复用JDK的ThreadPoolExecutor	已经实现了错误隔离	自己写Scheduler	必须特别注意run()方法的异常处理，以做到错误隔离
复用JDK的ThreadPoolExecutor	已经实现了错误隔离					
自己写Scheduler	必须特别注意run()方法的异常处理，以做到错误隔离					
<p>错误隔离指一个任务的处理失败不影响其他任务的处理。每个MethodRequest实例可以看作一个任务。那么，Scheduler的实现类在执行MethodRequest时需要注意错误隔离。选用JDK中现成的类（如ThreadPoolExecutor）来实现Scheduler的一个好处就是这些类可能已经实现了错误隔离。而如果自己编写代码实现Scheduler，用单个Active Object工作线程逐一执行所有任务，则需要特别注意线程的run方法的异常处理，确保不会因为个别任务执行时遇到一些运行时异常而导致整个线程终止。如清单8-6所示的示例代码。</p> <pre> public class CustomScheduler implements Runnable { private LinkedBlockingQueue<Runnable> activationQueue = new LinkedBlockingQueue<Runnable>(); @Override public void run() { dispatch(); } public <T> Future<T> enqueue(Callable<T> methodRequest) { final FutureTask<T> task = new FutureTask<T>(methodRequest) @Override public void run() { try { super.run(); //捕获所有可能抛出的对象，避免该任务运行失败而导致其所在的线程终止。 } catch (Throwable t) { this.setException(t); } } ; try { activationQueue.put(task); } catch (InterruptedException e) { } } } </pre>						

缓冲区 如果ActivationQueue是有界缓冲区，则对缓冲区的当前大小进行监控无论是对于运维还是测试来说都有其意义。从测试的角度来看，监控缓冲区有助于确定缓冲区容量的建议值（合理值）。如清单8-3所示的代码，即通过定时任务周期性地调用ThreadPoolExecutor的getQueue方法对缓冲区的大小进行监控。当然，在监控缓冲区的时候，往往只需要大致的值，因此在监控代码中要注意避免不必要的锁。

缓冲区 当任务的提交速率大于任务的执行速率时，缓冲区可能逐渐积压到满。这时新提交的任务会被拒绝。无论是自己编写代码还是利用JDK现有类来实现Scheduler，对于缓冲区满时新任务提交失败，我们需要一个处理策略用于决定此时哪个任务会成为“牺牲品”。若使用ThreadPoolExecutor来实现Scheduler有个好处，是它已经提供了几个缓冲区饱和处理策略的实现代码，应用代码可以直接调用。如清单8-3所示的代码，本章案例中我们选择了在任务的提交方线程中执行被拒绝的任务作为处理策略。

`java.util.concurrent.RejectedExecutionHandler`接口是ThreadPoolExecutor对缓冲区饱和处理策略的抽象，JDK中提供的具体实现类如表8-2所示。

表8-2 JDK提供的缓冲区饱和处理策略实现类

实现类	所实现的处理策略
ThreadPoolExecutor.AbortPolicy	直接抛出异常
ThreadPoolExecutor.DiscardPolicy	丢弃当前被拒绝的任务（而不抛出任何异常）
ThreadPoolExecutor.DiscardOldestPolicy	将缓冲区中最老的任务丢弃，然后重新尝试接纳被拒绝的任务
ThreadPoolExecutor.CallerRunsPolicy	在任务的提交方线程中运行被拒绝的任务

空闲线程 然而Scheduler采用多个工作者线程（如采用ThreadPoolExecutor这样的线程池）来执行任务，则可能需要清理空闲的线程以节约资源。清单8-3的代码就是直接使用了ThreadPoolExecutor的现有功能，在初始化实例时通过指定其构造器的第3、4个参数（long keepAliveTime、TimeUnit unit），告诉ThreadPoolExecutor对于核心工作者线程以外的线程，若已经空闲了指定时间，则将其清理掉。

JDK实现类 `java.util.concurrent.ThreadPoolExecutor`可以看成是Active Object模式的一个通用实现。ThreadPoolExecutor自身相当于Active Object模式的Proxy和Scheduler参与者实例。ThreadPoolExecutor的submit方法相当于Active Object模式对外暴露的异步方法。该方法的唯一参数（`java.util.concurrent.Callable`或者`java.lang.Runnable`）可以看作是MethodRequest参与者实例。该方法的返回值（`java.util.concurrent.Future`）相当于Future参与者实例，而ThreadPoolExecutor的构造方法中需要传入的BlockingQueue实例相当于ActivationQueue参与者实例。

可复用	<pre> /** * Active Object模式Proxy参与者的可复用实现。 * 模式角色: ActiveObject.Proxy * @author Viscent Huang */ public abstract class ActiveObjectProxy { </pre>
	<pre> private static class DispatchInvocationHandler implements InvocationHandler { private final Object delegate; private final ExecutorService scheduler; public DispatchInvocationHandler(Object delegate, ExecutorService executorService) { this.delegate = delegate; this.scheduler = executorService; } private String makeDelegateMethodName(final Method method, final Object[] args) { String name = method.getName(); name = "do" + Character.toUpperCase(name.charAt(0)) + name.substring(1); return name; } @Override public Object invoke(final Object proxy, final Method method, final Object[] args) throws Throwable { Object returnValue = null; final Object delegate = this.delegate; final Method delegateMethod; //如果拦截到的被调用方法是异步方法，则将其转发到相应的doXXX方法 if (Future.class.isAssignableFrom(method.getReturnType())) { delegateMethod = delegate.getClass().getMethod(makeDelegateMethodName(method, args)); </pre>

```

        method.getParameterTypes());
final ExecutorService scheduler = this.scheduler;

Callable<Object> methodRequest = new Callable<Object>() {
    @Override
    public Object call() throws Exception {
        Object rv = null;
        try {
            rv = delegateMethod.invoke(delegate, args);
        } catch (IllegalArgumentException e) {
            throw new Exception(e);
        } catch (IllegalAccessException e) {
            throw new Exception(e);
        } catch (InvocationTargetException e) {
            throw new Exception(e);
        }
        return rv;
    }
};

Future<Object> future = scheduler.submit(methodRequest);
returnValue = future;
} else {

//若拦截到的方法调用不是异步方法，则直接转发

delegateMethod = delegate.getClass()
    .getMethod(method.getName(), method.getParameterTypes());
returnValue = delegateMethod.invoke(delegate, args);
}
return returnValue;
}
}

private static class DispatchInvocationHandler implements InvocationHandler {
private final Object delegate;
private final ExecutorService scheduler;

public DispatchInvocationHandler(Object delegate,
    ExecutorService executorService) {
    this.delegate = delegate;
    this.scheduler = executorService;
}

private String makeDelegateMethodName(final Method method,
    final Object[] arg) {
    String name = method.getName();
    name = "do" + Character.toUpperCase(name.charAt(0))
        + name.substring(1);

    return name;
}

@Override
public Object invoke(final Object proxy, final Method method,
    final Object[] args) throws Throwable {

Object returnValue = null;
final Object delegate = this.delegate;
final Method delegateMethod;

//如果拦截到的被调用方法是异步方法，则将其转发到相应的doXXX方法
if (Future.class.isAssignableFrom(method.getReturnType())) {
    delegateMethod = delegate.getClass().getMethod(
        makeDelegateMethodName(method, args),
        method.getParameterTypes());
    final ExecutorService scheduler = this.scheduler;

Callable<Object> methodRequest = new Callable<Object>() {
    @Override
    public Object call() throws Exception {
        Object rv = null;
        try {
            rv = delegateMethod.invoke(delegate, args);
        } catch (IllegalArgumentException e) {
            throw new Exception(e);
        } catch (IllegalAccessException e) {
            throw new Exception(e);
        } catch (InvocationTargetException e) {
            throw new Exception(e);
        }
        return rv;
    }
};
Future<Object> future = scheduler.submit(methodRequest);
returnValue = future;
} else {

//若拦截到的方法调用不是异步方法，则直接转发

delegateMethod = delegate.getClass()
    .getMethod(method.getName(), method.getParameterTypes());
returnValue = delegateMethod.invoke(delegate, args);
}
return returnValue;
}

/**
 * 生成一个实现指定接口的Active Object proxy实例。
 * 对interf所定义的异步方法的调用会被转发到servant的相应的doXXX方法。
 * @param interf 要实现的Active Object接口
 * @param servant Active Object的Servant参与者实例
 * @param scheduler Active Object的Scheduler参与者实例
 * @return Active Object的Proxy参与者实例
 */
public static <T> T newInstance(Class<T> interf, Object servant,
    ExecutorService scheduler) {
}
}

```

参数
1) 指定Active Object模式对外暴露的接口，该接口作为第1个参数传入。
2) 创建Active Object模式对外暴露的接口的实现类。该类的实例作为第2个参数传入。
3) 指定一个java.util.concurrent.ExecutorService实例。该实例作为第3个参数传入。

```

    @SuppressWarnings("unchecked")
    T f = (T) Proxy.newProxyInstance(interf.getClassLoader(),
        new Class[] { interf }, new DispatchInvocationHandler(servant,
        scheduler));
    return f;
}
}

public static void main(String[] args) throws
InterruptedException, ExecutionException {
    SampleActiveObject sao = ActiveObjectProxy.newInstance(
        SampleActiveObject.class, new SampleActiveObjectImpl(),
        Executors.newCachedThreadPool());
    Future<String> ft = sao.process("Something", 1);
    Thread.sleep(50);
    System.out.println(ft.get());
}
}

```

例子

场景 该模块处理接收到的下发彩信请求的一个关键操作是，查询数据库以获得接收方短号对应的真实号码（长号）。该操作可能因为数据库故障而失败，从而使整个请求无法继续被处理。而数据库故障是可恢复的故障，因此在短号转换为长号的过程中如果出现数据库异常，可以先将整个下发彩信请求消息缓存到磁盘中，等到数据库恢复后，再从磁盘中读取请求消息，进行重试。为方便起见，我们可以通过Java的对象序列化API，将表示下发彩信的对象序列化到磁盘文件中从而实现请求缓存。下面我们将讨论这个请求缓存操作还需要考虑的其他因素，以及Active Object模式如何帮助我们满足这些考虑。

首先，**请求消息缓存到磁盘中涉及文件I/O这种慢的操作，我们不希望它在请求处理的主线程（即Web服务器的工作者线程）中执行**。因为这样会使该模块的响应延时增大，降低系统的响应性，并使得Web服务器的工作者线程因等待文件I/O而降低了系统的吞吐量。这时，异步处理就派上用场了。Active Object模式可以帮助我们实现请求缓存这个任务的提交和执行分离：任务的提交是在Web服务器的工作者线程中完成的，而任务的执行（包括序列化对象到磁盘文件中等操作）则是在Active Object工作者线程中执行的。这样，请求处理的主线程在侦测到短号转长号失败时即可触发对当前彩信下发请求进行缓存，接着继续其请求处理，如给客户端响应。而此时，当前请求消息可能正在被Active Object线程缓存到文件中，如图8-4所示。



图8-4 异步实现缓存

其次，每个短号转长号失败的彩信下发请求消息会被缓存为一个磁盘文件。但我们不希望这些缓存文件被存在同一个子目录下，而是希望多个缓存文件会被存储到多个子目录中。每个子目录最多可以存储指定个数（如2000个）的缓存文件。若当前子目录已存满，则新建一个子目录存放新的缓存文件，直到该子目录也存满，依此类推。当这些子目录的个数到达指定数量（如100个）时，最老的子目录（连同其下的缓存文件，如果有的话）会被删除，从而保证子目录的个数也是固定的。显然，在并发环境下，实现这种控制需要一些并发访问控制（如通过锁来控制），但是我们不希望这种控制暴露给处理请求的其他代码。而Active Object模式中的Proxy参与者可以帮助我们封装并发访问控制。

代码

```

public class MMSDeliveryServlet extends HttpServlet {
    //模式角色: ActiveObject.Proxy
    public class AsyncRequestPersistence implements RequestPersistence {
        private static final long ONE_MINUTE_IN_SECONDS = 60;
        private final Logger logger;
        private final AtomicLong taskTimeConsumedPerInterval = new AtomicLong(0);
        private final AtomicInteger requestSubmittedPerInterval = new AtomicInteger(0);

        //模式角色: ActiveObject.Servant
        private final DiskbasedRequestPersistence delegate = new
        DiskbasedRequestPersistence();

        //模式角色: ActiveObject.Scheduler
        private final ThreadPoolExecutor scheduler;

        //用于保存AsyncRequestPersistence的唯一实例
        private static class InstanceHolder {
            final static RequestPersistence INSTANCE = new AsyncRequestPersistence(); //因为是单例，所以构造函数声明为私有
        }
    }
}

```

触发ActiveObject方法调用：在某些异常场景下，执行一个慢操作应对故障
慢操作给ActiveObject异步执行，避免阻塞主线程：慢操作给ActiveObject异步执行，避免阻塞主线程
提交慢操作请求给ActiveObject：提交慢操作请求给ActiveObject
客户代码收到的原始request类型是MMSDeliveryRequest：客户代码收到的原始request类型是MMSDeliveryRequest
通过AsyncRequestPersistence#store(...)传给框架：通过AsyncRequestPersistence#store(...)传给框架
框架用Callable<boolean>包裹MMSDeliveryRequest，用作RequestMethod角色的实现：框架用Callable<boolean>包裹MMSDeliveryRequest，用作RequestMethod角色的实现
ActiveObject角色：AsyncRequestPersistence：是一单例、Servicant、Scheduler、ActivationQueue
MethodRequest等都封装在内部：MethodRequest等都封装在内部
只暴露getInstance()、store()方法给外部：只暴露getInstance()、store()方法给外部
Servant角色：客户代码提供：客户代码提供
Scheduler角色：复用JDK提供的ThreadPoolExecutor：复用JDK提供的ThreadPoolExecutor
//封装单例：
static class InstanceHolder {
private static RequestPersistence = new AsyncRequestPersistence();
}
...:

<pre>//私有构造器 private AsyncRequestPersistence() { logger = Logger.getLogger(AsyncRequestPersistence.class); scheduler = new ThreadPoolExecutor(1, 3, 60 * ONE_MINUTE_IN_SECONDS, TimeUnit.SECONDS, //模式角色: ActiveObject.ActivationQueue new ArrayBlockingQueue<Runnable>(200), new ThreadFactory() { @Override public Thread newThread(Runnable r) { Thread t; t = new Thread(r, "AsyncRequestPersistence"); return t; } }); scheduler .setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy()); } // 启动队列监控定时任务 Timer monitorTimer = new Timer(true); monitorTimer.scheduleAtFixedRate(new TimerTask() { @Override public void run() { if (logger.isInfoEnabled()) { logger.info("task count:" + requestSubmittedPerInterval + ", Queue size:" + scheduler.getQueue().size() + ", taskTimeConsumedPerInterval:" + taskTimeConsumedPerInterval.get() + " ms"); } taskTimeConsumedPerInterval.set(0); requestSubmittedPerInterval.set(0); } }, 0, ONE_MINUTE_IN_SECONDS * 1000); } </pre>	<pre>scheduler = new ThreadPoolExecutor(1, /*core pool size*/ 3, /*max pool size*/ 60 * ONE_MINUTE_IN_SECONDS, /*keep alive time*/ TimeUnit.SECONDS, /*time unit*/ //工作队列 new ArrayBlockingQueue<Runnable>(200), //线程工厂 new ThreadFactory() { @Override public Thread newThread(Runnable r) { Thread t; t = new Thread(r, "AsyncRequestPersistence"); return t; } } ... //set handler for un-executed tasks scheduler.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy()); ... }</pre> <p>ThreadPoolExecutor.CallerRunsPolicy A handler for rejected tasks that runs the rejected task directly in the calling thread of the execute method, unless the executor has been shut down, in which case the task is discarded.</p>
<pre>//获取类AsyncRequestPersistence的唯一实例 public static RequestPersistence getInstance() { return InstanceHolder.INSTANCE; } @Override public void store(final MMSDeliverRequest request) { /* * 将对store方法的调用封装成MethodRequest对象，并存入缓冲区。 */ //模式角色: ActiveObject.MethodRequest Callable<Boolean> methodRequest = new Callable<Boolean>() { @Override public Boolean call() throws Exception { long start = System.currentTimeMillis(); try { delegate.store(request); } finally { taskTimeConsumedPerInterval.addAndGet(System.currentTimeMillis() - start); } return Boolean.TRUE; } }; scheduler.submit(methodRequest); requestSubmittedPerInterval.incrementAndGet(); } </pre>	<p>MethodRequest角色: 复用IDK的Callable<ResultT>接口 客户代码传入Callable<ResultT>, 传入delegate</p> <p>MMSDeliverRequest, delegate的输入: Callable<boolean>实现为内部类, 因此可以访问调用上下文的MMSDeliverRequest对象 (MMSDeliverRequest是客户代码收到的最原始请求, 通过解析ServletRequest来得到)</p> <p>delegate.doSomething(): 在例子中对应于delegate.store(), 即下面的store()方法 DatabasedRequestPersistence implement RequestPersistence { @override public void store() { ... } ... }</p> <p>传递原始请求给框架。框架将其包裹为Callable<boolean> 存入任务队列供后台线程异步执行</p>
<pre>public interface RequestPersistence { void store(MMSDeliverRequest request); } </pre>	
<p>客户提供的Delegate方法: DatabasedRequestPersistence的store()方法</p> <pre>public class DiskbasedRequestPersistence implements RequestPersistence { 其中SectionBasedDiskStorage代码见电子书179 // 负责缓存文件的存储管理 private final SectionBasedDiskStorage storage = new SectionBasedDiskStorage(); private final Logger logger = Logger.getLogger(DiskbasedRequestPersistence.class); @Override public void store(MMSDeliverRequest request) { // 申请缓存文件的文件名 String[] fileNameParts = storage.apply4Filename(request); File file = new File(fileNameParts[0]); try { ObjectOutputStream objOut = new ObjectOutputStream(new FileOutputStream(file)); try { objOut.writeObject(request); } finally { objOut.close(); } } catch (FileNotFoundException e) { storage.decrementSectionFileCount(fileNameParts[1]); logger.error("Failed to store request", e); } catch (IOException e) { storage.decrementSectionFileCount(fileNameParts[1]); logger.error("Failed to store request", e); } } } </pre>	