# Netty篇

## netty简述

* Netty 是一个基于 JAVA **NIO** 类库的**异步通信框架**，用于创建**异步非阻塞、基于事件驱动、高性能、高可靠性和高可定制性的**网络**客户端**和**服务器端**
* netty通过Reactor模型基于多路复用器接收并处理用户请求，内部实现了两个线程池，boss线程池和work线程池，其中boss线程池的线程负责处理请求的accept事件，当接收到accept事件的请求时，把对应的socket封装到一个NioSocketChannel中，并交给work线程池，其中work线程池负责请求的read和write事件

### Reactor中的组件

* Reactor:Reactor是IO事件的派发者。
* Acceptor:Acceptor接受client连接，建立对应client的Handler，并向Reactor注册此Handler。
* Handler:和一个client通讯的实体，按这样的过程实现业务的处理。一般在基本的Handler基础上还会有更进一步的层次划分， 用来抽象诸如decode，process和encoder这些过程。比如对Web Server而言，decode通常是HTTP请求的解析， process的过程会进一步涉及到Listener和Servlet的调用。业务逻辑的处理在Reactor模式里被分散的IO事件所打破， 所以Handler需要有适当的机制在所需的信息还不全（读到一半）的时候保存上下文，并在下一次IO事件到来的时候（另一半可读了）能继续中断的处理。为了简化设计，Handler通常被设计成状态机，按GoF的state pattern来实现。
* Reactor模型和AWT事件模型很像，就是将消息放到了一个队列中，通过异步线程池对其进行消费
* **特点**
  + **异步、非阻塞、基于事件驱动**的NIO框架
  + 支持**多种传输层通信协议**，包括TCP、UDP等
  + 开发异步HTTP服务端和客户端应用程序
  + 提供对**多种应用层协议**的支持，包括TCP私有协议、HTTP协议、WebSocket协议、文件传输等
  + 默认提供多种编解码能力，包括**Java序列化、Google的ProtoBuf、二进制编解码、Jboss marshalling、文本字符串、base64、简单XML**等，这些编解码框架可以被用户直接使用
  + 提供形式**多样的编解码基础类库**，可以非常方便的实现私有协议栈编解码框架的二次定制和开发
  + 经典的**ChannelFuture-listener**机制，所有的异步IO操作都可以设置listener进行监听和获取操作结果
  + 基于**ChannelPipeline-ChannelHandler**的责任链模式，可以方便的**自定义业务拦截器**用于**业务逻辑定制**
  + 安全性：支持SSL、HTTPS
  + 可靠性：流量整形、读写超时控制机制、缓冲区最大容量限制、资源的优雅释放等
  + **简洁的API和启动辅助类，简化开发难度，减少代码量**
* 为什么**不是传统的IO**
  + **线程模型存在致命缺陷**：一连接一线程的模型导致服务端无法承受大量客户端的并发连接
  + **性能差**：频繁的线程上下文切换导致 CPU 利用效率不高
  + **可靠性差**：由于所有的 IO 操作都是同步的，所以业务线程只要进行 IO 操作，也会存在被同步阻塞的风险，这会导致系统的可靠性差，依赖外部组件的处理能力和网络的情况
* **什么是上下文切换**
* 即使是单核CPU也支持多线程执行代码，CPU通过给每个线程分配CPU时间片来实现这个机制。时间片是CPU分配给各个线程的时间，因为时间片非常短，所以CPU通过不停地切换线程执行，让我们感觉多个线程时同时执行的，时间片一般是几十毫秒（ms）。
* CPU通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再次加载这个任务的状态，**从任务保存到再加载的过程就是一次上下文切换**。
* 这就像我们同时读两本书，当我们在读一本英文的技术书籍时，发现某个单词不认识， 于是便打开中英文词典，但是在放下英文书籍之前，大脑必须先记住这本书读到了多少页的第多少行，等查完单词之后，能够继续读这本书。这样的切换是会影响读 书效率的，同样上下文切换也会影响多线程的执行速度。
* 使用**NIO**，同步阻塞 IO 的**三个缺陷**都将**迎刃而解**
  + **NIO 采用 Reactor 模式，一个 Reactor 线程聚合一个多路复用器 Selector，它可以同时注册、监听和轮询成百上千个 Channel，一个 IO 线程可以同时并发处理N个客户端连接，线程模型优化为1：N（N < 进程可用的最大句柄数）或者 M : N (M通常为 CPU 核数 + 1， N < 进程可用的最大句柄数)**
  + 由于 IO 线程总数有限，不会存在**频繁的 IO 线程之间上下文切换和竞争，CPU 利用率高**
  + 所有的 IO 操作都是异步的，即使业务线程直接进行 IO 操作，也不会被同步阻塞，系统不再依赖外部的网络环境和外部应用程序的处理性能
  + 切换到 NIO 编程之后可以为系统带来巨大的可靠性、性能提升，所以，目前采用 NIO 进行通信已经逐渐成为主流
* 为什么**不直接使用JDK NIO**类库
  + NIO的**类库和API繁杂**，使用麻烦，你需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等
  + 需要具备其它的额外技能做铺垫，例如**熟悉Java多线程**编程，因为NIO编程涉及到Reactor模式，你必须对多线程和网路编程非常熟悉，才能编写出高质量的NIO程序
  + **可靠性能力补齐，工作量和难度都非常大**。例如客户端面临**断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流**的处理等等，NIO编程的特点是功能开发相对容易，但是可靠性能力补齐工作量和难度都非常大
  + JDK NIO的BUG，例如臭名昭著的**epoll bug**，它会导致**Selector空轮询**，最终导致CPU 100%。官方声称在JDK1.6版本的update18修复了该问题，但是直到JDK1.7版本该问题仍旧存在，只不过该bug发生概率降低了一些而已，它并没有被根本解决
  + Netty解决策略:
    - 1) 根据该BUG的特征，首先侦测该BUG是否发生；
    - 2) 将问题Selector上注册的Channel转移到新建的Selector上；
    - 3) 老的问题Selector关闭，使用新建的Selector替换。
  + 一个高性能、高可靠性的 NIO 服务端**开发和维护成本**都是非常**高**的，开发者需要具有丰富的 NIO 编程经验和网络维护经验，很多时候甚至需要通过抓包来定位问题
  + 开发出一套 NIO 程序需要 1 个月，但是它的稳定很可能需要 1 年甚至更长的时间
  + Netty 的**健壮性、功能、性能、可定制性和可扩展性**在同类框架中都是**首屈一指**

## 2. netty原理

* **Netty逻辑架构**
  + **第一层**
    - **Reactor 通信调度层**，它由一系列辅助类组成，包括 Reactor 线程NioEventLoop 以及其父类、NioSocketChannel/NioServerSocketChannel 以及其父类、ByteBuffer 以及由其衍生出来的各种 Buffer、Unsafe 以及其衍生出的各种内部子类等
  + **第二层**
    - **职责链 ChannelPipeLine**，它负责调度事件在职责链中的传播，支持动态的编排职责链，职责链可以选择性的拦截自己关心的事件，对于其它IO操作和事件忽略，Handler同时支持inbound和outbound事件
  + **第三层**
    - **业务逻辑编排层**，业务逻辑编排层通常有两类：一类是纯粹的业务逻辑编排，还有一类是应用层协议插件，用于协议相关的编解码和链路管理，例如 CMPP 协议插件
* **灵拷贝**
  + “零拷贝”是指计算机操作的过程中，**CPU不需要为数据在内存之间的拷贝消耗资源**。而它通常是指计算机在网络上发送文件时，不需要将文件内容拷贝到用户空间（User Space）而**直接在内核空间（Kernel Space）中传输到网络的方式**
* **Netty的“零拷贝”主要体现在三个方面**
  + Netty的**接收和发送ByteBuffer采用DIRECT BUFFERS，使用堆外直接内存进行Socket读写，不需要进行字节缓冲区的二次拷贝**。如果使用传统的堆内存（HEAP BUFFERS）进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中，然后才写入Socket中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝
  + Netty提供了**组合Buffer对象**，可以聚合多个ByteBuffer对象，用户可以**像操作一个Buffer那样方便的对组合Buffer进行操作**，避免了传统通过内存拷贝的方式将几个小Buffer合并成一个大的Buffer
  + Netty的文件传输采用了**transferTo方法**，它可以直接将文件缓冲区的数据发送到目标Channel，避免了传统通过循环write方式导致的内存拷贝问题
* **Reactor线程模型**
  + **Reactor单线程模型**
    - 一个线程中，Acceptor进行请求派发，处理连接请求，验证， 通过Dispatch将对应的ByteBuffer派发到指定的Handler上进行消息解码，进行业务逻辑Handler工作
    - 小容量应用场景，可以使用单线程模型，对于高负载、大并发的应用却不合适
      * 一个NIO线程同时处理成百上千的链路，性能上无法支撑，即便NIO线程的CPU负荷达到100%，也无法满足海量消息的编码、解码、读取和发送
      * 当NIO线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重了NIO线程的负载，最终会导致大量消息积压和处理超时，NIO线程会成为系统的性能瓶颈
      * 可靠性问题：一旦NIO线程意外跑飞，或者进入死循环，会导致整个系统通信模块不可用，不能接收和处理外部消息，造成节点故障
  + **Reactor多线程模型**
    - 一个专门NIO Acceptor线程**监听服务端，接受客户端TCP连接请求**
    - 由一个NIO线程池（ 可以采用标准的JDK线程池实现，它包含一个任务队列和N个可用的线程）进行**IO操作，读写，编码，消息发送接受**
    - 1个NIO线程可以同时处理N条链路，但是1个链路只对应1个NIO线程，防止发生并发操作问题
    - 绝大多数场景下，Reactor多线程模型都可以满足性能需求， 在极特殊应用场景中，一个NIO线程负责监听和处理所有的客户端连接可能会存在性能问题
      * 例如百万客户端并发连接，或者服务端需要对客户端的握手消息进行安全认证，认证本身非常损耗性能
  + **主从Reactor多线程模型**
    - 服务端用于**接收客户端连接**的不再是个1个单独的NIO线程，而是**一个独立的NIO线程池**
    - **Acceptor**接收到**客户端TCP连接请求**处理完成后（可能包含接入认证等），将新创建的SocketChannel注册到IO线程池（sub reactor线程池）的某个IO线程上，由它负责**SocketChannel的读写和编解码工作**
    - **Acceptor线程池**仅仅只用于**客户端的登陆、握手和安全认证**，一旦链路建立成功，就将链路注册到后端subReactor线程池的IO线程上，由IO线程负责后续的IO操作
    - **子线程池**进行**消息接受，发送，编码等业务处理**
  + Netty的线程模型可以通过创建不同的**EventLoopGroup**实例并通过**适当的参数配置**，就可以支持上述**三种Reactor线程模型**
  + Netty 对Reactor线程模型的支持提供了灵活的定制能力，可以满足不同业务场景的性能诉求
* **buffer**
  + **取代JDK NIO的java.nio.ByteBuffer**，相比ByteBuffer
    - 可以根据需要自定义buffer type
    - 内置混合的buffer type，实现zero-copy
    - 提供类似StringBuffer的动态dynamic buffer
    - 不需要调用flip方法
    - 更快的性能

## 3. netty应用场景

* **弹性伸缩的分布式服务架构**
  + 阿里巴巴**Dubbo**内部私有通信协议-dubbo协议默认使用Netty作为高性能异步通信框架，为分布式服务节点之间提供**高性能的NIO客户端和服务端通信**
  + 大众点评服务框架Pigeon和消息中间件Swallow
* **大数据领域**
  + **Hadoop的子系统Apache Avro**
    - 丰富的数据结构
    - 压缩、高效、二进制的序列化框
    - 远程服务调用（RPC）
    - 多语言、灵活的集成能力
* **游戏行业**
  + **高并发**：由于采用异步非阻塞模式，一个Netty游戏服务端可以同时处理成千上万的游戏玩家登陆和在线
  + **高性能**：Netty的性能在各个NIO框架中最高，它的单节点吞吐量非常大，适合海量玩家同时在线游戏
  + **安全性**：支持HTTPS、SSL等，可以在传输层进行安全控制
  + **定制性**：可以方便的实现业务逻辑的定制、游戏编解码的定制，可以方便的与第三方进行集成，例如amf3等

参考: [Netty(RPC高性能之道)](http://blog.csdn.net/zhiguozhu/article/details/50517551)

# IO篇

IO的方式通常分为几种，**同步阻塞的BIO、同步非阻塞的NIO、异步非阻塞的AIO。**

## 一、BIO

在JDK1.4出来之前，我们建立网络连接的时候采用BIO模式，需要先在服务端启动一个ServerSocket，然后在客户端启动Socket来对服务端进行通信，默认情况下服务端需要对每个请求建立一堆线程等待请求，而客户端发送请求后，先咨询服务端是否有线程相应，如果没有则会一直等待或者遭到拒绝请求，如果有的话，客户端会线程会等待请求结束后才继续执行。

## 二、NIO

    NIO本身是基于事件驱动思想来完成的，其主要想解决的是BIO的大并发问题： 在使用同步I/O的网络应用中，如果要同时处理多个客户端请求，或是在客户端要同时和多个服务器进行通讯，就必须使用多线程来处理。也就是说，将每一个客户端请求分配给一个线程来单独处理。这样做虽然可以达到我们的要求，但同时又会带来另外一个问题。由于每创建一个线程，就要为这个线程分配一定的内存空间（也叫工作存储器），而且操作系统本身也对线程的总数有一定的限制。如果客户端的请求过多，服务端程序可能会因为不堪重负而拒绝客户端的请求，甚至服务器可能会因此而瘫痪。

    NIO基于Reactor，当socket有流可读或可写入socket时，操作系统会相应的通知引用程序进行处理，应用再将流读取到缓冲区或写入操作系统。  也就是说，这个时候，已经不是一个连接就要对应一个处理线程了，而是有效的请求，对应一个线程，当连接没有数据时，是没有工作线程来处理的。

   BIO与NIO一个比较重要的不同，是我们使用BIO的时候往往会引入多线程，每个连接一个单独的线程；而NIO则是使用单线程或者只使用少量的多线程，每个连接共用一个线程。

NIO的最重要的地方是当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器上面，所以所有的连接只需要一个线程就可以搞定，当这个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理，也就是一个请求一个线程模式。

      在NIO的处理方式中，当一个请求来的话，开启线程进行处理，可能会等待后端应用的资源(JDBC连接等)，其实这个线程就被阻塞了，当并发上来的话，还是会有BIO一样的问题。

　　HTTP/1.1出现后，有了Http长连接，这样除了超时和指明特定关闭的http header外，这个链接是一直打开的状态的，这样在NIO处理中可以进一步的进化，在后端资源中可以实现资源池或者队列，当请求来的话，开启的线程把请求和请求数据传送给后端资源池或者队列里面就返回，并且在全局的地方保持住这个现场(哪个连接的哪个请求等)，这样前面的线程还是可以去接受其他的请求，而后端的应用的处理只需要执行队列里面的就可以了，这样请求处理和后端应用是异步的.当后端处理完，到全局地方得到现场，产生响应，这个就实现了异步处理。

## 三、AIO

     与NIO不同，当进行读写操作时，只须直接调用API的read或write方法即可。这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入read方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序。  即可以理解为，read/write方法都是异步的，完成后会主动调用回调函数。  在JDK1.7中，这部分内容被称作NIO.2，主要在java.nio.channels包下增加了下面四个异步通道：

* AsynchronousSocketChannel
* AsynchronousServerSocketChannel
* AsynchronousFileChannel
* AsynchronousDatagramChannel

其中的read/write方法，会返回一个带回调函数的对象，当执行完读取/写入操作后，直接调用回调函数。

## 四、总结

**BIO是一个连接一个线程。**

**NIO是一个请求一个线程。**

**AIO是一个有效请求一个线程。**

先来个例子理解一下概念，以银行取款为例：

* 同步 ： 自己亲自出马持银行卡到银行取钱（使用同步IO时，Java自己处理IO读写）；
* 异步 ： 委托一小弟拿银行卡到银行取钱，然后给你（使用异步IO时，Java将IO读写委托给OS处理，需要将数据缓冲区地址和大小传给OS(银行卡和密码)，OS需要支持异步IO操作API）；
* 阻塞 ： ATM排队取款，你只能等待（使用阻塞IO时，Java调用会一直阻塞到读写完成才返回）；
* 非阻塞 ： 柜台取款，取个号，然后坐在椅子上做其它事，等号广播会通知你办理，没到号你就不能去，你可以不断问大堂经理排到了没有，大堂经理如果说还没到你就不能去（使用非阻塞IO时，如果不能读写Java调用会马上返回，当IO事件分发器会通知可读写时再继续进行读写，不断循环直到读写完成）

Java对BIO、NIO、AIO的支持：

* Java BIO ： 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。
* Java NIO ： 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。
* Java AIO(NIO.2) ： 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，

BIO、NIO、AIO适用场景分析:

* BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。
* NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。
* AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

另外，I/O属于底层操作，需要操作系统支持，并发也需要操作系统的支持，所以性能方面不同操作系统差异会比较明显。

在高性能的I/O设计中，有两个比较著名的模式Reactor和Proactor模式，其中Reactor模式用于同步I/O，而Proactor运用于异步I/O操作。

    在比较这两个模式之前，我们首先的搞明白几个概念，什么是阻塞和非阻塞，什么是同步和异步,同步和异步是针对应用程序和内核的交互而言的，同步指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪，而异步是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知。而阻塞和非阻塞是针对于进程在访问数据的时候，根据IO操作的就绪状态来采取的不同方式，说白了是一种读取或者写入操作函数的实现方式，阻塞方式下读取或者写入函数将一直等待，而非阻塞方式下，读取或者写入函数会立即返回一个状态值。

 一般来说I/O模型可以分为：同步阻塞，同步非阻塞，异步阻塞，异步非阻塞IO

**同步阻塞IO**：在此种方式下，用户进程在发起一个IO操作以后，必须等待IO操作的完成，只有当真正完成了IO操作以后，用户进程才能运行。JAVA传统的IO模型属于此种方式！

**同步非阻塞IO**:在此种方式下，用户进程发起一个IO操作以后边可返回做其它事情，但是用户进程需要时不时的询问IO操作是否就绪，这就要求用户进程不停的去询问，从而引入不必要的CPU资源浪费。其中目前JAVA的NIO就属于同步非阻塞IO。

**异步阻塞IO**：此种方式下是指应用发起一个IO操作以后，不等待内核IO操作的完成，等内核完成IO操作以后会通知应用程序，这其实就是同步和异步最关键的区别，同步必须等待或者主动的去询问IO是否完成，那么为什么说是阻塞的呢？因为此时是通过select系统调用来完成的，而select函数本身的实现方式是阻塞的，而采用select函数有个好处就是它可以同时监听多个文件句柄，从而提高系统的并发性！

**异步非阻塞IO**:在此种模式下，用户进程只需要发起一个IO操作然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知，此时用户进程只需要对数据进行处理就好了，不需要进行实际的IO读写操作，因为真正的IO读取或者写入操作已经由内核完成了。

# RPC篇

## Thrift(facebook)

静态数据结构,修改数据结构需重新编译IDL文件. 开发简单. 性能高于GRPC

Thrift序列协议

0.9.3使用BUG:传递Map时如果value为空在service.write报错

## GRPC(google)

支持HTTP2

主要面向移动端

Protobuf序列协议

本身非分布式,需自己实现

15年开源 不够成熟

## DUBBO(Ali)

支持多种协议:http/hessian2/thrift等

## AVRO(hadoop)

主要针对大数据

轻量级框架

动态模式,不生成代码,避免侵入性,利于扩展

不支持协议扩展

# Zookeeper篇

Zookeeper 分布式服务框架是 Apache Hadoop 的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。本文将从使用者角度详细介绍 Zookeeper 的安装和配置文件中各个配置项的意义，以及分析 Zookeeper 的典型的应用场景（配置文件的管理、集群管理、同步锁、Leader 选举、队列管理等），用 Java 实现它们并给出示例代码。

# 消息篇

## RocketMQ

稳定(支付宝集群50台,日处理百亿消息)

定时消费:

回溯消费: 重新将某一个时刻之前的消息重新消费一遍

消息过滤:broker端过滤,节省大量网络传输是否有消息重发造成的重复消费

消息堆积: 就是当缓存消息的内存满了之后的解决方案，一种是丢弃策略，这种不会影响吞吐量，还有一种就是将消息持久化到磁盘，这种会影响吞吐量，在评估影响程度上，RocketMQ的成绩稍微好一点

客户端不在线：RocketMQ可以在客户端上线后继续将未消费的消息推送到客户端

持久化

事务

顺序：我们的消息总线中的消息应该都是无状态的，所以对消息的处理顺序没有严格的要求，如果有特殊要求的话可以在业务层进行控制，activeMQ无法保证严格的顺序，RocketMQ可以保证严格的消费顺序

## ActiveMQ

RocketMQ需要新建一个特殊队列来接收优先级高的队列，无法实现从0-65535这种细粒度的控制，ActiveMQ可以精细控制

## Kafka

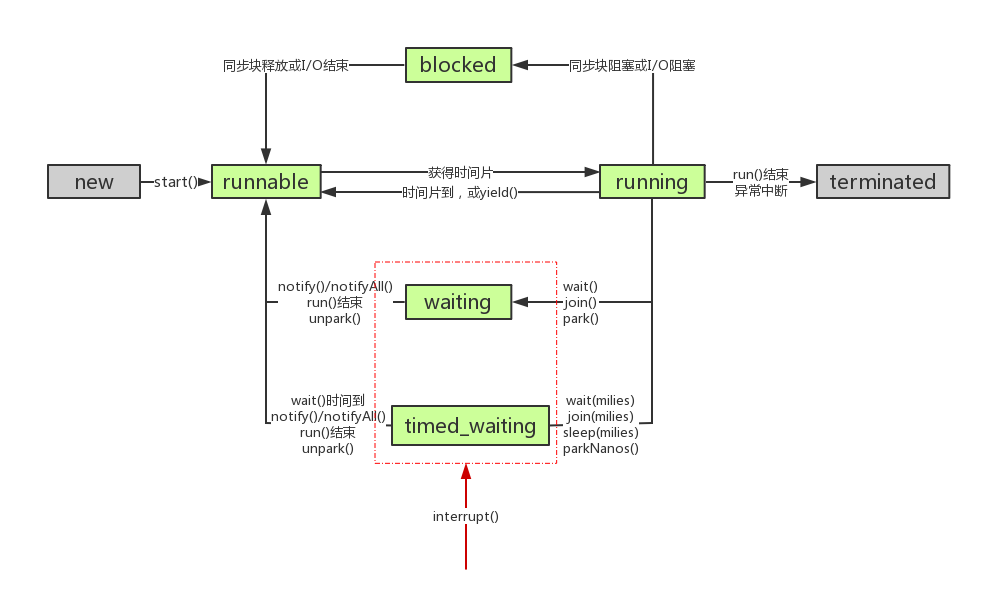
## PriorityBlockingQueue

基于优先级的阻塞队列

# 线程篇

## 基础篇

### 线程状态



public enum State {  
 */\*\*  
 \* 新建  
 \*/  
 NEW*,  
  
 */\*\*  
 \* 就绪  
 \*/  
 RUNNABLE*,  
  
 */\*\*  
 \* 阻塞  
 \* {****@link*** *Object#wait() Object.wait}.  
 \*/  
 BLOCKED*,  
  
 */\*\*  
 \* 等待  
 \** ***@important*** *调用wait()方法引起"当前线程等待",直到另外一个线程调用notify()或notifyAll()唤醒该线程。  
 \* 换句话说，这个方法和wait(0)的效果一样！。  
 \* “当前线程”在调用wait()时，必须拥有该对象的同步锁。该线程调用wait()之后，会释放该锁；  
 \* 然后一直等待直到“其它线程”调用对象的同步锁的notify()或notifyAll()方法。  
 \* 然后，该线程继续等待直到它重新获取“该对象的同步锁”，然后就可以接着运行。  
 \* following methods:  
 \* {****@link*** *Object#wait() Object.wait} with no timeout  
 \* {****@link*** *#join(long) Thread.join} with no timeout  
 \* {****@link*** *LockSupport#park() LockSupport.park}  
 \*/  
 WAITING*,  
  
 */\*\*  
 \* 带等待时间的WAITING  
 \* <ul>  
 \* <li>{****@link*** *#sleep Thread.sleep}</li>  
 \* <li>{****@link*** *Object#wait(long) Object.wait} with timeout</li>  
 \* <li>{****@link*** *#join(long) Thread.join} with timeout</li>  
 \* <li>{****@link*** *LockSupport#parkNanos LockSupport.parkNanos}</li>  
 \* <li>{****@link*** *LockSupport#parkUntil LockSupport.parkUntil}</li>  
 \* </ul>  
 \*/  
 TIMED\_WAITING*,  
  
 */\*\*  
 \* 销毁  
 \*/  
 TERMINATED*;  
}

### 线程启动

*/\*\*  
 \* 启动一个新线程，新线程会执行相应的run()方法。start()不能被重复调用。  
 \* start()实际上是通过本地方法start0()启动线程的。  
 \* 而start0()会新运行一个线程，一旦得到CPU时间片,新线程会调用run()方法。  
 \*/*public synchronized void start() {  
 */\*\*  
 \* 如果线程不是"就绪状态"，则抛出异常！  
 \*/* if (threadStatus != 0)  
 throw new IllegalThreadStateException();  
  
 /\* Notify the group that this thread is about to be started  
 \* so that it can be added to the group's list of threads  
 \* and the group's unstarted count can be decremented. \*/  
 group.add(this);  
  
 boolean started = false;  
 try {  
 start0();  
 started = true;  
 } finally {  
 try {  
 if (!started) {  
 group.threadStartFailed(this);  
 }  
 } catch (Throwable ignore) {  
 /\* do nothing. If start0 threw a Throwable then  
 it will be passed up the call stack \*/  
 }  
 }  
}  
  
@Override  
public void run() {  
 if (target != null) {  
 target.run();  
 }  
}  
  
*/\*\*  
 \* 调用JNI接口  
 \*/*private native void start0();  
*/\*\*  
 \*获取当前执行线程  
 \*/*public static native Thread currentThread();

### wait()/notify()

Object中的wait(), notify()等函数，和synchronized一样，会对“对象的同步锁”进行操作。

wait()会使“**当前线程**”等待，因为线程进入等待状态，所以线程应该释放它所持有的“同步锁”，否则其它线程获取不到该“同步锁”而无法运行！  
OK，线程调用wait()之后，会释放它锁持有的“同步锁”；而且，根据前面的介绍，我们知道：等待线程可以被notify()或notifyAll()唤醒。现在，请思考一个问题：notify()是依据什么唤醒等待线程的？或者说，wait()等待线程和notify()之间是通过什么关联起来的？答案是：依据“对象的同步锁”。

负责唤醒等待线程的那个线程(我们称为“**唤醒线程**”)，它只有在获取“该对象的同步锁”(**这里的同步锁必须和等待线程的同步锁是同一个**)，并且调用notify()或notifyAll()方法之后，才能唤醒等待线程。虽然，等待线程被唤醒；但是，它不能立刻执行，因为唤醒线程还持有“该对象的同步锁”。必须等到唤醒线程释放了“对象的同步锁”之后，等待线程才能获取到“对象的同步锁”进而继续运行。

总之，notify(), wait()依赖于“同步锁”，而“同步锁”是对象锁持有，并且每个对象有且仅有一个！这就是为什么notify(), wait()等函数定义在Object类，而不是Thread类中的原因。

 使用wait()、notify()和notifyAll()时需要先对调用对象加锁，调用wait()方法后会释放锁。

 调用wait()方法之后，线程状态由RUNNING变为WAITING，并将当前线程放置到对象的等待队列中。

 notify()或notifyAll()方法调用后，等待线程不会立刻从wait()中返回，需要等该线程释放锁之后，才有机会获取锁之后从wait()返回。

 notify()方法将等待队列中的一个等待线程从等待队列中移动到同步队列中；notifyAll()方法则是把等待队列中的所有线程都移动到同步队列中；被移动的线程状态从WAITING变为BLOCKED。

 从wait()方法返回的前提是，改线程获得了调用对象的锁。

**阻塞与等待的区别**

阻塞：当一个线程试图获取对象锁（非java.util.concurrent库中的锁，即synchronized），而该锁被其他线程持有，则该线程进入阻塞状态。它的特点是**使用简单，由JVM调度器来决定唤醒自己，而不需要由另一个线程来显式唤醒自己，不响应中断**。  
等待：当一个线程等待另一个线程通知调度器一个条件时，该线程进入等待状态。它的特点是**需要等待另一个线程显式地唤醒自己，实现灵活，语义更丰富，可响应中断**。例如调用：Object.wait()、Thread.join()以及等待Lock或Condition。

　　需要强调的是虽然synchronized和JUC里的Lock都实现锁的功能，但线程进入的状态是不一样的。**synchronized会让线程进入阻塞态，而JUC里的Lock是用LockSupport.park()/unpark()来实现阻塞/唤醒的，会让线程进入等待态**。但话又说回来，虽然等锁时进入的状态不一样，但被唤醒后又都进入runnable态，从行为效果来看又是一样的。

源码:

*/\*\*  
 \** ***@important*** *调用wait()方法引起"***当前线程等待***",直到另外一个线程调用notify()或notifyAll()唤醒该线程。  
 \* 换句话说，这个方法和wait(0)的效果一样！。  
 \* “当前线程”在调用wait()时，必须拥有该对象的同步锁。该线程调用wait()之后，会释放该锁；  
 \* 然后一直等待直到“其它线程”调用对象的同步锁的notify()或notifyAll()方法。  
 \* 然后，该线程继续等待直到它重新获取“该对象的同步锁”，然后就可以接着运行。  
 \*/*public final void wait() throws InterruptedException {  
 wait(0);  
}  
public final native void wait(long timeout) throws InterruptedException;  
public final native void notify();  
public final native void notifyAll();

### yield()/sleep()/join()/interrupt()

* 1. yield()的作用是让步。它能让当前线程由“运行状态”进入到“就绪状态”，从而让其它具有相同优先级的等待线程获取执行权；但是，并不能保证在当前线程调用yield()之后， 其它具有相同优先级的线程就一定能获得执行权；也有可能是当前线程又进入到“运行状态”继续运行！

public static native void yield();

* 1. 使当前线程休眠，即当前线程会从“运行状态”进入到“休眠(阻塞)状态”。sleep()会指定休眠时间，线程休眠的时间会大于/等于该休眠时间；在线程重新被唤醒时， 它会由“阻塞状态”变成“就绪状态”，从而等待cpu的调度执行。不会释放锁

public static native void sleep(long millis) throws InterruptedException;

* 1. 让“主线程”等待“子线程”结束之后才能继续运行。wait()的作用是让“当前线程”等待，而这里的“当前线程”是指当前在CPU上运行的线程。所以，虽然是调用子线程的wait()方法，但是它是通过“主线程”去调用的；所以，休眠的是主线程，而不是“子线程”！

public final synchronized void join(long millis) throws InterruptedException {  
 long base = System.*currentTimeMillis*();  
 long now = 0;  
  
 if (millis < 0) {  
 throw new IllegalArgumentException("timeout value is negative");  
 }  
  
 if (millis == 0) {  
 while (isAlive()) {  
 wait(0);  
 }  
 } else {  
 while (isAlive()) {  
 long delay = millis - now;  
 if (delay <= 0) {  
 break;  
 }  
 wait(delay);  
 now = System.*currentTimeMillis*() - base;  
 }  
 }  
}

* 1. interrupt()的作用是中断本线程。本线程中断自己是被允许的；其它线程调用本线程的interrupt()方法时，会通过checkAccess()检查权限。这有可能抛出SecurityException异常。如果本线程是处于阻塞状态：调用线程的wait(), wait(long)或wait(long, int)会让它进入等待(阻塞)状态， 或者调用线程的join(), join(long), join(long, int), sleep(long), sleep(long, int)也会让它进入阻塞状态。若线程在阻塞状态时，调用了它的interrupt()方法，那么它的“中断状态”会被清除并且会收到一个InterruptedException异常。例如，线程通过wait()进入阻塞状态，此时通过interrupt()中断该线程；调用interrupt()会立即将线程的中断标记设为“true”，但是由于线程处于阻塞状态，所以该“中断标记”会立即被清除为“false”，同时，会产生一个InterruptedException的异常。如果线程被阻塞在一个Selector选择器中，那么通过interrupt()中断它时；线程的中断标记会被设置为true，并且它会立即从选择操作中返回。如果不属于前面所说的情况，那么通过interrupt()中断线程时，它的中断标记会被设置为“true”。中断一个“已终止的线程”不会产生任何操作。

interrupted() 和 isInterrupted()都能够用于检测对象的“中断标记”。

1.interrupted 是作用于当前线程，isInterrupted 是作用于调用该方法的线程对象所对应的线程。（线程对象对应的线程不一定是当前运行的线程。例如我们可以在A线程中去调用B线程对象的isInterrupted方法。）

2.interrupted()会清除中断标记,只有当前线程才能清除自己的标记位

interrupt（）是用来设置中断状态的。返回true说明中断状态被设置了而不是被清除了。我们调用sleep、wait等此类可中断（throw InterruptedException）方法时，一旦方法抛出InterruptedException，当前调用该方法的线程的中断状态就会被jvm自动清除了，就是说我们调用该线程的isInterrupted 方法时是返回false。如果你想保持中断状态，可以再次调用interrupt方法设置中断状态。这样做的原因是，java的中断并不是真正的中断线程，而只设置标志位（中断位）来通知用户。如果你捕获到中断异常，说明当前线程已经被中断，不需要继续保持中断位。  
interrupted是静态方法，返回的是当前线程的中断状态。例如，如果当前线程被中断（没有抛出中断异常，否则中断状态就会被清除），你调用interrupted方法，第一次会返回true。然后，当前线程的中断状态被方法内部清除了。第二次调用时就会返回false。如果你刚开始一直调用isInterrupted，则会一直返回true，除非中间线程的中断状态被其他操作清除了。

public void interrupt() {  
 if (this != Thread.*currentThread*())  
 checkAccess();  
  
 synchronized (blockerLock) {  
 Interruptible b = blocker;  
 if (b != null) {  
 interrupt0(); // Just to set the interrupt flag  
 b.interrupt(this);  
 return;  
 }  
 }  
 interrupt0();  
}

public static boolean interrupted() {  
 return *currentThread*().isInterrupted(true);  
}  
public boolean isInterrupted() {  
 return isInterrupted(false);  
}

* 1. 其他JNI方法

/\* Some private helper methods \*/  
private native void setPriority0(int newPriority);  
private native void stop0(Object o);  
private native void suspend0();  
private native void resume0();  
private native void interrupt0();  
private native void setNativeName(String name);  
public final native boolean isAlive();

### Thread.stop()

**1. 即刻抛出ThreadDeath异常，在线程的run()方法内，任何一点都有可能抛出ThreadDeath Error，包括在catch或finally语句中。**

**2. 会释放该线程所持有的所有的锁，而这种释放是不可控制的，非预期的。**

**Java中多线程锁释放的条件：**

1）执行完同步代码块，就会释放锁。（synchronized）  
2）在执行同步代码块的过程中，遇到异常而导致线程终止，锁也会被释放。（exception）  
3）在执行同步代码块的过程中，执行了锁所属对象的wait()方法，这个线程会释放锁，进入对象的等待池。(wait)

从上面的三点我就可以看到stop方法释放锁是在第二点的，通过抛出异常来释放锁，释放该线程所持有的所有的锁。一般任何进行加锁的代码块，都是为了保护数据的一致性，如果在调用thread.stop()后导致了该线程所持有的所有锁的突然释放(不可控制)，那么被保护数据就有可能呈现不一致性，其他线程在使用这些被破坏的数据时，有可能导致一些很奇怪的应用程序错误。这种方式是不安全的，不可靠的。

### 线程优先级和守护线程

java 中的线程优先级的范围是1～10，默认的优先级是5。“高优先级线程”会优先于“低优先级线程”执行。

java 中有两种线程：**用户线程**和**守护线程**。可以通过isDaemon()方法来区别它们：如果返回false，则说明该线程是“用户线程”；否则就是“守护线程”。  
用户线程一般用户执行用户级任务，而守护线程也就是“后台线程”，一般用来执行后台任务。需要注意的是：Java虚拟机在“用户线程”都结束后会后退出。

JDK 中关于线程优先级和守护线程的介绍如下：

每个线程都有一个优先级。“高优先级线程”会优先于“低优先级线程”执行。每个线程都可以被标记为一个守护进程或非守护进程。在一些运行的主线程中创建新的子线程时，子线程的优先级被设置为等于“创建它的主线程的优先级”，当且仅当“创建它的主线程是守护线程”时“子线程才会是守护线程”。

当Java虚拟机启动时，通常有一个单一的非守护线程（该线程通过是通过main()方法启动）。JVM会一直运行直到下面的任意一个条件发生，JVM就会终止运行：

(01) 调用了exit()方法，并且exit()有权限被正常执行。

(02) 所有的“非守护线程”都死了(即JVM中仅仅只有“守护线程”)。

每一个线程都被标记为“守护线程”或“用户线程”。当只有守护线程运行时，JVM会自动退出。

//最小优先级  
public final static int *MIN\_PRIORITY* = 1;  
//默认优先级  
public final static int *NORM\_PRIORITY* = 5;  
//最大优先级  
public final static int *MAX\_PRIORITY* = 10;  
  
public final void setPriority(int newPriority) {  
 ThreadGroup g;  
 checkAccess();  
 if (newPriority > *MAX\_PRIORITY* || newPriority < *MIN\_PRIORITY*) {  
 throw new IllegalArgumentException();  
 }  
 if((g = getThreadGroup()) != null) {  
 if (newPriority > g.getMaxPriority()) {  
 newPriority = g.getMaxPriority();  
 }  
 setPriority0(priority = newPriority);  
 }  
}

cpu在执行t1和t2的时候，根据时间片轮循调度，所以能够并发执行。

public final void setDaemon(boolean on) {  
 checkAccess();  
 if (isAlive()) {  
 throw new IllegalThreadStateException();  
 }  
 daemon = on;  
}

(01) 主线程main是用户线程，它创建的子线程t1也是用户线程。  
(02) t2是守护线程。在“主线程main”和“子线程t1”(它们都是用户线程)执行完毕，只剩t2这个守护线程的时候，JVM自动退出。

### 生产者消费者问题

### Synchronized

* 1. **原理**

**在java中，每一个对象有且仅有一个同步锁。这也意味着，同步锁是依赖于对象而存在。**  
**当我们调用某对象的synchronized方法时，就获取了该对象的同步锁。**例如，synchronized(obj)就获取了“obj这个对象”的同步锁。  
**不同线程对同步锁的访问是互斥的。**也就是说，某时间点，对象的同步锁只能被一个线程获取到！通过同步锁，我们就能在多线程中，实现对“对象/方法”的互斥访问。 例如，现在有两个线程A和线程B，它们都会访问“对象obj的同步锁”。假设，在某一时刻，线程A获取到“obj的同步锁”并在执行一些操作；而此时，线程B也企图获取“obj的同步锁” —— 线程B会获取失败，它必须等待，直到线程A释放了“该对象的同步锁”之后线程B才能获取到“obj的同步锁”从而才可以运行。

* 1. **基本规则**

[第一条](http://www.cnblogs.com/skywang12345/p/3479202.html#a21): 当一个线程访问“某对象”的“synchronized方法”或者“synchronized代码块”时，其他线程对**“该对象”的该“synchronized方法”或者“synchronized代码块”的访问**将被阻塞。  
第二条: 当一个线程访问“某对象”的“synchronized方法”或者“synchronized代码块”时，其他线程仍然**可以访问“该对象”的非同步代码块**。  
第三条: 当一个线程访问“某对象”的“synchronized方法”或者“synchronized代码块”时，其他线程对**“该对象”的其他的“synchronized方法”或者“synchronized代码块”的访问**将被阻塞。

* 1. **实例锁和全局锁**

**实例锁**-- 锁在某一个实例对象上。**如果该类是单例，那么该锁也具有全局锁的概念**。实例锁对应的就是synchronized关键字。  
**全局锁** -- 该锁针对的是类，无论实例多少个对象，那么线程都共享该锁。全局锁对应的就是static synchronized（或者是锁在该类的class或者classloader对象上）。

* 1. **内存可见性**

线程获取锁时,JMM会吧该线程对应的本地内存置为无效,从而使得被监视器保护的临界区代码必须从主内存中读取共享变量值

线程释放锁时,JMM把该线程对应的工作内存中的数据刷新到主内存中

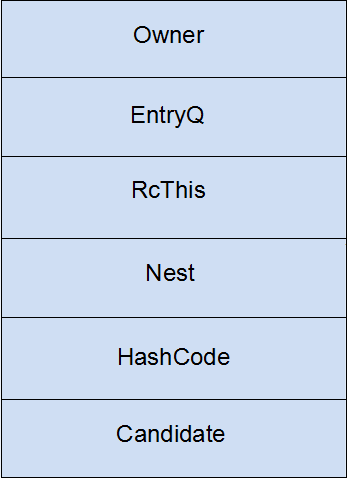
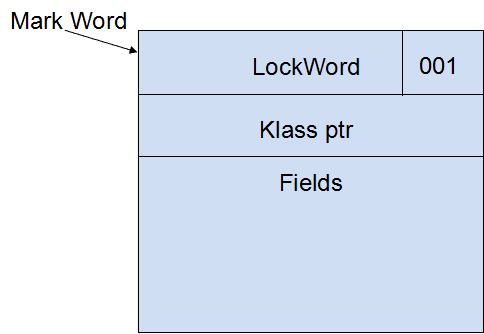
* 1. **个人理解**

同步代码使用**monitoenter**和**monitorexit**指令实现

同步方法使用**ACC\_SYNCHRONIZED** 实现

本质上都是对指定对象相关联的monitor的获取.

线程一旦进入被synchronized修饰的方法或代码块,指定锁对象将对象头中的lockword与monitor关联起来,monitor中的owner存放拥有该锁的线程的唯一标识,确保一次只有一个线程拥有代码的执行权.



对象头 **monitor**

### 死锁

* + - * 1. **概念**

所谓死锁,就是多个线程循环等待它方占用的资源而无限期的僵持下去的局面

* + - * 1. **产生死锁的必要条件**
* 〈1〉互斥条件。即某个资源在一段时间内只能由一个进程占有，不能同时被两个或两个以上的进程占有。这种独占资源如CD-ROM驱动器，打印机等等，必须在占有该资源的进程主动释放它之后，其它进程才能占有该资源。这是由资源本身的属性所决定的。如独木桥就是一种独占资源，两方的人不能同时过桥。
* 〈2〉不可抢占条件。进程所获得的资源在未使用完毕之前，资源申请者不能强行地从资源占有者手中夺取资源，而只能由该资源的占有者进程自行释放。如过独木桥的人不能强迫对方后退，也不能非法地将对方推下桥，必须是桥上的人自己过桥后空出桥面（即主动释放占有资源），对方的人才能过桥。
* 〈3〉占有且申请条件。进程至少已经占有一个资源，但又申请新的资源；由于该资源已被另外进程占有，此时该进程阻塞；但是，它在等待新资源之时，仍继续占用已占有的资源。还以过独木桥为例，甲乙两人在桥上相遇。甲走过一段桥面（即占有了一些资源），还需要走其余的桥面（申请新的资源），但那部分桥面被乙占有（乙走过一段桥面）。甲过不去，前进不能，又不后退；乙也处于同样的状况。
* 〈4〉循环等待条件。存在一个进程等待序列{P1，P2，...，Pn}，其中P1等待P2所占有的某一资源，P2等待P3所占有的某一源，......，而Pn等待P1所占有的的某一资源，形成一个进程循环等待环。就像前面的过独木桥问题，甲等待乙占有的桥面，而乙又等待甲占有的桥面，从而彼此循环等待。

## JUC篇(jdk1.8)

### 原子类框架

1. **基本类型**: AtomicInteger, AtomicLong, AtomicBoolean ;  
2. **数组类型**: AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray ;  
3. **引用类型**: AtomicReference, AtomicStampedRerence, AtomicMarkableReference ;  
4. **对象的属性修改类型**: AtomicIntegerFieldUpdater, AtomicLongFieldUpdater, AtomicReferenceFieldUpdater 。

**5.累加器:** DoubleAccumulator、DoubleAdder、LongAccumulator、LongAdder、Striped64

### CAS

1. **介绍**

CAS,compare and swap比较并替换, **CAS有三个操作数：内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做并返回false**。

1. **源码分析(AtomicInteger)**

public class AtomicInteger extends Number implements java.io.Serializable {  
 private static final long *serialVersionUID* = 6214790243416807050L;  
  
 // setup to use Unsafe.compareAndSwapInt for updates  
 private static final Unsafe *unsafe* = Unsafe.*getUnsafe*();  
 //表示变量值在内存中的偏移地址  
 private static final long *valueOffset*;  
  
 static {  
 try {  
 *valueOffset* = *unsafe*.objectFieldOffset  
 (java.util.concurrent.atomic.AtomicInteger.class.getDeclaredField("value"));  
 } catch (Exception ex) {  
 throw new Error(ex);  
 }  
 }  
  
 private volatile int value;  
  
 */\*\*  
 \* Creates a new AtomicInteger with the given initial value.  
 \*  
 \** ***@param*** *initialValue the initial value  
 \*/* public AtomicInteger(int initialValue) {  
 value = initialValue;  
 }  
  
 */\*\*  
 \* Creates a new AtomicInteger with initial value {****@code*** *0}.  
 \*/* public AtomicInteger() {  
 }  
 public final int incrementAndGet() {  
 return *unsafe*.getAndAddInt(this, *valueOffset*, 1) + 1;  
 }  
  
}

 Unsafe是CAS的核心类，Java无法直接访问底层操作系统，而是通过本地（native）方法来访问。不过尽管如此，JVM还是开了一个后门，JDK中有一个类Unsafe，它提供了硬件级别的**原子操作**。

 valueOffset表示的是变量值在内存中的偏移地址，因为Unsafe就是根据内存偏移地址获取数据的原值的, 偏移量可以简单理解为指针指向该变量的内存地址。

 value是用volatile修饰的，保证了多线程之间看到的value值是同一份。

**Unsafe源码(jdk1.8)**:

*/\*\*  
 \* Atomically adds the given value to the current value of a field  
 \* or array element within the given object <code>o</code>  
 \* at the given <code>offset</code>.  
 \*  
 \** ***@param*** *o object/array to update the field/element in  
 \** ***@param*** *offset field/element offset  
 \** ***@param*** *delta the value to add  
 \** ***@return*** *the previous value  
 \** ***@since*** *1.8  
 \*/*public final int getAndAddInt(Object o, long offset, int delta) {  
 int v;  
 do {  
 v = getIntVolatile(o, offset);  
 } while (!compareAndSwapInt(o, offset, v, v + delta));  
 return v;  
}

假设现在线程A和线程B同时执行getAndAdd操作：

1. AtomicInteger里面的value原始值为3，即主内存中AtomicInteger的value为3，根据Java内存模型，线程A和线程B各自持有一份value的副本，值为3。
2. 线程A通过getIntVolatile(var1, var2)方法获取到value值3，线程切换，线程A挂起。
3. 线程B通过getIntVolatile(var1, var2)方法获取到value值3，并利用compareAndSwapInt方法比较内存值也为3，比较成功，修改内存值为2，线程切换，线程B挂起。
4. 线程A恢复，利用compareAndSwapInt方法比较，发手里的值3和内存值4不一致，**此时value正在被另外一个线程修改，线程A不能修改value值**。
5. 线程的compareAndSwapInt实现，循环判断，重新获取value值，因为value是volatile变量，所以线程对它的修改，线程A总是能够看到。线程A继续利用compareAndSwapInt进行比较并替换，直到compareAndSwapInt修改成功返回true。
6. **CAS缺点**

CAS存在一个很明显的问题，即ABA问题。  
如果变量V初次读取的时候是A，并且在准备赋值的时候检查到它仍然是A，那能说明它的值没有被其他线程修改过了吗？如果在这段期间它的值曾经被改成了B，然后又改回A，那CAS操作就会误认为它从来没有被修改过。针对这种情况，java并发包中提供了一个带有标记的原子引用类"AtomicStampedReference"，它可以通过控制变量值的版本来保证CAS的正确性。

### JUC锁

#### 框架



#### AQS ( AbstractQueuedSynchronizer )

##### 3.2.1 概述

抽象的队列式的同步器，AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch...。

说明：AbstractQueuedSynchronizer类底层的数据结构是使用双向链表，是队列的一种实现，故也可看成是队列，其中Sync queue，即同步队列，是双向链表，包括head结点和tail结点，head结点主要用作后续的调度。而Condition queue不是必须的，其是一个单向链表，只有当使用Condition时，才会存在此单向链表。并且可能会有多个Condition queue

Node源码:

static final class Node {  
 */\*\*  
 \* Marker to indicate a node is waiting in shared mode  
 \*/* static final Node *SHARED* = new Node();  
 */\*\*  
 \* Marker to indicate a node is waiting in exclusive mode  
 \*/* static final Node *EXCLUSIVE* = null;  
 //取消  
 static final int *CANCELLED* = 1;  
 //等待触发  
 static final int *SIGNAL* = -1;  
 //等待条件  
 static final int *CONDITION* = -2;  
 //状态需要向后传播  
 static final int *PROPAGATE* = -3;  
  
 volatile int waitStatus;  
 volatile Node prev;  
 volatile Node next;  
 volatile Thread thread;  
 Node nextWaiter;  
}

###### 3.2.1 等待状态

① CANCELLED，值为1，表示当前的线程被取消。

② SIGNAL，值为-1，表示当前节点的后继节点包含的线程需要运行，需要进行unpark操作。

一般发生情况是：当前线程的后继线程处于阻塞状态，而当前线程被release或cancel掉，因此需要唤醒当前线程的后继线程。

　　③ CONDITION，值为-2，表示当前节点在等待condition，也就是在condition queue中。

　　④ PROPAGATE，值为-3，(共享锁) 表示当前场景下后续的acquireShared能够得以执行。

　　⑤ 值为0，表示当前节点在sync queue中，等待着获取锁。



###### 3.2.2 数据结构





说明：AbstractQueuedSynchronizer类底层的数据结构是使用双向链表，是队列的一种实现，故也可看成是队列，其中Sync queue，即同步队列，是双向链表，包括head结点和tail结点，head结点主要用作后续的调度。而Condition queue不是必须的，其是一个单向链表，只有当使用Condition时，才会存在此单向链表。并且可能会有多个Condition queue。

AQS定义两种资源共享方式：Exclusive（独占，只有一个线程能执行，如ReentrantLock）和Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）。

不同的自定义同步器争用共享资源的方式也不同。**自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可**，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

* isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
* tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
* tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
* tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
* tryReleaseShared(int)：共享方式。尝试释放资源，成功则返回true，失败则返回false。

##### 3.2.2源码详解

###### 3.2.2.1 acquire(int)

此方法是独占模式下线程获取共享资源的顶层入口。如果获取到资源，线程直接返回，否则进入等待队列，直到获取到资源为止，且整个过程忽略中断的影响,即线程在acquire过程中,中断此线程是无效的。这也正是lock()的语义，当然不仅仅只限于lock()。获取到资源后，线程就可以去执行其临界区代码了。源码：

public final void acquire(int arg) {  
 if (!tryAcquire(arg) &&  
 acquireQueued(addWaiter(Node.*EXCLUSIVE*), arg))  
 *selfInterrupt*();  
}

函数流程如下：

tryAcquire()尝试直接去获取资源，如果成功则直接返回；

addWaiter()将该线程加入等待队列的尾部，并标记为独占模式；

acquireQueued()使线程在等待队列中获取资源，一直获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。

如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上。



tryAcquire()

尝试直接去获取资源，如果成功则直接返回；

protected boolean tryAcquire(int arg) {  
 throw new UnsupportedOperationException();  
}

具体资源获取/释放方式交由自定义同步器实现

ReentrantLock中公平锁和非公平锁的实现源码:

//公平锁  
protected final boolean tryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {  
 if (!hasQueuedPredecessors() &&  
 compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0)  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 return false;  
}  
//非公平锁  
final boolean nonfairTryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {  
 if (compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0) // overflow  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 return false;  
}

addWaiter()

将该线程加入等待队列的尾部，并标记为独占模式；

private Node addWaiter(Node mode) {  
 //以给定模式构造节点.mode有两种：EXCLUSIVE（独占）和SHARED（共享）  
 Node node = new Node(Thread.*currentThread*(), mode);  
 //尝试快速方式直接放到队尾。  
 Node pred = tail;  
 if (pred != null) {  
 node.prev = pred;  
 if (compareAndSetTail(pred, node)) {  
 pred.next = node;  
 return node;  
 }  
 }  
 //上一步失败则通过enq入队。  
 enq(node);  
 return node;  
}  
private Node enq(final Node node) {  
 //CAS"自旋"，直到成功加入队尾  
 for (;;) {  
 Node t = tail;  
 if (t == null) { // 队列为空，创建一个空的标志结点作为head结点，并将tail也指向它。  
 if (compareAndSetHead(new Node()))  
 tail = head;  
 } else {//正常流程，放入队尾  
 node.prev = t;  
 if (compareAndSetTail(t, node)) {  
 t.next = node;  
 return t;  
 }  
 }  
 }  
}

acquireQueued()

使线程在等待队列中获取资源，一直获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。具体流程:

1.结点进入队尾后，检查状态，找到安全休息点；

2.调用park()进入waiting状态，等待unpark()或interrupt()唤醒自己；

3.被唤醒后，看自己是不是有资格能拿到号。如果拿到，head指向当前结点，并返回从入队到拿到号的整个过程中是否被中断过；如果没拿到，继续流程1。

源码:

final boolean acquireQueued(final Node node, int arg) {  
 boolean failed = true;//标记是否成功拿到资源  
 try {  
 boolean interrupted = false;//标记等待过程中是否被中断过  
 //自旋  
 for (; ; ) {  
 final Node p = node.predecessor();//拿到前节点  
 //如果前节点是head，即该结点已成next，那么便有资格去尝试获取资源（可能是head释放完资源唤醒自己的，当然也可能被interrupt了）。  
 if (p == head && tryAcquire(arg)) {  
 setHead(node);//拿到资源后，将head指向该结点。所以head所指的标杆结点，就是当前获取到资源的那个结点或null。  
 p.next = null; // help GC  
 failed = false;  
 return interrupted;//返回等待过程中是否被中断过  
 }  
 //如果自己可以休息了，就进入waiting状态，直到被unpark()  
 if (*shouldParkAfterFailedAcquire*(p, node) &&  
 parkAndCheckInterrupt())  
 interrupted = true;//如果等待过程中被中断过，就将interrupted标记为true  
 }  
 } finally {  
 if (failed)  
 cancelAcquire(node);  
 }  
}  
  
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {  
 int ws = pred.waitStatus;//拿到前节点的状态  
 if (ws == Node.*SIGNAL*)  
 //如果已经告诉前节点拿完号后通知自己一下，那就可以安心休息了  
 return true;  
 if (ws > 0) {  
 */\*\*  
 \* 如果前节点放弃了，那就一直往前找，直到找到最近一个正常等待的状态，并排在它的后边。  
 \* 注意：那些放弃的结点，由于被自己“加塞”到它们前边，它们相当于形成一个无引用链，稍后就会被GC回收  
 \*/* do {  
 node.prev = pred = pred.prev;  
 } while (pred.waitStatus > 0);  
 pred.next = node;  
 } else {  
 //如果前节点正常，那就把前节点的状态设置成SIGNAL，告诉它拿完号后通知自己一下。有可能失败，人家说不定刚刚释放完呢  
 compareAndSetWaitStatus(pred, ws, Node.*SIGNAL*);  
 }  
 return false;  
}  
*/\*\*如果线程找好安全休息点后，那就可以安心去休息了。此方法就是让线程去休息，真正进入等待状态。\*/*private final boolean parkAndCheckInterrupt() {  
 LockSupport.*park*(this);//使线程进入waiting状态  
 return Thread.*interrupted*();//如果被唤醒，查看自己是不是被中断的。  
}

park()会让当前线程进入waiting状态。在此状态下，有两种途径可以唤醒该线程：

1）被unpark()；“前继节点对应的线程”使用完锁之后，通过unpark()方式唤醒当前线程。

2）被interrupt()。需要注意的是，Thread.interrupted()会清除当前线程的中断标记位。

如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上。

###### 3.2.2.2 release(int)

此方法是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。这也正是unlock()的语义，当然不仅仅只限于unlock()。下面是release()的源码：

public final boolean release(int arg) {  
 if (tryRelease(arg)) {  
 Node h = head;//找到头节点  
 if (h != null && h.waitStatus != 0)  
 unparkSuccessor(h);//唤醒等待队列的下一个线程  
 return true;  
 }  
 return false;  
}

tryRelease(int)

此方法尝试去释放指定量的资源。下面是tryRelease()的源码：

protected boolean tryRelease(int arg) {  
 throw new UnsupportedOperationException();  
}

跟tryAcquire()一样，这个方法是需要独占模式的自定义同步器去实现的。正常来说，tryRelease()都会成功的，因为这是独占模式，该线程来释放资源，那么它肯定已经拿到独占资源了，直接减掉相应量的资源即可(state-=arg)，也不需要考虑线程安全的问题。但要注意它的返回值，**release()是根据tryRelease()的返回值来判断该线程是否已经完成释放掉资源了！**所以自义定同步器在实现时，如果已经彻底释放资源(state=0)，要返回true，否则返回false

unparkSuccessor(Node)

说明: 唤醒等待队列里的下一个线程

private void unparkSuccessor(Node node) {  
 //这里，node一般为当前线程所在的结点。  
 int ws = node.waitStatus;  
 if (ws < 0)//置零当前线程所在的结点状态，允许失败。  
 compareAndSetWaitStatus(node, ws, 0);  
  
 Node s = node.next;//找到下一个需要唤醒的结点  
 if (s == null || s.waitStatus > 0) {  
 s = null;  
 for (Node t = tail; t != null && t != node; t = t.prev)  
 if (t.waitStatus <= 0)  
 s = t;  
 }  
 if (s != null)  
 LockSupport.*unpark*(s.thread);  
}

小结

release()是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。

###### 3.2.2.3 acquireShared(int)

此方法是共享模式下线程获取共享资源的顶层入口。它会获取指定量的资源，获取成功则直接返回，获取失败则进入等待队列，直到获取到资源为止，整个过程忽略中断。源码：

public final void acquireShared(int arg) {  
 if (tryAcquireShared(arg) < 0)  
 doAcquireShared(arg);  
}

这里tryAcquireShared()依然需要自定义同步器去实现。但是AQS已经把其返回值的语义定义好了：负值代表获取失败；0代表获取成功，但没有剩余资源；正数表示获取成功，还有剩余资源，其他线程还可以去获取。所以这里acquireShared()的流程就是：

* + 1. tryAcquireShared()尝试获取资源，成功则直接返回；
    2. 失败则通过doAcquireShared()进入等待队列park()，直到被unpark()/interrupt并成功获取到资源为止才返回。整个等待过程也是忽略中断的

tryAcquiredShared(int)

tryAcquireShared()在ReentrantReadWriteLock中实现源码(详细解析见[ReadWriteLock篇 tryAcquireShared(int)](#_tryAcquireShared(int))):

protected final int tryAcquireShared(int unused) {  
   
 Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (exclusiveCount(c) != 0 &&  
 getExclusiveOwnerThread() != current)  
 return -1;  
 int r = sharedCount(c);  
 if (!readerShouldBlock() &&  
 r < MAX\_COUNT &&  
 compareAndSetState(c, c + SHARED\_UNIT)) {  
 if (r == 0) {  
 firstReader = current;  
 firstReaderHoldCount = 1;  
 } else if (firstReader == current) {  
 firstReaderHoldCount++;  
 } else {  
 HoldCounter rh = cachedHoldCounter;  
 if (rh == null || rh.tid != getThreadId(current))  
 cachedHoldCounter = rh = readHolds.get();  
 else if (rh.count == 0)  
 readHolds.set(rh);  
 rh.count++;  
 }  
 return 1;  
 }  
 return fullTryAcquireShared(current);  
}

doAcquireShared(int)

此方法用于将当前线程加入等待队列尾部休息，直到其他线程释放资源唤醒自己，自己成功拿到相应量的资源后才返回。源码：

private void doAcquireShared(int arg) {  
 final Node node = addWaiter(Node.*SHARED*);//加入队列尾部  
 boolean failed = true;//是否成功标记  
 try {  
 boolean interrupted = false;//等待过程中是否被中断过标记  
 for (;;) {  
 final Node p = node.predecessor();//前节点  
 if (p == head) {//如果前节点为head,此时唤醒node  
 int r = tryAcquireShared(arg);//尝试获取资源  
 if (r >= 0) {  
 setHeadAndPropagate(node, r);//将head指向自己,如果还有剩余资源可以再唤醒之后的线程  
 p.next = null; // help GC  
 if (interrupted)//如果等待过程中被中断,此时将中断标记补上  
 selfInterrupt();  
 failed = false;  
 return;  
 }  
 }  
 //判断状态,寻找安全点,进入waiting状态,等待被unpark()或interrupt()  
 if (*shouldParkAfterFailedAcquire*(p, node) &&  
 parkAndCheckInterrupt())  
 interrupted = true;  
 }  
 } finally {  
 if (failed)  
 cancelAcquire(node);  
 }  
}

setHeadAndPropagate(Node,int)

成功获取资源后,如果还有剩余资源,继续唤醒下一个线程

private void setHeadAndPropagate(Node node, int propagate) {  
 Node h = head; // Record old head for check below  
 setHead(node);//head指向自己  
 //如果还有剩余资源,继续唤醒下一个线程  
 if (propagate > 0 || h == null || h.waitStatus < 0 ||  
 (h = head) == null || h.waitStatus < 0) {  
 Node s = node.next;  
 if (s == null || s.isShared())  
 doReleaseShared();  
 }  
}

###### 3.2.2.4 releaseShared(int)

此方法是共享模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。源码:

public final boolean releaseShared(int arg) {  
 if (tryReleaseShared(arg)) {  
 doReleaseShared();//唤醒后续节点  
 return true;  
 }  
 return false;  
}

释放掉资源后，唤醒后继线程 .跟独占模式下的release()相似，但有一点稍微需要注意：独占模式下的tryRelease()在完全释放掉资源（state=0）后，才会返回true去唤醒其他线程，这主要是基于可重入的考量；而共享模式下的releaseShared()则没有这种要求，一是共享的实质--多线程可并发执行；二是共享模式基本也不会重入，所以自定义同步器可以根据需要决定返回值。

tryReleaseShared(int)

tryReleaseShared()在ReentrantReadWriteLock中的实现[见ReadWriteLock篇 tryReleaseShared(int)](#_tryReleaseShared(int)):

protected final boolean tryReleaseShared(int unused) {  
 Thread current = Thread.*currentThread*();  
 if (firstReader == current) {  
 // assert firstReaderHoldCount > 0;  
 if (firstReaderHoldCount == 1)  
 firstReader = null;  
 else  
 firstReaderHoldCount--;  
 } else {  
 HoldCounter rh = cachedHoldCounter;  
 if (rh == null || rh.tid != *getThreadId*(current))  
 rh = readHolds.get();  
 int count = rh.count;  
 if (count <= 1) {  
 readHolds.remove();  
 if (count <= 0)  
 throw unmatchedUnlockException();  
 }  
 --rh.count;  
 }  
 for (;;) {  
 int c = getState();  
 int nextc = c - *SHARED\_UNIT*;  
 if (compareAndSetState(c, nextc))  
 // Releasing the read lock has no effect on readers,  
 // but it may allow waiting writers to proceed if  
 // both read and write locks are now free.  
 return nextc == 0;  
 }  
}

doReleaseShared(int)

主要用于唤醒下一个线程,源码:

private void doReleaseShared() {  
 //自旋  
 for (;;) {  
 Node h = head;  
 if (h != null && h != tail) {  
 int ws = h.waitStatus;  
 if (ws == Node.*SIGNAL*) {  
 if (!compareAndSetWaitStatus(h, Node.*SIGNAL*, 0))  
 continue; // loop to recheck cases  
 unparkSuccessor(h);//唤醒后续线程  
 }  
 else if (ws == 0 &&  
 !compareAndSetWaitStatus(h, 0, Node.*PROPAGATE*))  
 continue; // loop on failed CAS  
 }  
 if (h == head) // loop if head changed  
 break;  
 }  
}

###### 3.2.2.5 其他方法

//独占式超时获取锁  
public final boolean tryAcquireNanos(int arg, long nanosTimeout)  
//共享式超时获取锁   
public final boolean tryAcquireSharedNanos(int arg, long nanosTimeout)

如果未在指定时间内获取锁,返回false

##### 3.2.3 小结

acquire()和acquireSahred()两种方法下，线程在等待队列中都是忽略中断的。AQS也支持响应中断的，acquireInterruptibly()/acquireSharedInterruptibly()即是，这里相应的源码跟acquire()和acquireSahred()差不多，这里就不再详解了。

##### 3.2.4 应用

不同的自定义同步器争用共享资源的方式也不同。**自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可**，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

* isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
* tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
* tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
* tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
* tryReleaseShared(int)：共享方式。尝试释放资源，成功则返回true，失败则返回false。
* **在AQS中维护着一个FIFO的同步队列，当线程获取同步状态失败后，则会加入到这个CLH同步队列的对尾并一直保持着自旋。在CLH同步队列中的线程在自旋时会判断其前驱节点是否为首节点，如果为首节点则不断尝试获取同步状态，获取成功则退出CLH同步队列。当线程执行完逻辑后，会释放同步状态，释放后会唤醒其后继节点。**

#### ReetrantLock

##### 概述

ReentrantLock是一个可重入的互斥锁，又被称为“独占锁”。

顾名思义，ReentrantLock锁在同一个时间点只能被一个线程锁持有；而可重入的意思是，ReentrantLock锁，可以被单个线程多次获取。  
ReentrantLock分为“**公平锁**”和“**非公平锁**”。它们的区别体现在获取锁的机制上是否公平。“锁”是为了保护竞争资源，防止多个线程同时操作线程而出错，ReentrantLock在同一个时间点只能被一个线程获取(当某线程获取到“锁”时，其它线程就必须等待)；ReentraantLock是通过一个FIFO的等待队列来管理获取该锁所有线程的。在“公平锁”的机制下，线程依次排队获取锁；而“非公平锁”在锁是可获取状态时，不管自己是不是在队列的开头都会获取锁。

##### 继承关系及内部类

ReentrantLock实现了Lock接口，Lock接口中定义了lock与unlock相关操作，并且还存在newCondition方法，表示生成一个条件。源码:

public class ReentrantLock extends AbstractQueuedSynchronizer implements Lock, java.io.Serializable



##### Lock.lock()

###### 3.3.3.1 公平锁和非公平锁

公平锁 -- 公平锁的lock()函数，会直接调用acquire(1)。

非公平锁 -- 非公平锁会先判断当前锁的状态是不是空闲，是的话，就不排队，而是直接获取锁。

源码:

//非公平锁获取  
final void lock() {  
 if (compareAndSetState(0, 1))  
 setExclusiveOwnerThread(Thread.*currentThread*());  
 else  
 acquire(1);  
}  
//公平锁获取  
final void lock() {  
 acquire(1);  
}

acquire()源码:

public final void acquire(int arg) {  
 if (!tryAcquire(arg) &&  
 acquireQueued(addWaiter(Node.*EXCLUSIVE*), arg))  
 selfInterrupt();  
}

tryAcquire()源码对比:

//公平锁  
protected final boolean tryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {  
 if (!hasQueuedPredecessors() &&  
 compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0)  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 return false;  
}  
//非公平锁  
final boolean nonfairTryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {  
 if (compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0) // overflow  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 return false;  
}

###### 3.3.3.2“公平锁”和“非公平锁”关于acquire()的对比

公平锁和非公平锁，它们尝试获取锁的方式不同。

公平锁在尝试获取锁时，即使“锁”没有被任何线程锁持有，它也会判断自己是不是CLH等待队列的表头；是的话，才获取锁。

而非公平锁在尝试获取锁时，如果“锁”没有被任何线程持有，则不管它在CLH队列的何处，它都直接获取锁。



##### Lock.unlock()

public void unlock() {  
 sync.release(1);  
}

protected final boolean tryRelease(int releases) {  
 int c = getState() - releases;  
 if (Thread.*currentThread*() != getExclusiveOwnerThread())  
 throw new IllegalMonitorStateException();  
 boolean free = false;  
 if (c == 0) {  
 free = true;  
 setExclusiveOwnerThread(null);  
 }  
 setState(c);  
 return free;  
}

[见3.2.2.2 release(int)](#_3.2.2.2_release(int))

##### Condition条件

###### 概述

Condition的作用是对锁进行更精确的控制。Condition中的await()方法相当于Object的wait()方法，Condition中的signal()方法相当于Object的notify()方法，Condition中的signalAll()相当于Object的notifyAll()方法。不同的是，Object中的wait(),notify(),notifyAll()方法是和"同步锁"(synchronized关键字)捆绑使用的；而Condition是需要与"互斥锁"/"共享锁"捆绑使用的。

###### 函数列表

// 造成当前线程在接到信号或被中断之前一直处于等待状态。  
void await()  
// 造成当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。  
boolean await(long time, TimeUnit unit)  
// 造成当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。  
long awaitNanos(long nanosTimeout)  
// 造成当前线程在接到信号之前一直处于等待状态。  
void awaitUninterruptibly()  
// 造成当前线程在接到信号、被中断或到达指定最后期限之前一直处于等待状态。  
boolean awaitUntil(Date deadline)  
// 唤醒一个等待线程。  
void signal()  
// 唤醒所有等待线程。  
void signalAll()

#### LockSupport

##### 概述

* + - 1. LockSupport是用来创建锁和其他同步类的基本线程阻塞原语。
      2. LockSupport中的park() 和 unpark() 的作用分别是阻塞线程和解除阻塞线程，而且park()和unpark()不会遇到“Thread.suspend 和 Thread.resume所可能引发的死锁”问题。  
         因为park() 和 unpark()有许可的存在；调用 park() 的线程和另一个试图将其 unpark() 的线程之间的竞争将保持活性。
      3. park和wait的区别。wait让线程阻塞前，必须通过synchronized获取同步锁。

##### 源码解析

// 返回提供给最近一次尚未解除阻塞的 park 方法调用的 blocker 对象，如果该调用不受阻塞，则返回 null。  
static Object getBlocker(Thread t)  
// 为了线程调度，禁用当前线程，除非许可可用。  
static void park()  
// 为了线程调度，在许可可用之前禁用当前线程。  
static void park(Object blocker)  
// 为了线程调度禁用当前线程，最多等待指定的等待时间，除非许可可用。  
static void parkNanos(long nanos)  
// 为了线程调度，在许可可用前禁用当前线程，并最多等待指定的等待时间。  
static void parkNanos(Object blocker, long nanos)  
// 为了线程调度，在指定的时限前禁用当前线程，除非许可可用。  
static void parkUntil(long deadline)  
// 为了线程调度，在指定的时限前禁用当前线程，除非许可可用。  
static void parkUntil(Object blocker, long deadline)  
// 如果给定线程的许可尚不可用，则使其可用。  
static void unpark(Thread thread)

说明:所有方法均是调用UNSAFE类的native接口实现

#### ReadWriteLock

##### 概述

Java的JUC(java.util.concurrent)包中的锁包括"独占锁(ReentrantLock)"和"共享锁"。JUC中的共享锁有CountDownLatch, CyclicBarrier, Semaphore, ReentrantReadWriteLock,JDK1.8新增的StampedLock等

ReadWriteLock，顾名思义，是读写锁。它维护了一对相关的锁 — — “读取锁”和“写入锁”，一个用于读取操作，另一个用于写入操作。  
“读取锁”用于只读操作，它是“共享锁”，能同时被多个线程获取。  
“写入锁”用于写入操作，它是“独占锁”，写入锁只能被一个线程锁获取。  
注意：不能同时存在读取锁和写入锁,持有写入锁的线程可以继续获取读取锁,称为锁降级！  
ReadWriteLock是一个接口。ReentrantReadWriteLock是它的实现类，ReentrantReadWriteLock包括子类ReadLock和WriteLock。

源码:

public interface ReadWriteLock {  
 */\*\*  
 \* 读锁,共享锁  
 \*/* Lock readLock();  
  
 */\*\*  
 \* 写锁,独占锁  
 \*/* Lock writeLock();  
}

###### 数据结构



从中可以看出：

1. ReentrantReadWriteLock实现了ReadWriteLock接口。ReadWriteLock是一个读写锁的接口，提供了"获取读锁的readLock()函数" 和 "获取写锁的writeLock()函数"。
2. ReentrantReadWriteLock中包含：sync对象，读锁readerLock和写锁writerLock。读锁ReadLock和写锁WriteLock都实现了Lock接口。读锁ReadLock和写锁WriteLock中也都分别包含了"Sync对象"，它们的Sync对象和ReentrantReadWriteLock的Sync对象 是一样的，就是通过sync，读锁和写锁实现了对同一个对象的访问。
3. 和"ReentrantLock"一样，sync是Sync类型；而且，Sync也是一个继承于AQS的抽象类。Sync也包括"公平锁"FairSync和"非公平锁"NonfairSync。sync对象是"FairSync"和"NonfairSync"中的一个，默认是"NonfairSync"。

Sync

abstract static class Sync extends AbstractQueuedSynchronizer {  
 private static final long *serialVersionUID* = 6317671515068378041L;  
 // 最多支持65535个写锁和65535个读锁；低16位表示写锁计数，高16位表示持有读锁的线程数  
 static final int *SHARED\_SHIFT* = 16;  
 // 由于读锁用高位部分，读锁个数加1，其实是状态值加 2^16  
 static final int *SHARED\_UNIT* = (1 << *SHARED\_SHIFT*);  
 // 读锁最大数量  
 static final int *MAX\_COUNT* = (1 << *SHARED\_SHIFT*) - 1;  
 // 写锁的掩码，用于状态的低16位有效值  
 static final int *EXCLUSIVE\_MASK* = (1 << *SHARED\_SHIFT*) - 1;  
 */\*\* 读锁计数，当前持有读锁的线程数，c的高16位 \*/* static int sharedCount(int c) { return c >>> *SHARED\_SHIFT*; }  
 */\*\* 写锁的计数，也就是它的重入次数,c的低16位\*/* static int exclusiveCount(int c) { return c & *EXCLUSIVE\_MASK*; }  
  
 //表示占有读锁的线程数量  
 static int sharedCount(int c) { return c >>> *SHARED\_SHIFT*; }  
 //表示占有写锁的线程数量  
 static int exclusiveCount(int c) { return c & *EXCLUSIVE\_MASK*; }  
 */\*\*  
 \* 当前线程持有的可重入读锁的数量，仅在构造方法和readObject(反序列化)  
 \* 时被初始化，当持有锁的数量为0时，移除此对象。  
 \*/* private transient ThreadLocalHoldCounter readHolds;  
 */\*\*  
 \* 最近一个成功获取读锁的线程的计数。这省却了ThreadLocal查找，  
 \* 通常情况下，下一个释放线程是最后一个获取线程。这不是 volatile 的，  
 \* 因为它仅用于试探的，线程进行缓存也是可以的  
 \* （因为判断是否是当前线程是通过线程id来比较的）。  
 \*/* private transient HoldCounter cachedHoldCounter;  
 // 第一个读线程  
 private transient Thread firstReader = null;  
 // 第一个读线程的计数  
 private transient int firstReaderHoldCount;  
}  
// 构造函数  
Sync() {  
 // 本地线程计数器  
 readHolds = new ThreadLocalHoldCounter();  
 // 设置AQS的状态  
 setState(getState()); // ensures visibility of readHolds  
}

Sync类内部存在两个内部类，分别为HoldCounter和ThreadLocalHoldCounter，其中HoldCounter主要与读锁配套使用

HoldCounter

// 计数器  
static final class HoldCounter {  
 // 计数  
 int count = 0;  
 // Use id, not reference, to avoid garbage retention  
 // 获取当前线程的TID属性的值  
 final long tid = getThreadId(Thread.*currentThread*());  
}

说明：HoldCounter主要有两个属性，count和tid，其中count表示某个读线程重入的次数，tid表示该线程的tid字段的值，该字段可以用来唯一标识一个线程。

ThreadLocalHoldCounter

// 本地线程计数器  
static final class ThreadLocalHoldCounter  
 extends ThreadLocal<HoldCounter> {  
 // 重写初始化方法，在没有进行set的情况下，获取的都是该HoldCounter值  
 public HoldCounter initialValue() {  
 return new HoldCounter();  
 }  
}

说明：ThreadLocalHoldCounter重写了ThreadLocal的initialValue方法，ThreadLocal类可以将线程与对象相关联。在没有进行set的情况下，get到的均是initialValue方法里面生成的那个HolderCounter对象。

###### 函数列表

// 创建一个新的 ReentrantReadWriteLock，默认是采用“非公平策略”。  
ReentrantReadWriteLock()  
// 创建一个新的 ReentrantReadWriteLock，fair是“公平策略”。fair为true，意味着公平策略；否则，意味着非公平策略。  
ReentrantReadWriteLock(boolean fair)  
  
// 返回当前拥有写入锁的线程，如果没有这样的线程，则返回 null。  
protected Thread getOwner()  
// 返回一个 collection，它包含可能正在等待获取读取锁的线程。  
protected Collection<Thread> getQueuedReaderThreads()  
// 返回一个 collection，它包含可能正在等待获取读取或写入锁的线程。  
protected Collection<Thread> getQueuedThreads()  
// 返回一个 collection，它包含可能正在等待获取写入锁的线程。  
protected Collection<Thread> getQueuedWriterThreads()  
// 返回等待获取读取或写入锁的线程估计数目。  
int getQueueLength()  
// 查询当前线程在此锁上保持的重入读取锁数量。  
int getReadHoldCount()  
// 查询为此锁保持的读取锁数量。  
int getReadLockCount()  
// 返回一个 collection，它包含可能正在等待与写入锁相关的给定条件的那些线程。  
protected Collection<Thread> getWaitingThreads(Condition condition)  
// 返回正等待与写入锁相关的给定条件的线程估计数目。  
int getWaitQueueLength(Condition condition)  
// 查询当前线程在此锁上保持的重入写入锁数量。  
int getWriteHoldCount()  
// 查询是否给定线程正在等待获取读取或写入锁。  
boolean hasQueuedThread(Thread thread)  
// 查询是否所有的线程正在等待获取读取或写入锁。  
boolean hasQueuedThreads()  
// 查询是否有些线程正在等待与写入锁有关的给定条件。  
boolean hasWaiters(Condition condition)  
// 如果此锁将公平性设置为 ture，则返回 true。  
boolean isFair()  
// 查询是否某个线程保持了写入锁。  
boolean isWriteLocked()  
// 查询当前线程是否保持了写入锁。  
boolean isWriteLockedByCurrentThread()  
// 返回用于读取操作的锁。  
ReentrantReadWriteLock.ReadLock readLock()  
// 返回用于写入操作的锁。  
ReentrantReadWriteLock.WriteLock writeLock()

##### 3.5.2 ReadLock

###### 3.5.2.1 源码

public static class ReadLock implements Lock, java.io.Serializable {  
 private static final long *serialVersionUID* = -5992448646407690164L;  
 //持有的AQS对象  
 private final Sync sync;  
  
 protected ReadLock(ReentrantReadWriteLock lock) {  
 sync = lock.sync;  
 }  
  
 //获取共享锁  
 public void lock() {  
 sync.acquireShared(1);  
 }  
  
 //获取共享锁(响应中断)  
 public void lockInterruptibly() throws InterruptedException {  
 sync.acquireSharedInterruptibly(1);  
 }  
  
 //尝试获取共享锁  
 public boolean tryLock(long timeout, TimeUnit unit)  
 throws InterruptedException {  
 return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));  
 }  
  
 //释放锁  
 public void unlock() {  
 sync.releaseShared(1);  
 }  
  
 //新建条件  
 public Condition newCondition() {  
 throw new UnsupportedOperationException();  
 }  
  
 public String toString() {  
 int r = sync.getReadLockCount();  
 return super.toString() +  
 "[Read locks = " + r + "]";  
 }  
}

###### 3.5.2.2 lock()

public void lock() {  
 sync.acquireShared(1);  
}

acquireShared(int)

[见AQS篇 acquireShared(int)](#_3.2.2.3_acquireShared(int))

tryAcquireShared(int)

protected final int tryAcquireShared(int unused) {  
 //获取当前线程  
 Thread current = Thread.*currentThread*();  
 int c = getState();  
 //持有写锁的线程可以获取读锁，如果获取锁的线程不是current线程；则返回-1。  
 if (exclusiveCount(c) != 0 &&  
 getExclusiveOwnerThread() != current)  
 return -1;  
 //获取共享计数  
 int r = sharedCount(c);  
 // 如果“不需要阻塞”，并且“读取锁”的共享计数小于MAX\_COUNT  
 // 则通过CAS函数更新“锁的状态”，将“读取锁”的共享计数+1  
 if (!readerShouldBlock() &&  
 r < MAX\_COUNT &&  
 compareAndSetState(c, c + SHARED\_UNIT)) {  
 // 第1次获取“共享锁”  
 if (r == 0) {  
 firstReader = current;  
 firstReaderHoldCount = 1;  
 //如果想要获取锁的线程(current)是第1个获取锁(firstReader)的线程  
 } else if (firstReader == current) {  
 firstReaderHoldCount++;  
 } else {  
 // HoldCounter用来统计该线程获取“读取锁”的次数。  
 HoldCounter rh = cachedHoldCounter;  
 if (rh == null || rh.tid != getThreadId(current))  
 cachedHoldCounter = rh = readHolds.get();  
 else if (rh.count == 0)  
 readHolds.set(rh);  
 // 将该线程获取“读取锁”的次数+1。  
 rh.count++;  
 }  
 return 1;  
 }  
 return fullTryAcquireShared(current);  
}

**说明**：tryAcquireShared()的作用是尝试获取“共享锁”。  
如果在尝试获取锁时，“不需要阻塞等待”并且“读取锁的共享计数小于MAX\_COUNT”，则直接通过CAS函数更新“读取锁的共享计数”，以及将“当前线程获取读取锁的次数+1”。  
否则，通过fullTryAcquireShared()获取读取锁。

fullTryAcquireShared(Thread current)

final int fullTryAcquireShared(Thread current) {  
 HoldCounter rh = null;  
 for (;;) {  
 // 获取“锁”的状态  
 int c = getState();  
 // 如果“锁”是“互斥锁”，并且获取锁的线程不是current线程；则返回-1。  
 if (exclusiveCount(c) != 0) {  
 if (getExclusiveOwnerThread() != current)  
 return -1;  
 // 如果“需要阻塞等待”。  
 // (01) 当“需要阻塞等待”的线程是第1个获取锁的线程的话，则继续往下执行。  
 // (02) 当“需要阻塞等待”的线程获取锁的次数=0时，则返回-1。  
 } else if (readerShouldBlock()) {  
 // 如果想要获取锁的线程(current)是第1个获取锁(firstReader)的线程  
 if (firstReader == current) {  
 } else {  
 if (rh == null) {  
 rh = cachedHoldCounter;  
 if (rh == null || rh.tid != current.getId()) {  
 rh = readHolds.get();  
 if (rh.count == 0)  
 readHolds.remove();  
 }  
 }  
 // 如果当前线程获取锁的计数=0,则返回-1。  
 if (rh.count == 0)  
 return -1;  
 }  
 }  
 // 如果“不需要阻塞等待”，则获取“读取锁”的共享统计数；  
 // 如果共享统计数超过MAX\_COUNT，则抛出异常。  
 if (sharedCount(c) == MAX\_COUNT)  
 throw new Error("Maximum lock count exceeded");  
 // 将线程获取“读取锁”的次数+1。  
 if (compareAndSetState(c, c + SHARED\_UNIT)) {  
 // 如果是第1次获取“读取锁”，则更新firstReader和firstReaderHoldCount。  
 if (sharedCount(c) == 0) {  
 firstReader = current;  
 firstReaderHoldCount = 1;  
 // 如果想要获取锁的线程(current)是第1个获取锁(firstReader)的线程，  
 // 则将firstReaderHoldCount+1。  
 } else if (firstReader == current) {  
 firstReaderHoldCount++;  
 } else {  
 if (rh == null)  
 rh = cachedHoldCounter;  
 if (rh == null || rh.tid != current.getId())  
 rh = readHolds.get();  
 else if (rh.count == 0)  
 readHolds.set(rh);  
 // 更新线程的获取“读取锁”的共享计数  
 rh.count++;  
 cachedHoldCounter = rh; // cache for release  
 }  
 return 1;  
 }  
 }  
}

**说明**：fullTryAcquireShared()会根据“是否需要阻塞等待”，“读取锁的共享计数是否超过限制”等等进行处理。如果不需要阻塞等待，并且锁的共享计数没有超过限制，则通过CAS尝试获取锁，并返回1。

doAcquireShared(int)

[见AQS篇 doAcquireShared(int)](#_doAcquireShared(int))

###### 3.5.2.3 unlock()

public void unlock() {  
 sync.releaseShared(1);  
}

releaseShared(int)

public final boolean releaseShared(int arg) {  
 if (tryReleaseShared(arg)) {  
 doReleaseShared();//唤醒后续节点  
 return true;  
 }  
 return false;  
}

[见AQS篇 releaseShared(int)](#_3.2.2.4_releaseShared(int))

tryReleaseShared(int)

protected final boolean tryReleaseShared(int unused) {  
 // 获取当前线程，即释放共享锁的线程。  
 Thread current = Thread.*currentThread*();  
 // 如果想要释放锁的线程(current)是第1个获取锁(firstReader)的线程，  
 // 并且“第1个获取锁的线程获取锁的次数”=1，则设置firstReader为null；  
 // 否则，将“第1个获取锁的线程的获取次数”-1。  
 if (firstReader == current) {  
 // assert firstReaderHoldCount > 0;  
 if (firstReaderHoldCount == 1)  
 firstReader = null;  
 else  
 firstReaderHoldCount--;  
 // 获取rh对象，并更新“当前线程获取锁的信息”。  
 } else {  
  
 HoldCounter rh = cachedHoldCounter;  
 if (rh == null || rh.tid != current.getId())  
 rh = readHolds.get();  
 int count = rh.count;  
 if (count <= 1) {  
 readHolds.remove();  
 if (count <= 0)  
 throw unmatchedUnlockException();  
 }  
 --rh.count;  
 }  
 for (;;) {  
 // 获取锁的状态  
 int c = getState();  
 // 将锁的获取次数-1。  
 int nextc = c - SHARED\_UNIT;  
 // 通过CAS更新锁的状态。  
 if (compareAndSetState(c, nextc))  
 return nextc == 0;  
 }  
}

**说明**：tryReleaseShared()的作用是尝试释放共享锁。

doReleaseShared()

[见AQS篇 doReleaseShared(int)](#_doReleaseShared(int))

#### CountDownLatch

##### 概述

CountDownLatch是一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

**CountDownLatch和CyclicBarrier的区别**  
(01) CountDownLatch的作用是允许1或N个线程等待其他线程完成执行；而CyclicBarrier则是允许N个线程相互等待。  
(02) CountDownLatch的计数器无法被重置；CyclicBarrier的计数器可以被重置后使用，因此它被称为是循环的barrier。

###### 函数列表

CountDownLatch(int count)  
构造一个用给定计数初始化的 CountDownLatch。  
  
// 使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断。  
void await()  
// 使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断或超出了指定的等待时间。  
boolean await(long timeout, TimeUnit unit)  
// 递减锁存器的计数，如果计数到达零，则释放所有等待的线程。  
void countDown()  
// 返回当前计数。  
long getCount()

###### 数据结构



##### 源码详解

###### CountDownLatch(int)

public CountDownLatch(int count) {  
 if (count < 0) throw new IllegalArgumentException("count < 0");  
 this.sync = new Sync(count);  
}

**说明**：该函数是创建一个Sync对象，而Sync是继承于AQS类。Sync构造函数如下：

Sync(int count) {  
 setState(count);  
}

protected final void setState(int newState) {  
 state = newState;  
}

**说明**：在AQS中，state是一个private volatile long类型的对象。对于CountDownLatch而言，state表示的”锁计数器“。CountDownLatch中的getCount()最终是调用AQS中的getState()，返回的state对象，即”锁计数器“。

###### await()

使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断或超出了指定的等待时间

public void await() throws InterruptedException {  
 sync.acquireSharedInterruptibly(1);  
}

**说明**：该函数实际上是调用的AQS的acquireSharedInterruptibly(1);

acquireSharedInterruptibly(int)

public final void acquireSharedInterruptibly(int arg)  
 throws InterruptedException {  
 if (Thread.*interrupted*())  
 throw new InterruptedException();  
 if (tryAcquireShared(arg) < 0)  
 doAcquireSharedInterruptibly(arg);  
}

**说明**：acquireSharedInterruptibly()的作用是获取共享锁。  
如果当前线程是中断状态，则抛出异常InterruptedException。否则，调用tryAcquireShared(arg)尝试获取共享锁；尝试成功则返回，否则就调用doAcquireSharedInterruptibly()。doAcquireSharedInterruptibly()会使当前线程一直等待，直到当前线程获取到共享锁(或被中断)才返回。

tryAcquireShared()

tryAcquireShared()在CountDownLatch.java中被重写，它的源码如下：

protected int tryAcquireShared(int acquires) {  
 return (getState() == 0) ? 1 : -1;  
}

**说明**：tryAcquireShared()的作用是尝试获取共享锁。  
如果"锁计数器=0"，即锁是可获取状态，则返回1；否则，锁是不可获取状态，则返回-1。

doAcquireSharedInterruptibly(int)

private void doAcquireSharedInterruptibly(long arg)  
 throws InterruptedException {  
 // 创建"当前线程"的Node节点，且Node中记录的锁是"共享锁"类型；并将该节点添加到CLH队列末尾。  
 final Node node = addWaiter(Node.*SHARED*);  
 boolean failed = true;  
 try {  
 for (;;) {  
 // 获取上一个节点。  
 // 如果上一节点是CLH队列的表头，则"尝试获取共享锁"。  
 final Node p = node.predecessor();  
 if (p == head) {  
 long r = tryAcquireShared(arg);  
 if (r >= 0) {  
 setHeadAndPropagate(node, r);  
 p.next = null; // help GC  
 failed = false;  
 return;  
 }  
 }  
 // (上一节点不是CLH队列的表头) 当前线程一直等待，直到获取到共享锁。  
 // 如果线程在等待过程中被中断过，则再次中断该线程(还原之前的中断状态)。  
 if (shouldParkAfterFailedAcquire(p, node) &&  
 parkAndCheckInterrupt())  
 throw new InterruptedException();  
 }  
 } finally {  
 if (failed)  
 cancelAcquire(node);  
 }  
}

**说明**：  
(01) addWaiter(Node.SHARED)的作用是，创建”当前线程“的Node节点，且Node中记录的锁的类型是”共享锁“(Node.SHARED)；并将该节点添加到CLH队列末尾。  
(02) node.predecessor()的作用是，获取上一个节点。如果上一节点是CLH队列的表头，则”尝试获取共享锁“。  
(03) shouldParkAfterFailedAcquire()的作用和它的名称一样，如果在尝试获取锁失败之后，线程应该等待，则返回true；否则，返回false。  
(04) 当shouldParkAfterFailedAcquire()返回ture时，则调用parkAndCheckInterrupt()，当前线程会进入等待状态，直到获取到共享锁才继续运行。

响应中断的doAcquireShared(int),[见AQS篇doAcquireShared(int)](#_doAcquireShared(int))

###### countDown()

public void countDown() {  
 sync.releaseShared(1);  
}

**说明**：该函数实际上调用releaseShared(1)释放共享锁。

releaseShared(int)

public final boolean releaseShared(int arg) {  
 if (tryReleaseShared(arg)) {  
 doReleaseShared();//唤醒后续节点  
 return true;  
 }  
 return false;  
}

[见AQS篇 3.2.2.4 releaseShared(int)](#_3.2.2.4_releaseShared(int))

##### 小结

CountDownLatch是通过“共享锁”实现的。在创建CountDownLatch中时，会传递一个int类型参数count，该参数是“锁计数器”的初始状态，表示该“共享锁”最多能被count给线程同时获取。当某线程调用该CountDownLatch对象的await()方法时，该线程会等待“共享锁”可用时，才能获取“共享锁”进而继续运行。而“共享锁”可用的条件，就是“锁计数器”的值为0！而“锁计数器”的初始值为count，每当一个线程调用该CountDownLatch对象的countDown()方法时，才将“锁计数器”-1；通过这种方式，必须有count个线程调用countDown()之后，“锁计数器”才为0，而前面提到的等待线程才能继续运行！

#### CyclicBarrier

##### 概述

CyclicBarrier是一个同步辅助类，允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。因为该 barrier 在释放等待线程后可以重用，所以称它为循环 的 barrier。

注意比较CountDownLatch和CyclicBarrier：  
(01) CountDownLatch的作用是允许1或N个线程等待其他线程完成执行；而CyclicBarrier则是允许N个线程相互等待。  
(02) CountDownLatch的计数器无法被重置；CyclicBarrier的计数器可以被重置后使用，因此它被称为是循环的barrier。

###### 函数列表

// 每次对barrier的使用可以表现为一个 generation 实例。当条件 trip 改变或者重置 generation 也会  
// 随之改变。可以有多个 generation 和使用barrier的线程关联，但是只有一个可以获得锁。  
private static class Generation {  
 boolean broken = false;  
}  
*/\*\* 守护barrier入口的锁 \*/*private final ReentrantLock lock = new ReentrantLock();  
*/\*\* 等待条件，直到所有线程到达barrier \*/*private final Condition trip = lock.newCondition();  
*/\*\* 要屏障的线程数 \*/*private final int parties;  
/\* 当线程都到达barrier，运行的 Runnable \*/  
private final Runnable barrierCommand;  
*/\*\* The current generation \*/*private Generation generation = new Generation();  
  
//还要等待多少个线程到达。线程到达屏障点就减去 1。  
//每次新建 generation 的时候或者屏障 broken，count重新设置为 parties 参数值  
private int count;  
  
//创建一个新的 CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，但它不会在启动 barrier 时执行预定义的操作。  
CyclicBarrier(int parties)  
//创建一个新的 CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，并在启动 barrier 时执行给定的屏障操作，该操作由最后一个进入 barrier 的线程执行。  
CyclicBarrier(int parties, Runnable barrierAction);  
//在 一个 barrier 完成后, 重新初始化值  
private void nextGeneration();  
//用于等待的线程当被中断, 或等待超时执行  
private void breakBarrier();  
private int dowait(boolean timed, long nanos);  
//在所有参与者都已经在此 barrier 上调用 await 方法之前，将一直等待。  
public int await();  
//在所有参与者都已经在此屏障上调用 await 方法之前将一直等待,或者超出了指定的等待时间。  
public int await(long timeout, TimeUnit unit);  
//返回要求启动此 barrier 的参与者数目。  
public int getParties();  
//查询此屏障是否处于损坏状态。  
public boolean isBroken();  
//将屏障重置为其初始状态。  
public void reset();  
//返回当前在屏障处等待的参与者数目。  
public int getNumberWaiting();

##### 源码详解

###### await()

public int await() throws InterruptedException, BrokenBarrierException {  
 try {  
 return dowait(false, 0L);  
 } catch (TimeoutException toe) {  
 throw new Error(toe); // cannot happen  
 }  
}

public int await(long timeout, TimeUnit unit)  
 throws InterruptedException,  
 BrokenBarrierException,  
 TimeoutException {  
 return dowait(true, unit.toNanos(timeout));  
}

doawait(boolean timed, long nanos)

private int dowait(boolean timed, long nanos)  
 throws InterruptedException, BrokenBarrierException,  
 TimeoutException {  
 final ReentrantLock lock = this.lock;  
 // 获取“独占锁(lock)”  
 lock.lock();  
 try {  
 // 保存“当前的generation”  
 final Generation g = generation;  
  
 // 若“当前generation已损坏”，则抛出异常。  
 if (g.broken)  
 throw new BrokenBarrierException();  
  
 // 如果当前线程被中断，则通过breakBarrier()终止CyclicBarrier，唤醒CyclicBarrier中所有等待线程。  
 if (Thread.*interrupted*()) {  
 breakBarrier();  
 throw new InterruptedException();  
 }  
  
 // 将“count计数器”-1  
 int index = --count;  
 // 如果index=0，则意味着“有parties个线程到达barrier”。  
 if (index == 0) { // tripped  
 boolean ranAction = false;  
 try {  
 // 如果barrierCommand不为null，则执行该动作。  
 final Runnable command = barrierCommand;  
 if (command != null)  
 command.run();  
 ranAction = true;  
 // 唤醒所有等待线程，并更新generation。  
 nextGeneration();  
 return 0;  
 } finally {  
 if (!ranAction)  
 breakBarrier();  
 }  
 }  
  
 // 当前线程一直阻塞，直到“有parties个线程到达barrier” 或 “当前线程被中断” 或 “超时”这3者之一发生，  
 // 当前线程才继续执行。  
 for (;;) {  
 try {  
 // 如果不是“超时等待”，则调用awati()进行等待；否则，调用awaitNanos()进行等待。  
 if (!timed)  
 trip.await();  
 else if (nanos > 0L)  
 nanos = trip.awaitNanos(nanos);  
 } catch (InterruptedException ie) {  
 // 如果等待过程中，线程被中断，则执行下面的函数。  
 if (g == generation && ! g.broken) {  
 breakBarrier();  
 throw ie;  
 } else {  
 Thread.*currentThread*().interrupt();  
 }  
 }  
  
 // 如果“当前generation已经损坏”，则抛出异常。  
 if (g.broken)  
 throw new BrokenBarrierException();  
  
 // 如果“generation已经换代”，则返回index。  
 if (g != generation)  
 return index;  
  
 // 如果是“超时等待”，并且时间已到，则通过breakBarrier()终止CyclicBarrier，唤醒CyclicBarrier中所有等待线程。  
 if (timed && nanos <= 0L) {  
 breakBarrier();  
 throw new TimeoutException();  
 }  
 }  
 } finally {  
 // 释放“独占锁(lock)”  
 lock.unlock();  
 }  
}

**说明**：dowait()的作用就是让当前线程阻塞，直到“有parties个线程到达barrier” 或 “当前线程被中断” 或 “超时”这3者之一发生，当前线程才继续执行。

(01) generation是CyclicBarrier的一个成员遍历，它的定义如下：

private static class Generation {  
 boolean broken = false;  
}

在CyclicBarrier中，同一批的线程属于同一代，即同一个Generation；CyclicBarrier中通过generation对象，记录属于哪一代。  
当有parties个线程到达barrier，generation就会被更新换代。

(02) 如果当前线程被中断，即Thread.interrupted()为true；则通过breakBarrier()终止CyclicBarrier。breakBarrier()的源码如下：

private void breakBarrier() {  
 generation.broken = true;  
 count = parties;  
 trip.signalAll();  
}

breakBarrier()会设置当前中断标记broken为true，意味着“将该Generation中断”；同时，设置count=parties，即重新初始化count；最后，通过signalAll()唤醒CyclicBarrier上所有的等待线程。

(03) 将“count计数器”-1，即--count；然后判断是不是“有parties个线程到达barrier”，即index是不是为0。当index=0时，如果barrierCommand不为null，则执行该barrierCommand，barrierCommand就是我们创建CyclicBarrier时，传入的Runnable对象。然后，调用nextGeneration()进行换代工作，nextGeneration()的源码如下：

private void nextGeneration() {  
 // signal completion of last generation  
 trip.signalAll();  
 // set up next generation  
 count = parties;  
 generation = new Generation();  
}

首先，它会调用signalAll()唤醒CyclicBarrier上所有的等待线程；接着，重新初始化count；最后，更新generation的值。

1. 在for(;;)循环中。timed是用来表示当前是不是“超时等待”线程。如果不是，则通过trip.await()进行等待；否则，调用awaitNanos()进行超时等待

#### Semaphore

##### 概述

Semaphore是一个计数信号量，它的本质是一个"**共享锁**"。

信号量维护了一个信号量许可集。线程可以通过调用acquire()来获取信号量的许可；当信号量中有可用的许可时，线程能获取该许可；否则线程必须等待，直到有可用的许可为止。 线程可以通过release()来释放它所持有的信号量许可。

###### 函数列表

// 创建具有给定的许可数和非公平的公平设置的 Semaphore。  
Semaphore(int permits)  
// 创建具有给定的许可数和给定的公平设置的 Semaphore。  
Semaphore(int permits, boolean fair)  
  
// 从此信号量获取一个许可，在提供一个许可前一直将线程阻塞，否则线程被中断。  
void acquire()  
// 从此信号量获取给定数目的许可，在提供这些许可前一直将线程阻塞，或者线程已被中断。  
void acquire(int permits)  
// 从此信号量中获取许可，在有可用的许可前将其阻塞。  
void acquireUninterruptibly()  
// 从此信号量获取给定数目的许可，在提供这些许可前一直将线程阻塞。  
void acquireUninterruptibly(int permits)  
// 返回此信号量中当前可用的许可数。  
int availablePermits()  
// 获取并返回立即可用的所有许可。  
int drainPermits()  
// 返回一个 collection，包含可能等待获取的线程。  
protected Collection<Thread> getQueuedThreads()  
// 返回正在等待获取的线程的估计数目。  
int getQueueLength()  
// 查询是否有线程正在等待获取。  
boolean hasQueuedThreads()  
// 如果此信号量的公平设置为 true，则返回 true。  
boolean isFair()  
// 根据指定的缩减量减小可用许可的数目。  
protected void reducePermits(int reduction)  
// 释放一个许可，将其返回给信号量。  
void release()  
// 释放给定数目的许可，将其返回到信号量。  
void release(int permits)  
// 返回标识此信号量的字符串，以及信号量的状态。  
String toString()  
// 仅在调用时此信号量存在一个可用许可，才从信号量获取许可。  
boolean tryAcquire()  
// 仅在调用时此信号量中有给定数目的许可时，才从此信号量中获取这些许可。  
boolean tryAcquire(int permits)  
// 如果在给定的等待时间内此信号量有可用的所有许可，并且当前线程未被中断，则从此信号量获取给定数目的许可。  
boolean tryAcquire(int permits, long timeout, TimeUnit unit)  
// 如果在给定的等待时间内，此信号量有可用的许可并且当前线程未被中断，则从此信号量获取一个许可。  
boolean tryAcquire(long timeout, TimeUnit unit)

###### 数据结构



(01) 和"ReentrantLock"一样，Semaphore也包含了sync对象，sync是Sync类型；而且，Sync是一个继承于AQS的抽象类。(02) Sync包括两个子类："公平信号量"FairSync 和 "非公平信号量"NonfairSync。sync是"FairSync的实例"，或者"NonfairSync的实例"；默认情况下，sync是NonfairSync(即，默认是非公平信号量)。

Sync源码:

abstract static class Sync extends AbstractQueuedSynchronizer {  
 private static final long *serialVersionUID* = 1192457210091910933L;  
  
 Sync(int permits) {  
 setState(permits);  
 }  
 //获取许可  
 final int getPermits() {  
 return getState();  
 }  
 //非公平获取  
 final int nonfairTryAcquireShared(int acquires) {  
 for (;;) {  
 int available = getState();  
 int remaining = available - acquires;  
 if (remaining < 0 ||  
 compareAndSetState(available, remaining))  
 return remaining;  
 }  
 }  
 //释放  
 protected final boolean tryReleaseShared(int releases) {  
 for (;;) {  
 int current = getState();  
 int next = current + releases;  
 if (next < current) // overflow  
 throw new Error("Maximum permit count exceeded");  
 if (compareAndSetState(current, next))  
 return true;  
 }  
 }  
 //减少指定许可数  
 final void reducePermits(int reductions) {  
 for (;;) {  
 int current = getState();  
 int next = current - reductions;  
 if (next > current) // underflow  
 throw new Error("Permit count underflow");  
 if (compareAndSetState(current, next))  
 return;  
 }  
 }  
 //获取并返回立即可用的所有许可  
 final int drainPermits() {  
 for (;;) {  
 int current = getState();  
 if (current == 0 || compareAndSetState(current, 0))  
 return current;  
 }  
 }  
}

*/\*\*非公平Sync\*/*static final class NonfairSync extends Sync {  
 private static final long *serialVersionUID* = -2694183684443567898L;  
  
 NonfairSync(int permits) {  
 super(permits);  
 }  
  
 protected int tryAcquireShared(int acquires) {  
 return nonfairTryAcquireShared(acquires);  
 }  
}  
*/\*\*公平Sync\*/*static final class FairSync extends Sync {  
 private static final long *serialVersionUID* = 2014338818796000944L;  
  
 FairSync(int permits) {  
 super(permits);  
 }  
  
 protected int tryAcquireShared(int acquires) {  
 for (;;) {  
 if (hasQueuedPredecessors())  
 return -1;  
 int available = getState();  
 int remaining = available - acquires;  
 if (remaining < 0 ||  
 compareAndSetState(available, remaining))  
 return remaining;  
 }  
 }  
}

**"公平信号量"和"非公平信号量"的区别**

"公平信号量"和"非公平信号量"的释放信号量的机制是一样的！不同的是它们获取信号量的机制：线程在尝试获取信号量许可时，对于公平信号量而言，如果当前线程不在CLH队列的头部，则排队等候；而对于非公平信号量而言，无论当前线程是不是在CLH队列的头部，它都会直接获取信号量。该差异具体的体现在，它们的tryAcquireShared()函数的实现不同。

##### 源码分析

###### acquire()

//获取信号量  
public void acquire() throws InterruptedException {  
 sync.acquireSharedInterruptibly(1);  
}  
//获取指定permits数的信号量  
public void acquire(int permits) throws InterruptedException {  
 if (permits < 0) throw new IllegalArgumentException();  
 sync.acquireSharedInterruptibly(permits);  
}  
//获取指定permits数的信号量,不响应中断  
public void acquireUninterruptibly(int permits) {  
 if (permits < 0) throw new IllegalArgumentException();  
 sync.acquireShared(permits);  
}

**说明:** 可以看出,Semaphore的获取锁是响应中断的,实际调用的是AQS的acquireSharedInterruptibly(int),

acquireSharedInterruptibly(int)

public final void acquireSharedInterruptibly(int arg)  
 throws InterruptedException {  
 if (Thread.*interrupted*())  
 throw new InterruptedException();  
 if (tryAcquireShared(arg) < 0)  
 doAcquireSharedInterruptibly(arg);  
}

[见CountDownLatch篇acquireSharedInterruptibly(int)](#_acquireSharedInterruptibly(int))

tryAcquireShared(int)

tryAcquireShared(int)在Semaphore中被重写,具体实现如下:

//非公平信号量获取  
protected int tryAcquireShared(int acquires) {  
 return nonfairTryAcquireShared(acquires);  
}  
//公平信号量获取  
protected int tryAcquireShared(int acquires) {  
 for (;;) {  
 // 判断当前线程是否还有前任线程  
 if (hasQueuedPredecessors())  
 return -1;  
 //可获得的信号数  
 int available = getState();  
 //获取信号数之后剩余的信号数  
 int remaining = available - acquires;  
 if (remaining < 0 ||  
 compareAndSetState(available, remaining))  
 return remaining;  
 }  
}

**说明:**

(1)非公平获取中,调用Sync的nonfairTryAcquireShared (int),源码如下:

final int nonfairTryAcquireShared(int acquires) {  
 for (;;) {  
 int available = getState();  
 int remaining = available - acquires;  
 if (remaining < 0 ||  
 compareAndSetState(available, remaining))  
 return remaining;  
 }  
}

(2)公平获取信号量中,首先根据hasQueuePredecessors()判断是否还有前任线程,有的话则当前线程需要通过doAcquireSharedInterruptibly(int)进入等待队列自旋等待.

doAcquireSharedInterruptibly(int)

**说明:** [见CountDownLatch篇doAcquireSharedInterruptibly(int)](#_doAcquireSharedInterruptibly(int))

###### release()

"公平信号量"和"非公平信号量"的释放信号量的机制是相同的

//释放信号量  
public void release() {  
 sync.releaseShared(1);  
}  
//释放指定permits数的信号量  
public void release(int permits) {  
 if (permits < 0) throw new IllegalArgumentException();  
 sync.releaseShared(permits);  
}

**说明:** 实际调用的是AQS的releaseShared(int).

releaseShared(int)

public final boolean releaseShared(int arg) {  
 if (tryReleaseShared(arg)) {  
 doReleaseShared();  
 return true;  
 }  
 return false;  
}

**说明:** tryReleaseShared(int)在Semaphore中的Sync重写.

tryReleaseShared(int)

protected final boolean tryReleaseShared(int releases) {  
 for (;;) {  
 //可获得的信号数  
 int current = getState();  
 //释放releases个信号后,剩余可获得的信号数  
 int next = current + releases;  
 if (next < current) // overflow  
 throw new Error("Maximum permit count exceeded");  
 //设置可获得的信号数为next  
 if (compareAndSetState(current, next))  
 return true;  
 }  
}

说明: 如果在tryReleaseShared()中尝试释放共享锁失败,则会调用doReleaseShared()进行释放.

doReleaseShared()

[见AQS篇doReleaseShared(int)](#_doReleaseShared(int))

#### StampedLock

##### 概述

StampedLock是JDK1.8新增的锁,与之前的ReentrantLock，ReentrantReadWriteLock使用队列同步列AQS实现有所不同，StampedLock的state改为了一个long型的变量，同时状态的设计也有所不同。同时由于没有使用AQSStampedLock直接在内部实现了同步等待队列，并且节点属性中有一个叫做cowait的分支用于标识另一个等待获取读状态的链。

StampedLock采用了一个long型作为state，把这个64位的state的前7位作为读状态，第8位标识写状态，这也是为什么不支持重入的原因吧。

##### 相关属性定义

//获取CPU的可用线程数量，用于确定自旋的时候循环次数  
private static final int *NCPU* = Runtime.*getRuntime*().availableProcessors();  
//根据NCPU确定自旋的次数限制(并不是一定这么多次，因为实际代码中是随机的)  
private static final int *SPINS* = (*NCPU* > 1) ? 1 << 6 : 0;  
//头节点上的自旋次数  
private static final int *HEAD\_SPINS* = (*NCPU* > 1) ? 1 << 10 : 0;  
//头节点上的最大自旋次数  
private static final int *MAX\_HEAD\_SPINS* = (*NCPU* > 1) ? 1 << 16 : 0;  
private static final int *LG\_READERS* = 7;  
//一个读状态单位  
private static final long *RUNIT* = 1L;  
//写状态标识  
private static final long *WBIT* = 1L << *LG\_READERS*;  
//读状态标识(前7位)  
private static final long *RBITS* = *WBIT* - 1L;  
//最大的读状态  
private static final long *RFULL* = *RBITS* - 1L;  
//用于获取读写状态  
private static final long *ABITS* = *RBITS* | *WBIT*;  
private static final long *SBITS* = ~*RBITS*; // note overlap with ABITS  
//初始化状态  
private static final long *ORIGIN* = *WBIT* << 1;  
//中断标识  
private static final long *INTERRUPTED* = 1L;  
// 等待/取消  
private static final int *WAITING* = -1;  
private static final int *CANCELLED* = 1;  
//读/写状态  
private static final int *RMODE* = 0;  
private static final int *WMODE* = 1;  
//因为读状态只有7位很小，所以当超过了128之后将使用一个int变量来记录  
private transient int readerOverflow;

状态标识:

写线程获取到了写状态只需判断：state < WBIT

读状态是否超出：(state & ABITS) < RFULL

获取读状态： state + RUNIT(或者readerOverflow + 1)

获取写状态: state + WBIT

释放读状态： state - RUNIT(或者readerOverflow - 1)

释放写状态： (s += WBIT) == 0L ? ORIGIN : s

是否为写锁： (state & WBIT) != 0L

是否为读锁： (state & RBITS) != 0L

##### 小结

一 StampedLock是不可重入的，

二StampedLock对于带有中断的线程的处理可能导致CPU暴涨。

*/\*\*  
 \* 先开启一个线程获取写锁并保持10秒，再开启10个带着中断状态的线程去获取读锁（readLock方法），结果是3个核心被占据了近10秒。  
 \* 原因在于没有使用保存/复原中断状态的机制  
 \* Created by zhaoshengqi on 2017/8/1.  
 \*/*public class TestBugStampedLock {  
  
 public static void main(String[] args) throws InterruptedException{  
 final StampedLock lock = new StampedLock();  
 new Thread(){  
 public void run(){  
 long readLong = lock.writeLock();  
 LockSupport.*parkNanos*(10100000000L);  
 lock.unlockWrite(readLong);  
 }  
 }.start();  
 Thread.*sleep*(100);  
 for( int i = 0; i < 10; ++i)  
 new Thread(new OccupiedCPUReadThread(lock)).start();  
 }  
 private static class OccupiedCPUReadThread implements Runnable{  
 private StampedLock lock;  
 public OccupiedCPUReadThread(StampedLock lock){  
 this.lock = lock;  
 }  
 public void run(){  
 Thread.*currentThread*().interrupt();  
 long lockr = lock.readLock();  
 System.*out*.println(Thread.*currentThread*().getName() + " get read lock");  
 lock.unlockRead(lockr);  
 }  
 }  
}

三StampedLock的所有的try开头的获取都将直接尝试获取锁。

四StampedLock适用于读多写少的场景。

五 源码暂时看不懂

### JUC集合篇

#### 框架

##### List和Set



(01) CopyOnWriteArrayList相当于线程安全的ArrayList，它实现了List接口。CopyOnWriteArrayList是支持高并发的。  
(02) CopyOnWriteArraySet相当于线程安全的HashSet，它继承于AbstractSet类。CopyOnWriteArraySet内部包含一个CopyOnWriteArrayList对象，它是通过CopyOnWriteArrayList实现的。

(03) ConcurrentSkipListSet在Map中说明。

##### Map



(01) ConcurrentHashMap是线程安全的哈希表(相当于线程安全的HashMap)；它继承于AbstractMap类，并且实现ConcurrentMap接口。ConcurrentHashMap是通过“锁分段”来实现的，它支持并发。

(02) ConcurrentSkipListMap是线程安全的有序的哈希表(相当于线程安全的TreeMap); 它继承于AbstractMap类，并且实现ConcurrentNavigableMap接口。ConcurrentSkipListMap是通过“跳表”来实现的，它支持并发。

(03) ConcurrentSkipListSet是线程安全的有序的集合(相当于线程安全的TreeSet)；它继承于AbstractSet，并实现了NavigableSet接口。ConcurrentSkipListSet是通过ConcurrentSkipListMap实现的，它也支持并发。

##### Queue



(01) ArrayBlockingQueue是数组实现的线程安全的有界的阻塞队列。  
(02) LinkedBlockingQueue是单向链表实现的(指定大小)阻塞队列,使用锁机制实现，该队列按 FIFO（先进先出）排序元素。  
(03) LinkedBlockingDeque是双向链表实现的(指定大小)双向并发阻塞队列，使用锁机制实现,该阻塞队列同时支持FIFO和FILO两种操作方式。  
(04) ConcurrentLinkedQueue是单向链表实现的无界并发队列，通过cas实现并发,该队列按 FIFO（先进先出）排序元素。  
(05) ConcurrentLinkedDeque是双向链表实现的无界并发队列，通过cas实现并发,该队列同时支持FIFO和FILO两种操作方式。  
(06) DelayQueue延时无界阻塞队列。  
(07) SynchronousQueue同步队列  
(08) LinkedTransferQueue 1.7新增,是ConcurrentLinkedQueue、SynchronousQueue（公平模式）和LinkedBlockingQueue的超集, 它不仅仅综合了这几个类的功能，同时也提供了更高效的实现。

#### CopyOnwriteArrayList

##### 概述

它相当于线程安全的[ArrayList](http://www.cnblogs.com/skywang12345/p/3308556.html)。和ArrayList一样，它是个可变数组；但是和ArrayList不同的时，它具有以下特性：  
1. 它最适合于具有以下特征的应用程序：List 大小通常保持很小，只读操作远多于可变操作，需要在遍历期间防止线程间的冲突。  
2. 它是线程安全的,使用ReentrantLock和volatile保证线程安全。  
3. 因为通常需要复制整个基础数组，所以可变操作（add()、set() 和 remove() 等等）的开销很大。  
4. 迭代器支持hasNext(), next()等不可变操作，但不支持可变 remove()等操作。  
5. 使用迭代器进行遍历的速度很快，并且不会与其他线程发生冲突。在构造迭代器时，迭代器依赖于不变的数组快照。

###### 数据结构



**说明**：  
1. CopyOnWriteArrayList实现了List接口，因此它是一个队列。  
2. CopyOnWriteArrayList包含了成员lock。每一个CopyOnWriteArrayList都和一个互斥锁lock绑定，通过lock，实现了对CopyOnWriteArrayList的互斥访问。  
3. CopyOnWriteArrayList包含了用volatile修饰的成员array数组，这说明CopyOnWriteArrayList本质上通过数组实现的。

下面从“动态数组”和“线程安全”两个方面进一步对CopyOnWriteArrayList的原理进行说明。  
1.**CopyOnWriteArrayList的“动态数组”机制** -- 它内部有个“volatile数组”(array)来保持数据。在“添加/修改/删除”数据时，都会新建一个数组，并将更新后的数据拷贝到新建的数组中，最后再将该数组赋值给“volatile数组”。这就是它叫做CopyOnWriteArrayList的原因！CopyOnWriteArrayList就是通过这种方式实现的动态数组；不过正由于它在“添加/修改/删除”数据时，都会新建数组，所以涉及到修改数据的操作，CopyOnWriteArrayList效率很低；但是单单只是进行遍历查找的话，效率比较高。  
2.**CopyOnWriteArrayList的“线程安全”机制** -- 是通过volatile和互斥锁来实现的。  
(01) CopyOnWriteArrayList是通过“volatile数组”来保存数据的。一个线程读取volatile数组时，总能看到其它线程对该volatile变量最后的写入；就这样，通过volatile提供了“读取到的数据总是最新的”这个机制的保证。  
(02) CopyOnWriteArrayList通过互斥锁来保护数据。在“添加/修改/删除”数据时，会先“获取互斥锁”，再修改完毕之后，先将数据更新到“volatile数组”中，然后再“释放互斥锁”；这样，就达到了保护数据的目的。

###### 函数列表

// 创建一个空列表。  
CopyOnWriteArrayList();  
// 创建一个按 collection 的迭代器返回元素的顺序包含指定 collection 元素的列表。  
CopyOnWriteArrayList(Collection<? extends E> c)  
//创建一个保存给定数组的副本的列表。  
CopyOnWriteArrayList(E[] toCopyIn)  
  
// 将指定元素添加到此列表的尾部。  
boolean add(E e)  
// 在此列表的指定位置上插入指定元素。  
void add(int index, E element)  
// 按照指定 collection 的迭代器返回元素的顺序，将指定 collection 中的所有元素添加此列表的尾部。  
boolean addAll(Collection<? extends E> c)  
// 从指定位置开始，将指定 collection 的所有元素插入此列表。  
boolean addAll(int index, Collection<? extends E> c)  
// 按照指定 collection 的迭代器返回元素的顺序，将指定 collection 中尚未包含在此列表中的所有元素添加列表的尾部。  
int addAllAbsent(Collection<? extends E> c)  
// 添加元素（如果不存在）。  
boolean addIfAbsent(E e)  
// 从此列表移除所有元素。  
void clear()  
// 返回此列表的浅表副本。  
Object clone()  
// 如果此列表包含指定的元素，则返回 true。  
boolean contains(Object o)  
// 如果此列表包含指定 collection 的所有元素，则返回 true。  
boolean containsAll(Collection<?> c)  
// 比较指定对象与此列表的相等性。  
boolean equals(Object o)  
// 返回列表中指定位置的元素。  
E get(int index)  
// 返回此列表的哈希码值。  
int hashCode()  
// 返回第一次出现的指定元素在此列表中的索引，从 index 开始向前搜索，如果没有找到该元素，则返回 -1。  
int indexOf(E e, int index)  
// 返回此列表中第一次出现的指定元素的索引；如果此列表不包含该元素，则返回 -1。  
int indexOf(Object o)  
// 如果此列表不包含任何元素，则返回 true。  
boolean isEmpty()  
// 返回以恰当顺序在此列表元素上进行迭代的迭代器。  
Iterator<E> iterator()  
// 返回最后一次出现的指定元素在此列表中的索引，从 index 开始向后搜索，如果没有找到该元素，则返回 -1。  
int lastIndexOf(E e, int index)  
// 返回此列表中最后出现的指定元素的索引；如果列表不包含此元素，则返回 -1。  
int lastIndexOf(Object o)  
// 返回此列表元素的列表迭代器（按适当顺序）。  
ListIterator<E> listIterator()  
// 返回列表中元素的列表迭代器（按适当顺序），从列表的指定位置开始。  
ListIterator<E> listIterator(int index)  
// 移除此列表指定位置上的元素。  
E remove(int index)  
// 从此列表移除第一次出现的指定元素（如果存在）。  
boolean remove(Object o)  
// 从此列表移除所有包含在指定 collection 中的元素。  
boolean removeAll(Collection<?> c)  
// 只保留此列表中包含在指定 collection 中的元素。  
boolean retainAll(Collection<?> c)  
// 用指定的元素替代此列表指定位置上的元素。  
E set(int index, E element)  
// 返回此列表中的元素数。  
int size()  
// 返回此列表中 fromIndex（包括）和 toIndex（不包括）之间部分的视图。  
List<E> subList(int fromIndex, int toIndex)  
// 返回一个按恰当顺序（从第一个元素到最后一个元素）包含此列表中所有元素的数组。  
Object[] toArray()  
// 返回以恰当顺序（从第一个元素到最后一个元素）包含列表所有元素的数组；返回数组的运行时类型是指定数组的运行时类型。  
<T> T[] toArray(T[] a)  
// 返回此列表的字符串表示形式。  
String toString();

##### 源码解析

###### 创建

private transient volatile Object[] array;  
final Object[] getArray() {  
 return array;  
}  
final void setArray(Object[] a) {  
 array = a;  
}  
  
public CopyOnWriteArrayList() {  
 setArray(new Object[0]);  
}  
  
public CopyOnWriteArrayList(Collection<? extends E> c) {  
 Object[] elements;  
 if (c.getClass() == CopyOnWriteArrayList.class)  
 elements = ((CopyOnWriteArrayList<?>)c).getArray();  
 else {  
 elements = c.toArray();  
 // c.toArray might (incorrectly) not return Object[] (see 6260652)  
 if (elements.getClass() != Object[].class)  
 elements = Arrays.*copyOf*(elements, elements.length, Object[].class);  
 }  
 setArray(elements);  
}  
  
public CopyOnWriteArrayList(E[] toCopyIn) {  
 setArray(Arrays.*copyOf*(toCopyIn, toCopyIn.length, Object[].class));  
}

**说明**：setArray()的作用是给array赋值；其中，array是volatile transient Object[]类型，即array是“volatile数组”。  
关于volatile关键字，我们知道“volatile能让变量变得可见”，即对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。正在由于这种特性，每次更新了“volatile数组”之后，其它线程都能看到对它所做的更新。  
关于transient关键字，它是在序列化中才起作用，transient变量不会被自动序列化。

###### add(index, E)

public boolean add(E e) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] elements = getArray();  
 int len = elements.length;  
 Object[] newElements = Arrays.*copyOf*(elements, len + 1);  
 newElements[len] = e;  
 setArray(newElements);  
 return true;  
 } finally {  
 lock.unlock();  
 }  
}  
public void add(int index, E element) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] elements = getArray();  
 int len = elements.length;  
 if (index > len || index < 0)  
 throw new IndexOutOfBoundsException("Index: "+index+  
 ", Size: "+len);  
 Object[] newElements;  
 //计算偏移量  
 int numMoved = len - index;  
 if (numMoved == 0)  
 //作为add(E)处理  
 newElements = Arrays.*copyOf*(elements, len + 1);  
 else {  
 newElements = new Object[len + 1];  
 //调用native方法根据index拷贝原数组的前半段  
 System.*arraycopy*(elements, 0, newElements, 0, index);  
 //拷贝后半段  
 System.*arraycopy*(elements, index, newElements, index + 1,  
 numMoved);  
 }  
 newElements[index] = element;  
 setArray(newElements);  
 } finally {  
 lock.unlock();  
 }  
}

//System中arrayCopy的实现  
public static native void arraycopy(Object src, int srcPos,  
 Object dest, int destPos,  
 int length);

**说明**：add(E e)的作用就是将数据e添加到”volatile数组“中。它的实现方式是，新建一个数组，接着将原始的”volatile数组“的数据拷贝到新数组中，然后将新增数据也添加到新数组中；最后，将新数组赋值给”volatile数组“。add(int index, E e)中,调用了native方法,根据index把原E分为两段拷贝

###### addIfAbsent(E)

public boolean addIfAbsent(E e) {  
 Object[] snapshot = getArray();  
 return indexOf(e, snapshot, 0, snapshot.length) >= 0 ? false :  
 addIfAbsent(e, snapshot);  
}  
  
private boolean addIfAbsent(E e, Object[] snapshot) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] current = getArray();  
 int len = current.length;  
 if (snapshot != current) {  
 // Optimize for lost race to another addXXX operation  
 //操作中有别的线程对array做了修改,取较小的那个length  
 int common = Math.*min*(snapshot.length, len);  
 for (int i = 0; i < common; i++)  
 if (current[i] != snapshot[i] && eq(e, current[i]))  
 return false;  
 if (indexOf(e, current, common, len) >= 0)  
 return false;  
 }  
 Object[] newElements = Arrays.*copyOf*(current, len + 1);  
 newElements[len] = e;  
 setArray(newElements);  
 return true;  
 } finally {  
 lock.unlock();  
 }  
}

**说明**：addIfAbsent(E e)与add不同的是,如果当前元素存在就不添加,并返回false“  
注意:addIfAbsent(E e)首先判断indexOf,这个操作是没有lock的,所以addIfAbsent(E, O[])会判断参数数组是否跟加了lock之后获取到的数组是否相同

###### set(index, e)

public E set(int index, E element) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] elements = getArray();  
 E oldValue = get(elements, index);  
  
 if (oldValue != element) {  
 int len = elements.length;  
 Object[] newElements = Arrays.*copyOf*(elements, len);  
 newElements[index] = element;  
 setArray(newElements);  
 } else {  
 // Not quite a no-op; ensures volatile write semantics  
 setArray(elements);  
 }  
 return oldValue;  
 } finally {  
 lock.unlock();  
 }  
}

**说明**：和add()相比,add返回boolean,set返回修改之前的值oldValue

###### remove()

public E remove(int index) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] elements = getArray();  
 int len = elements.length;  
 E oldValue = get(elements, index);  
 int numMoved = len - index - 1;  
 if (numMoved == 0)  
 setArray(Arrays.*copyOf*(elements, len - 1));  
 else {  
 Object[] newElements = new Object[len - 1];  
 System.*arraycopy*(elements, 0, newElements, 0, index);  
 System.*arraycopy*(elements, index + 1, newElements, index,  
 numMoved);  
 setArray(newElements);  
 }  
 return oldValue;  
 } finally {  
 lock.unlock();  
 }  
}  
//移除某个元素  
public boolean remove(Object o) {  
 Object[] snapshot = getArray();  
 int index = indexOf(o, snapshot, 0, snapshot.length);  
 return (index < 0) ? false : remove(o, snapshot, index);  
}  
  
private boolean remove(Object o, Object[] snapshot, int index) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] current = getArray();  
 int len = current.length;  
 //判断是否有别的线程对array做了修改  
 if (snapshot != current) findIndex: {  
 int prefix = Math.*min*(index, len);  
 for (int i = 0; i < prefix; i++) {  
 //找到其他线程修改后的不同的元素 比较是否为此线程需要操作的元素  
 if (current[i] != snapshot[i] && eq(o, current[i])) {  
 index = i;  
 break findIndex;  
 }  
 }  
 if (index >= len)  
 return false;  
 if (current[index] == o)  
 break findIndex;  
 index = indexOf(o, current, index, len);  
 if (index < 0)  
 return false;  
 }  
 Object[] newElements = new Object[len - 1];  
 System.*arraycopy*(current, 0, newElements, 0, index);  
 System.*arraycopy*(current, index + 1,  
 newElements, index,  
 len - index - 1);  
 setArray(newElements);  
 return true;  
 } finally {  
 lock.unlock();  
 }  
}  
  
void removeRange(int fromIndex, int toIndex) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] elements = getArray();  
 int len = elements.length;  
  
 if (fromIndex < 0 || toIndex > len || toIndex < fromIndex)  
 throw new IndexOutOfBoundsException();  
 int newlen = len - (toIndex - fromIndex);  
 int numMoved = len - toIndex;  
 if (numMoved == 0)  
 setArray(Arrays.*copyOf*(elements, newlen));  
 else {  
 Object[] newElements = new Object[newlen];  
 System.*arraycopy*(elements, 0, newElements, 0, fromIndex);  
 System.*arraycopy*(elements, toIndex, newElements,  
 fromIndex, numMoved);  
 setArray(newElements);  
 }  
 } finally {  
 lock.unlock();  
 }  
}

**说明**：(1) remove(int index)的作用就是将”volatile数组“中第index个元素删除。它的实现方式是，如果被删除的是最后一个元素，则直接通过Arrays.copyOf()进行处理，而不需要新建数组。否则，新建数组，然后将”volatile数组中被删除元素之外的其它元素“拷贝到新数组中；最后，将新数组赋值给”volatile数组“。

(2)remove(O)是把数组中的o元素移除(如果存在),成功返回true.调用了remove(O,snapshot,index),跟addIfAbsent一样,也会判断是否在操作中有其他线程对数组做了修改.

###### iterator()

public Iterator<E> iterator() {  
 return new COWIterator<E>(getArray(), 0);  
}

**说明**：iterator()会返回COWIterator对象。

COWIterator实现额ListIterator接口，它的源码如下：

static final class COWIterator<E> implements ListIterator<E> {  
 */\*\* Snapshot of the array \*/* private final Object[] snapshot;  
 */\*\* Index of element to be returned by subsequent call to next. \*/* private int cursor;  
  
 private COWIterator(Object[] elements, int initialCursor) {  
 cursor = initialCursor;  
 snapshot = elements;  
 }  
  
 public boolean hasNext() {  
 return cursor < snapshot.length;  
 }  
  
 public boolean hasPrevious() {  
 return cursor > 0;  
 }  
  
 @SuppressWarnings("unchecked")  
 public E next() {  
 if (! hasNext())  
 throw new NoSuchElementException();  
 return (E) snapshot[cursor++];  
 }  
  
 @SuppressWarnings("unchecked")  
 public E previous() {  
 if (! hasPrevious())  
 throw new NoSuchElementException();  
 return (E) snapshot[--cursor];  
 }  
  
 public int nextIndex() {  
 return cursor;  
 }  
  
 public int previousIndex() {  
 return cursor-1;  
 }  
  
 */\*\*  
 \* Not supported. Always throws UnsupportedOperationException.  
 \** ***@throws*** *UnsupportedOperationException always; {****@code*** *remove}  
 \* is not supported by this iterator.  
 \*/* public void remove() {  
 throw new UnsupportedOperationException();  
 }  
  
 */\*\*  
 \* Not supported. Always throws UnsupportedOperationException.  
 \** ***@throws*** *UnsupportedOperationException always; {****@code*** *set}  
 \* is not supported by this iterator.  
 \*/* public void set(E e) {  
 throw new UnsupportedOperationException();  
 }  
  
 */\*\*  
 \* Not supported. Always throws UnsupportedOperationException.  
 \** ***@throws*** *UnsupportedOperationException always; {****@code*** *add}  
 \* is not supported by this iterator.  
 \*/* public void add(E e) {  
 throw new UnsupportedOperationException();  
 }  
  
 @Override  
 public void forEachRemaining(Consumer<? super E> action) {  
 Objects.*requireNonNull*(action);  
 Object[] elements = snapshot;  
 final int size = elements.length;  
 for (int i = cursor; i < size; i++) {  
 @SuppressWarnings("unchecked") E e = (E) elements[i];  
 action.accept(e);  
 }  
 cursor = size;  
 }  
}

**说明**：COWIterator不支持修改元素的操作。例如，对于remove(),set(),add()等操作，COWIterator都会抛出异常！  
另外，需要提到的一点是，CopyOnWriteArrayList返回迭代器不会抛出ConcurrentModificationException异常，即它不是fail-fast机制的

#### CopyOnwriteArraySet

##### 概述

它是线程安全的无序的集合，可以将它理解成线程安全的HashSet。CopyOnWriteArraySet和HashSet虽然都继承于共同的父类AbstractSet；但是，HashSet是通过“散列表(HashMap)”实现的，而CopyOnWriteArraySet则是通过“动态数组(CopyOnWriteArrayList)”实现的，并不是散列表。  
和CopyOnWriteArrayList类似，CopyOnWriteArraySet具有以下特性：  
1. 它最适合于具有以下特征的应用程序：Set 大小通常保持很小，只读操作远多于可变操作，需要在遍历期间防止线程间的冲突。  
2. 它是线程安全的。  
3. 因为通常需要复制整个基础数组，所以可变操作（add()、set() 和 remove() 等等）的开销很大。  
4. 迭代器支持hasNext(), next()等不可变操作，但不支持可变 remove()等 操作。  
5. 使用迭代器进行遍历的速度很快，并且不会与其他线程发生冲突。在构造迭代器时，迭代器依赖于不变的数组快照。

###### 数据结构



**说明**：  
  1. CopyOnWriteArraySet继承于AbstractSet，这就意味着它是一个集合。  
  2. CopyOnWriteArraySet包含CopyOnWriteArrayList对象，它是通过CopyOnWriteArrayList实现的。而CopyOnWriteArrayList本质是个动态数组队列，所以CopyOnWriteArraySet相当于通过通过动态数组实现的“集合”. CopyOnWriteArrayList中允许有重复的元素；但是，CopyOnWriteArraySet是一个集合，所以它不能有重复集合。因此，CopyOnWriteArrayList额外提供了addIfAbsent()和addAllAbsent()这两个添加元素的API，通过这些API来添加元素时，只有当元素不存在时才执行添加操作！  
   至于CopyOnWriteArraySet的“线程安全”机制，和CopyOnWriteArrayList一样，是通过volatile和互斥锁来实现的。

###### 函数列表

// 创建一个空 set。  
CopyOnWriteArraySet()  
// 创建一个包含指定 collection 所有元素的 set。  
CopyOnWriteArraySet(Collection<? extends E> c)  
  
// 如果指定元素并不存在于此 set 中，则添加它。  
boolean add(E e)  
// 如果此 set 中没有指定 collection 中的所有元素，则将它们都添加到此 set 中。  
boolean addAll(Collection<? extends E> c)  
// 移除此 set 中的所有元素。  
void clear()  
// 如果此 set 包含指定元素，则返回 true。  
boolean contains(Object o)  
// 如果此 set 包含指定 collection 的所有元素，则返回 true。  
boolean containsAll(Collection<?> c)  
// 比较指定对象与此 set 的相等性。  
boolean equals(Object o)  
// 如果此 set 不包含任何元素，则返回 true。  
boolean isEmpty()  
// 返回按照元素添加顺序在此 set 中包含的元素上进行迭代的迭代器。  
Iterator<E> iterator()  
// 如果指定元素存在于此 set 中，则将其移除。  
boolean remove(Object o)  
// 移除此 set 中包含在指定 collection 中的所有元素。  
boolean removeAll(Collection<?> c)  
// 仅保留此 set 中那些包含在指定 collection 中的元素。  
boolean retainAll(Collection<?> c)  
// 返回此 set 中的元素数目。  
int size()  
// 返回一个包含此 set 所有元素的数组。  
Object[] toArray()  
// 返回一个包含此 set 所有元素的数组；返回数组的运行时类型是指定数组的类型。  
<T> T[] toArray(T[] a);

##### 源码解析

public boolean add(E e) {  
 return al.addIfAbsent(e);  
}  
public boolean remove(Object o) {  
 return al.remove(o);  
}

CopyOnWriteArraySet是通过CopyOwriteArrayList实现的,具体源码分析见[CopyOwriteArrayList篇源码解析](#_源码解析)

#### ConcurrentHashMap

##### 概述

ConcurrentHashMap是线程安全的哈希表。HashMap, Hashtable, ConcurrentHashMap之间的关联如下：

**HashMap**是非线程安全的哈希表，常用于单线程程序中。

**Hashtable**是线程安全的哈希表，它是通过synchronized来保证线程安全的；即，多线程通过同一个“对象的同步锁”来实现并发控制。Hashtable在线程竞争激烈时，效率比较低(此时建议使用ConcurrentHashMap)！因为当一个线程访问Hashtable的同步方法时，其它线程就访问Hashtable的同步方法时，可能会进入阻塞状态。

**ConcurrentHashMap**是线程安全的哈希表，1.7之前是通过“锁分段”来保证线程安全的,ConcurrentHashMap将哈希表分成许多片段(Segment)，每一个片段除了保存哈希表之外，本质上也是一个“可重入的互斥锁”(ReentrantLock)。多线程对同一个片段的访问，是互斥的；但是，对于不同片段的访问，却是可以同步进行的。1.8之后放弃了这种臃肿的设计,取而代之的是Node+CAS+synchronized来保证并发安全.



图4.4.1 jdk1.7实现方式



图4.4.2 jdk1.8实现方式

##### 函数列表

// 创建一个带有默认初始容量 (16)、加载因子 (0.75) 和 concurrencyLevel (16) 的新的空映射。  
ConcurrentHashMap()  
// 创建一个带有指定初始容量、默认加载因子 (0.75) 和 concurrencyLevel (16) 的新的空映射。  
ConcurrentHashMap(int initialCapacity)  
// 创建一个带有指定初始容量、加载因子和默认 concurrencyLevel (16) 的新的空映射。  
ConcurrentHashMap(int initialCapacity, float loadFactor)  
// 创建一个带有指定初始容量、加载因子和并发级别的新的空映射。  
ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)  
// 构造一个与给定映射具有相同映射关系的新映射。  
ConcurrentHashMap(Map<? extends K,? extends V> m)  
  
// 从该映射中移除所有映射关系  
void clear()  
// 一种遗留方法，测试此表中是否有一些与指定值存在映射关系的键。  
boolean contains(Object value)  
// 测试指定对象是否为此表中的键。  
boolean containsKey(Object key)  
// 如果此映射将一个或多个键映射到指定值，则返回 true。  
boolean containsValue(Object value)  
// 返回此表中值的枚举。  
Enumeration<V> elements()  
// 返回此映射所包含的映射关系的 Set 视图。  
Set<Map.Entry<K,V>> entrySet()  
// 返回指定键所映射到的值，如果此映射不包含该键的映射关系，则返回 null。  
V get(Object key)  
// 如果此映射不包含键-值映射关系，则返回 true。  
boolean isEmpty()  
// 返回此表中键的枚举。  
Enumeration<K> keys()  
// 返回此映射中包含的键的 Set 视图。  
Set<K> keySet()  
// 将指定键映射到此表中的指定值。  
V put(K key, V value)  
// 将指定映射中所有映射关系复制到此映射中。  
void putAll(Map<? extends K,? extends V> m)  
// 如果指定键已经不再与某个值相关联，则将它与给定值关联。  
V putIfAbsent(K key, V value)  
// 从此映射中移除键（及其相应的值）。  
V remove(Object key)  
// 只有目前将键的条目映射到给定值时，才移除该键的条目。  
boolean remove(Object key, Object value)  
// 只有目前将键的条目映射到某一值时，才替换该键的条目。  
V replace(K key, V value)  
// 只有目前将键的条目映射到给定值时，才替换该键的条目。  
boolean replace(K key, V oldValue, V newValue)  
// 返回此映射中的键-值映射关系数。  
int size()  
// 返回此映射中包含的值的 Collection 视图。  
Collection<V> values();

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*以下为1.8新增方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
  
/\*\*对于指定key做remappingFunction函数调用，remappingFunction函数返回值即为新的value，  
如果返回值为null，则从map中删除对应的key。compute返回key更新后的值（remappingFunction函数返回值）\*/*public V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction);  
  
*/\*\*如果指定的key不存在，对该key做mappingFunction函数操作，mappingFunction函数返回值不为null，则将对应的k-v放到map中，否则不操作。  
如果key存在返回key对应的value（此时mappingFunction不会调用）\*/*public V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction);  
*/\*\*类似与computeIfAbsent，仅对已经存在的key才计算新value。同样，如果remappingFunction返回值为null，会删除对应的k-v。\*/*public V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction);  
*/\*\*当key不存在，直接插入对应value，remappingFunction不会被调用；否则，对oldValue与value做remappingFunction函数，结果作为新的newValue插入到map中。  
 同样null结果会删除对应的k-v。\*/*public V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction);

##### 数据结构



###### 重要属性

*/\*\*存放Node元素的数组,在第一次插入数据时初始化\*/*transient volatile Node<K,V>[] table;  
  
*/\*\*一个过渡的table表 只有在扩容的时候才会使用 \*/*private transient volatile Node<K,V>[] nextTable;  
  
*/\*\*  
 \* Base counter value, used mainly when there is no contention,  
 \* but also as a fallback during table initialization  
 \* races. Updated via CAS.  
 \*/*private transient volatile long baseCount;  
  
*/\*\*  
 \* Table initialization and resizing control. When negative, the  
 \* table is being initialized or resized: -1 for initialization,  
 \* else -(1 + the number of active resizing threads). Otherwise,  
 \* when table is null, holds the initial table size to use upon  
 \* creation, or 0 for default. After initialization, holds the  
 \* next element count value upon which to resize the table.  
 \* 用来标识table的初始化和扩容操作  
 负数代表正在进行初始化或扩容操作  
 -1代表正在初始化  
 -N 表示有N-1个线程正在进行扩容操作  
 正数或0代表table还没有被初始化，这个数值表示初始化或下一次进行扩容的大小  
 如果获得了初始化权限，就用CAS方法将sizeCtl置为-1，防止其他线程进入。  
 初始化数组后，将sizeCtl的值改为0.75\*n（n - (n >>> 2)）  
 \*/*private transient volatile int sizeCtl;  
  
*/\*\*  
 \* The next table index (plus one) to split while resizing.  
 \*  
 \* 初值为最后一个桶，表示从transferIndex开始到后面所有的桶的迁移任务已经被分配出去了。  
 \* 所以每次线程领取扩容任务，则需要对该属性进行CAS的减操作，即一般是transferIndex-stride。  
 \*/*private transient volatile int transferIndex;

*/\*\*  
 \* Spinlock (locked via CAS) used when resizing and/or creating CounterCells.  
 \* 元素变化时用与锁控制  
 \*/*private transient volatile int cellsBusy;  
  
*/\*\*  
 \* Table of counter cells. When non-null, size is a power of 2.  
 \* 保存部分元素的变化个数  
 \* size = baseCount + CounterCell[i].value  
 \*/*private transient volatile CounterCell[] counterCells;

**说明:**

**table**：默认为null，初始化发生在第一次插入操作，默认大小为16的数组，用来存储Node节点数据，扩容时大小总是2的幂次方。

**nextTable**：默认为null，扩容时新生成的数组，其大小为原数组的两倍。

**sizeCtl** ：默认为0，用来控制table的初始化和扩容操作

* **-1** 代表table正在初始化
* **-N** 表示有N-1个线程正在进行扩容操作
* 正数或0代表table还没有被初始化,表示初始化或下一次扩容的大小
* 初始化数组后,将saizeCtl的值改为0.75\*n (n-(n >>> 2))

**transferIndex**：扩容时用到,初始时为table.length，表示剩余还未分配出去的扩容任务, 所以每次线程领取扩容任务后,需要对该属性进行CAS减操作(transferIndex-stride)

###### Node<K, V>

*/\*\*  
 \* Key-value entry. This class is never exported out as a  
 \* user-mutable Map.Entry (i.e., one supporting setValue; see  
 \* MapEntry below), but can be used for read-only traversals used  
 \* in bulk tasks. Subclasses of Node with a negative hash field  
 \* are special, and contain null keys and values (but are never  
 \* exported). Otherwise, keys and vals are never null.  
 \* 这个Node内部类与HashMap中定义的Node类很相似，但是有一些差别  
 它对value和next属性设置了volatile同步锁  
 \*/*static class Node<K,V> implements Map.Entry<K,V> {  
 final int hash;  
 final K key;  
 volatile V val;  
 volatile Node<K,V> next;  
  
 Node(int hash, K key, V val, Node<K,V> next) {  
 this.hash = hash;  
 this.key = key;  
 this.val = val;  
 this.next = next;  
 }  
  
 public final K getKey() { return key; }  
 public final V getValue() { return val; }  
 public final int hashCode() { return key.hashCode() ^ val.hashCode(); }  
 public final String toString(){ return key + "=" + val; }  
 //不允许直接改变value的值  
 public final V setValue(V value) {  
 throw new UnsupportedOperationException();  
 }  
  
 public final boolean equals(Object o) {  
 Object k, v, u; Map.Entry<?,?> e;  
 return ((o instanceof Map.Entry) &&  
 (k = (e = (Map.Entry<?,?>)o).getKey()) != null &&  
 (v = e.getValue()) != null &&  
 (k == key || k.equals(key)) &&  
 (v == (u = val) || v.equals(u)));  
 }  
  
 */\*\*  
 \* Virtualized support for map.get(); overridden in subclasses.  
 \* 提供find方法辅助map.get()  
 \*/* Node<K,V> find(int h, Object k) {  
 Node<K,V> e = this;  
 if (k != null) {  
 do {  
 K ek;  
 if (e.hash == h &&  
 ((ek = e.key) == k || (ek != null && k.equals(ek))))  
 return e;  
 } while ((e = e.next) != null);  
 }  
 return null;  
 }  
}

**说明:** 保存k-v,k的hash值和链表的下一个节点next,其中V和next用volatile修饰,保证多线程环境下的可见性.

###### TreeNode<K, V>

*/\*\*  
 \* Nodes for use in TreeBins  
 \* 树节点类，另外一个核心的数据结构。当链表长度过长的时候，会转换为TreeNode。  
 \* 但是与HashMap不相同的是，它并不是直接转换为红黑树，而是把这些结点包装成TreeNode放在TreeBin对象中，  
 \* 由TreeBin完成对红黑树的包装。而且TreeNode在ConcurrentHashMap集成自Node类，  
 \* 而并非HashMap中的集成自LinkedHashMap.Entry<K,V>类，也就是说TreeNode带有next指针，  
 \* 这样做的目的是方便基于TreeBin的访问。  
 \*/*static final class TreeNode<K,V> extends Node<K,V> {  
 TreeNode<K,V> parent; // red-black tree links  
 TreeNode<K,V> left;  
 TreeNode<K,V> right;  
 TreeNode<K,V> prev; // needed to unlink next upon deletion  
 boolean red;  
  
 TreeNode(int hash, K key, V val, Node<K,V> next,  
 TreeNode<K,V> parent) {  
 super(hash, key, val, next);  
 this.parent = parent;  
 }  
  
 Node<K,V> find(int h, Object k) {  
 return findTreeNode(h, k, null);  
 }  
  
 */\*\*  
 \* Returns the TreeNode (or null if not found) for the given key  
 \* starting at given root.  
 \*/* final TreeNode<K,V> findTreeNode(int h, Object k, Class<?> kc) {  
 if (k != null) {  
 TreeNode<K,V> p = this;  
 do {  
 int ph, dir; K pk; TreeNode<K,V> q;  
 TreeNode<K,V> pl = p.left, pr = p.right;  
 if ((ph = p.hash) > h)  
 p = pl;  
 else if (ph < h)  
 p = pr;  
 else if ((pk = p.key) == k || (pk != null && k.equals(pk)))  
 return p;  
 else if (pl == null)  
 p = pr;  
 else if (pr == null)  
 p = pl;  
 else if ((kc != null ||  
 (kc = *comparableClassFor*(k)) != null) &&  
 (dir = *compareComparables*(kc, k, pk)) != 0)  
 p = (dir < 0) ? pl : pr;  
 else if ((q = pr.findTreeNode(h, k, kc)) != null)  
 return q;  
 else  
 p = pl;  
 } while (p != null);  
 }  
 return null;  
 }  
}

**说明:** 红黑树节点类,当链表长度>=8且数组长度>=64时,node会转为treeNode,但它不是直接转为红黑树,而是把这些treeNode节点放入treeBin对象中,由treeBin完成红黑树的包装

###### TreeBin<K, V>

*/\*\*  
 \* TreeNodes used at the heads of bins. TreeBins do not hold user  
 \* keys or values, but instead point to list of TreeNodes and  
 \* their root. They also maintain a parasitic read-write lock  
 \* forcing writers (who hold bin lock) to wait for readers (who do  
 \* not) to complete before tree restructuring operations.  
 \*  
 \* 这个类并不负责包装用户的key、value信息，而是包装的很多TreeNode节点。  
 \* 它代替了TreeNode的根节点，也就是说在实际的ConcurrentHashMap“数组”中，  
 \* 存放的是TreeBin对象，而不是TreeNode对象，这是与HashMap的区别。另外这个类还带有了读写锁。  
 \*/*static final class TreeBin<K,V> extends Node<K,V> {  
 TreeNode<K,V> root;  
 volatile TreeNode<K,V> first;  
 volatile Thread waiter;  
 volatile int lockState;  
 // values for lockState  
 static final int *WRITER* = 1; // set while holding write lock  
 static final int *WAITER* = 2; // set when waiting for write lock  
 static final int *READER* = 4; // increment value for setting read lock

*//省略部分代码*

说明: 封装了TreeNode节点,代替了TreeNode的根节点,也就是说在concurrentHashMap中红黑树存储的是TreeBin对象,这也是与HashMap的区别之一

###### ForwardingNode<K, V>

*/\*\*  
 \* A node inserted at head of bins during transfer operations.  
 \*  
 \* 一个用于连接两个table的节点类。它包含一个nextTable指针，用于指向下一张表。  
 \* 而且这个节点的key value next指针全部为null，它的hash值为-1.  
 \* 这里面定义的find的方法是从nextTable里进行查询节点，而不是以自身为头节点进行查找  
 \* 只有在扩容时发挥作用,作为一个占位符放在table中表示当前节点为null或者已经被移动  
 \*/*static final class ForwardingNode<K,V> extends Node<K,V> {  
 final Node<K,V>[] nextTable;  
 ForwardingNode(Node<K,V>[] tab) {  
 super(*MOVED*, null, null, null);  
 this.nextTable = tab;  
 }  
  
 Node<K,V> find(int h, Object k) {  
 // loop to avoid arbitrarily deep recursion on forwarding nodes  
 outer: for (Node<K,V>[] tab = nextTable;;) {  
 Node<K,V> e; int n;  
 if (k == null || tab == null || (n = tab.length) == 0 ||  
 (e = *tabAt*(tab, (n - 1) & h)) == null)  
 return null;  
 for (;;) {  
 int eh; K ek;  
 if ((eh = e.hash) == h &&  
 ((ek = e.key) == k || (ek != null && k.equals(ek))))  
 return e;  
 if (eh < 0) {  
 if (e instanceof ForwardingNode) {  
 tab = ((ForwardingNode<K,V>)e).nextTable;  
 continue outer;  
 }  
 else  
 return e.find(h, k);  
 }  
 if ((e = e.next) == null)  
 return null;  
 }  
 }  
 }  
}

**说明:** 扩容时用到,作为一个占位符放到table中表示当前节点已经被处理

##### 源码解析

###### 构造方法

*/\*\*  
 \* Creates a new, empty map with the default initial table size (16).  
 \*/*public ConcurrentHashMap() {  
}  
  
*/\*\*  
 \* Creates a new, empty map with an initial table size  
 \* accommodating the specified number of elements without the need  
 \* to dynamically resize.  
 \*  
 \** ***@param*** *initialCapacity The implementation performs internal  
 \* sizing to accommodate this many elements.  
 \** ***@throws*** *IllegalArgumentException if the initial capacity of  
 \* elements is negative  
 \*/*public ConcurrentHashMap(int initialCapacity) {  
 if (initialCapacity < 0)  
 throw new IllegalArgumentException();  
 //初始化容量,通过tableSizeFor计算,确保为2的幂次方  
 int cap = ((initialCapacity >= (*MAXIMUM\_CAPACITY* >>> 1)) ?  
 *MAXIMUM\_CAPACITY* :  
 *tableSizeFor*(initialCapacity + (initialCapacity >>> 1) + 1));  
 this.sizeCtl = cap;  
}  
  
*/\*\*  
 \* Creates a new map with the same mappings as the given map.  
 \*  
 \** ***@param*** *m the map  
 \*/*public ConcurrentHashMap(Map<? extends K, ? extends V> m) {  
 this.sizeCtl = *DEFAULT\_CAPACITY*;  
 putAll(m);  
}  
  
*/\*\*  
 \* Creates a new, empty map with an initial table size based on  
 \* the given number of elements ({****@code*** *initialCapacity}) and  
 \* initial table density ({****@code*** *loadFactor}).  
 \*  
 \** ***@param*** *initialCapacity the initial capacity. The implementation  
 \* performs internal sizing to accommodate this many elements,  
 \* given the specified load factor.  
 \** ***@param*** *loadFactor the load factor (table density) for  
 \* establishing the initial table size  
 \** ***@throws*** *IllegalArgumentException if the initial capacity of  
 \* elements is negative or the load factor is nonpositive  
 \*  
 \** ***@since*** *1.6  
 \*/*public ConcurrentHashMap(int initialCapacity, float loadFactor) {  
 this(initialCapacity, loadFactor, 1);  
}  
  
*/\*\*  
 \* Creates a new, empty map with an initial table size based on  
 \* the given number of elements ({****@code*** *initialCapacity}), table  
 \* density ({****@code*** *loadFactor}), and number of concurrently  
 \* updating threads ({****@code*** *concurrencyLevel}).  
 \*  
 \** ***@param*** *initialCapacity the initial capacity. The implementation  
 \* performs internal sizing to accommodate this many elements,  
 \* given the specified load factor.  
 \** ***@param*** *loadFactor the load factor (table density) for  
 \* establishing the initial table size  
 \** ***@param*** *concurrencyLevel the estimated number of concurrently  
 \* updating threads. The implementation may use this value as  
 \* a sizing hint.  
 \** ***@throws*** *IllegalArgumentException if the initial capacity is  
 \* negative or the load factor or concurrencyLevel are  
 \* nonpositive  
 \*/*public ConcurrentHashMap(int initialCapacity,  
 float loadFactor, int concurrencyLevel) {  
 if (!(loadFactor > 0.0f) || initialCapacity < 0 || concurrencyLevel <= 0)  
 throw new IllegalArgumentException();  
 if (initialCapacity < concurrencyLevel) // Use at least as many bins  
 initialCapacity = concurrencyLevel; // as estimated threads  
 long size = (long)(1.0 + (long)initialCapacity / loadFactor);  
 int cap = (size >= (long)*MAXIMUM\_CAPACITY*) ?  
 *MAXIMUM\_CAPACITY* : *tableSizeFor*((int)size);  
 this.sizeCtl = cap;  
}

**说明:** 可以看到简单构造中并没有初始化table,table的初始化放到了第一次put

实例化ConcurrentHashMap时带参数时，会根据参数调整table的大小，假设参数为100，最终会调整成256，确保table的大小总是2的幂次方,算法源码:

private static final int tableSizeFor(int c) {  
 int n = c - 1;  
 n |= n >>> 1;  
 n |= n >>> 2;  
 n |= n >>> 4;  
 n |= n >>> 8;  
 n |= n >>> 16;  
 return (n < 0) ? 1 : (n >= *MAXIMUM\_CAPACITY*) ? *MAXIMUM\_CAPACITY* : n + 1;  
}

###### put(K, V)

public V put(K key, V value) {  
 return putVal(key, value, false);  
}  
  
*/\*\* Implementation for put and putIfAbsent \*/*final V putVal(K key, V value, boolean onlyIfAbsent) {  
 if (key == null || value == null) throw new NullPointerException();  
 //计算hash值  
 int hash = *spread*(key.hashCode());  
 int binCount = 0;//  
 for (Node<K,V>[] tab = table;;) {//保证拿到最新的table,然后自旋  
 //f:索引元素; n:tab.length; i:(n - 1) & hash; fh:f.hash  
 Node<K,V> f; int n, i, fh;  
 if (tab == null || (n = tab.length) == 0)  
 //初始化Node  
 tab = initTable();  
 //通过hash找到在node中的位置(tabAt),f=索引元素  
 //如果还未初始化则通过cas操作(casTabAt)插入相应数据  
 else if ((f = *tabAt*(tab, i = (n - 1) & hash)) == null) {  
 //cas插入元素,成功则跳出循环  
 if (*casTabAt*(tab, i, null,  
 new Node<K,V>(hash, key, value, null)))  
 break; // no lock when adding to empty bin  
 }  
 //当前节点处于移动状态-其他线程正在进行扩容操作  
 else if ((fh = f.hash) == *MOVED*)  
 //一起进行tab扩容  
 tab = helpTransfer(tab, f);  
 else {  
 V oldVal = null;  
 //使用内置锁synchronized控制并发  
 synchronized (f) {  
 //在节点f上进行同步，节点插入之前，再次利用tabAt(tab, i) == f判断，防止被其它线程修改。  
 if (*tabAt*(tab, i) == f) {  
 //f.hash(fh)>=0,说明f是链表的头结点  
 if (fh >= 0) {  
 binCount = 1;  
 for (Node<K,V> e = f;; ++binCount) {  
 K ek;  
 //修改  
 if (e.hash == hash &&  
 ((ek = e.key) == key ||  
 (ek != null && key.equals(ek)))) {  
 oldVal = e.val;  
 if (!onlyIfAbsent)  
 e.val = value;  
 break;  
 }  
 Node<K,V> pred = e;  
 //如果遍历到了最后一个结点，那么就证明新的节点需要插入 就把它插入在链表尾部  
 if ((e = e.next) == null) {  
 pred.next = new Node<K,V>(hash, key,  
 value, null);  
 break;  
 }  
 }  
 }  
 //红黑树节点操作  
 else if (f instanceof TreeBin) {  
 Node<K,V> p;  
 binCount = 2;  
 if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,  
 value)) != null) {  
 oldVal = p.val;  
 if (!onlyIfAbsent)  
 p.val = value;  
 }  
 }  
 }  
 }  
 if (binCount != 0) {  
 //如果链表中节点数binCount >= TREEIFY\_THRESHOLD(默认是8)，则把链表转化为红黑树结构  
 if (binCount >= *TREEIFY\_THRESHOLD*)  
 treeifyBin(tab, i);  
 if (oldVal != null)  
 return oldVal;  
 break;  
 }  
 }  
 }  
 //尝试更新新元素个数  
 addCount(1L, binCount);  
 return null;  
}

**说明:** put方法依然沿用HashMap的put方法的思想，根据hash值计算这个新插入的点在table中的位置i，如果i位置是空的，直接放进去，否则进行判断，如果i位置是树节点，按照树的方式插入新的节点，否则把i插入到链表的末尾。

ConcurrentHashMap中依然沿用这个思想，有一个最重要的不同点就是ConcurrentHashMap不允许key或value为null值。另外由于涉及到多线程，put方法就要复杂一点。在多线程中可能有以下两个情况:

4.4.4.2.1 initTable()

* 1. 初始化操作,ConcurrentHashMap会在第一次插入的时候(判断table是否为null)进行table的初始化,源码如下:

*/\*\*  
 \* Initializes table, using the size recorded in sizeCtl.  
 \*/*private final Node<K,V>[] initTable() {  
 Node<K,V>[] tab; int sc;  
 while ((tab = table) == null || tab.length == 0) {  
 //如果一个线程发现sizeCtl<0，意味着另外的线程执行CAS操作成功，当前线程只需要让出cpu时间片  
 if ((sc = sizeCtl) < 0)  
 Thread.*yield*(); // lost initialization race; just spin  
 else if (*U*.compareAndSwapInt(this, *SIZECTL*, sc, -1)) {//利用CAS方法把sizectl的值置为-1 表示本线程正在进行初始化  
 try {  
 if ((tab = table) == null || tab.length == 0) {  
 int n = (sc > 0) ? sc : *DEFAULT\_CAPACITY*;  
 @SuppressWarnings("unchecked")  
 Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];  
 table = tab = nt;  
 sc = n - (n >>> 2);//相当于0.75\*n 设置一个扩容的阈值  
 }  
 } finally {  
 sizeCtl = sc;//扩容完成后,将sizeCtl设为0.75\*n  
 }  
 break;  
 }  
 }  
 return tab;  
}

4.4.4.2.2 helpTransfer(Node<K,V>[], Node<K,V>)

* 1. 如果一个或多个线程正在对ConcurrentHashMap进行扩容操作，当前线程也要进入扩容的操作中。这个扩容的操作之所以能被检测到，是因为transfer方法中在空结点上插入forward节点，如果检测到需要插入的位置被forward节点占有，就调用helpTransfer()方法帮助进行扩容,源码如下:

final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {  
 Node<K,V>[] nextTab; int sc;  
 if (tab != null && (f instanceof ForwardingNode) &&  
 (nextTab = ((ForwardingNode<K,V>)f).nextTable) != null) {  
 //计算操作栈校验码  
 int rs = *resizeStamp*(tab.length);  
 while (nextTab == nextTable && table == tab &&  
 (sc = sizeCtl) < 0) {  
 if ((sc >>> *RESIZE\_STAMP\_SHIFT*) != rs || sc == rs + 1 ||  
 sc == rs + *MAX\_RESIZERS* || transferIndex <= 0)  
 break;  
 if (*U*.compareAndSwapInt(this, *SIZECTL*, sc, sc + 1)) {  
 transfer(tab, nextTab);  
 break;  
 }  
 }  
 return nextTab;  
 }  
 return table;  
}

4.4.4.2.3 treeifyBin(Node<K,V>[] , index)

* 1. 如果检测到要插入的节点是非空且不是forward节点，就对这个节点加锁，这样就保证了线程安全。尽管这个有一些影响效率，但是还是会比hashTable的synchronized要好得多。

整体流程就是首先利用spread方法对key的hashcode进行一次hash计算，由此来确定这个值在table中的位置。

如果这个位置是空的，那么直接放入，而且不需要加锁操作。

如果这个位置存在结点，说明发生了hash碰撞，首先判断这个节点的类型。如果是链表节点（fh>0）,则得到的结点就是hash值相同的节点组成的链表的头节点。需要依次向后遍历确定这个新加入的值所在位置。如果遇到hash值与key值都与新加入节点是一致的情况，则只需要更新value值即可。否则依次向后遍历，直到链表尾插入这个结点。如果这个节点的类型已经是树节点的话，直接调用树节点的插入方法进行插入新的值. 如果加入链表节点以后链表长度大于8，就调用treeifyBin() 方法来决定扩容还是把当前节点链表转为红黑树,源码:

private final void treeifyBin(Node<K,V>[] tab, int index) {  
 Node<K,V> b; int n, sc;  
 if (tab != null) {  
 //当数组长度还未超过64,优先数组的扩容,否则将链表转为红黑树  
 if ((n = tab.length) < *MIN\_TREEIFY\_CAPACITY*)  
 //两倍扩容  
 tryPresize(n << 1);  
 else if ((b = *tabAt*(tab, index)) != null && b.hash >= 0) {  
 synchronized (b) {  
 if (*tabAt*(tab, index) == b) {  
 TreeNode<K,V> hd = null, tl = null;  
 for (Node<K,V> e = b; e != null; e = e.next) {  
 TreeNode<K,V> p =  
 new TreeNode<K,V>(e.hash, e.key, e.val,  
 null, null);  
 if ((p.prev = tl) == null)  
 hd = p;  
 else  
 tl.next = p;  
 tl = p;  
 }  
 *setTabAt*(tab, index, new TreeBin<K,V>(hd));  
 }  
 }  
 }  
 }  
}

**说明:** 从源码中可以看出,当数组长度未超过MIN\_TREEIFY\_CAPACITY = 64时,就调用tryPresize()尝试对对table进行扩容,源码如下:

*/\*\*  
 \* Tries to presize table to accommodate the given number of elements.  
 \*  
 \** ***@param*** *size number of elements (doesn't need to be perfectly accurate)  
 \*/*private final void tryPresize(int size) {  
 int c = (size >= (*MAXIMUM\_CAPACITY* >>> 1)) ? *MAXIMUM\_CAPACITY* :  
 *tableSizeFor*(size + (size >>> 1) + 1);  
 int sc;  
 while ((sc = sizeCtl) >= 0) {  
 Node<K,V>[] tab = table; int n;  
 //未初始化  
 if (tab == null || (n = tab.length) == 0) {  
 n = (sc > c) ? sc : c;  
 if (*U*.compareAndSwapInt(this, *SIZECTL*, sc, -1)) {  
 try {  
 if (table == tab) {  
 @SuppressWarnings("unchecked")  
 Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];  
 table = nt;  
 sc = n - (n >>> 2);  
 }  
 } finally {  
 sizeCtl = sc;  
 }  
 }  
 }  
 //已达到最大容量  
 else if (c <= sc || n >= *MAXIMUM\_CAPACITY*)  
 break;  
 else if (tab == table) {  
 int rs = *resizeStamp*(n);  
 //正在进行扩容操作  
 if (sc < 0) {  
 Node<K,V>[] nt;  
 if ((sc >>> *RESIZE\_STAMP\_SHIFT*) != rs || sc == rs + 1 ||  
 sc == rs + *MAX\_RESIZERS* || (nt = nextTable) == null ||  
 transferIndex <= 0)  
 break;  
 if (*U*.compareAndSwapInt(this, *SIZECTL*, sc, sc + 1))  
 transfer(tab, nt);  
 }  
 else if (*U*.compareAndSwapInt(this, *SIZECTL*, sc,  
 (rs << *RESIZE\_STAMP\_SHIFT*) + 2))  
 transfer(tab, null);  
 }  
 }  
}

4.4.4.2.4 addCount(long,int)

* 1. 最后调用addCount(long, int)方法尝试更新元素个数.源码如下:

*/\*\*  
 \* 更新元素个数  
 \* 可能产生扩容操作(@param check>=0)  
 \* \*/*private final void addCount(long x, int check) {  
 CounterCell[] as; long b, s;  
 if ((as = counterCells) != null ||  
 !*U*.compareAndSwapLong(this, *BASECOUNT*, b = baseCount, s = b + x)) {  
 //未成功执行到CAS的线程会执行以下逻辑  
 CounterCell a; long v; int m;  
 boolean uncontended = true;  
 if (as == null || (m = as.length - 1) < 0 ||  
 (a = as[ThreadLocalRandom.*getProbe*() & m]) == null ||  
 !(uncontended =  
 *U*.compareAndSwapLong(a, *CELLVALUE*, v = a.value, v + x))) {  
 fullAddCount(x, uncontended);  
 return;  
 }  
 if (check <= 1)  
 return;  
 s = sumCount();  
 }  
 if (check >= 0) {  
 Node<K,V>[] tab, nt; int n, sc;  
 while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&  
 (n = tab.length) < *MAXIMUM\_CAPACITY*) {  
 int rs = *resizeStamp*(n);  
 if (sc < 0) {  
 if ((sc >>> *RESIZE\_STAMP\_SHIFT*) != rs || sc == rs + 1 ||  
 sc == rs + *MAX\_RESIZERS* || (nt = nextTable) == null ||  
 transferIndex <= 0)  
 break;  
 if (*U*.compareAndSwapInt(this, *SIZECTL*, sc, sc + 1))  
 transfer(tab, nt);  
 }  
 else if (*U*.compareAndSwapInt(this, *SIZECTL*, sc,  
 (rs << *RESIZE\_STAMP\_SHIFT*) + 2))  
 transfer(tab, null);  
 s = sumCount();  
 }  
 }  
}

4.4.4.2.5 transfer(Node<K,V> tab, Node<K,V> nextTab)

**发生扩容的条件:**

* 当某个Node的链表节点数>=TREEIFY\_THRESHOLD(8)时:

调用treeifyBin方法尝试把链表转为红黑树,如果此时数组长度小于MIN\_TREEIFY\_CAPACITY(64)时优先进行扩容;否则转换链表为红黑树. 源码如下:

//当数组长度还未超过64,优先数组的扩容,否则将链表转为红黑树  
if ((n = tab.length) < *MIN\_TREEIFY\_CAPACITY*)  
 //两倍扩容  
 tryPresize(n << 1);  
else if ((b = *tabAt*(tab, index)) != null && b.hash >= 0) {  
 synchronized (b) {  
 if (*tabAt*(tab, index) == b) {  
 TreeNode<K,V> hd = null, tl = null;  
 //遍历找到头节点  
 for (Node<K,V> e = b; e != null; e = e.next) {  
 TreeNode<K,V> p =  
 new TreeNode<K,V>(e.hash, e.key, e.val,  
 null, null);  
 if ((p.prev = tl) == null)  
 hd = p;  
 else  
 tl.next = p;  
 tl = p;  
 }  
 *setTabAt*(tab, index, new TreeBin<K,V>(hd));  
 }  
 }  
}

* 所有节点元素总数(通过sumCount()计算) > sizeCtl( 0.75\*n )时进行扩容操作

并且数组长度不超过MIN\_TREEIFY\_CAPACITY(64)时进行扩容操作,

整个扩容操作分为两部分:

* 1. 构建一个nextTable,它的容量是原来的两倍
  2. 将原来table中的元素复制到nextTable中，这里允许多线程进行操作。

具体源码如下:

private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {  
 int n = tab.length, stride;  
 //任务分片处理  
 if ((stride = (*NCPU* > 1) ? (n >>> 3) / *NCPU* : n) < *MIN\_TRANSFER\_STRIDE*)  
 stride = *MIN\_TRANSFER\_STRIDE*; // subdivide range  
 if (nextTab == null) { // initiating  
 try {  
 //根据当前数组长度,新建一个两倍长度的数组nextTab  
 @SuppressWarnings("unchecked")  
 Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];  
 nextTab = nt;  
 } catch (Throwable ex) { // try to cope with OOME  
 sizeCtl = Integer.*MAX\_VALUE*;  
 return;  
 }  
 nextTable = nextTab;  
 transferIndex = n;  
 }  
 int nextn = nextTab.length;  
 //初始化ForwardingNode节点,持有nextTab的引用,在处理完每个槽位的节点之后当做占位节点，表示该槽位已经处理过了；  
 ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);  
 boolean advance = true;//并发扩容的关键属性 如果等于true 说明这个节点已经处理过  
 boolean finishing = false; // to ensure sweep before committing nextTab  
 //通过for自循环处理每个槽位中的链表元素，默认advace为true，通过CAS设置transferIndex属性值，  
 // 并初始化i和bound值，i指当前处理的槽位序号，bound指需要当前线程处理的扩容任务数  
 for (int i = 0, bound = 0;;) {  
 //f:当前处理i位置的node; fh:f.hash  
 Node<K,V> f; int fh;  
 //这个while循环体的作用就是在控制i-- 通过i--可以依次遍历原hash表中的节点  
 while (advance) {  
 //nextIndex:剩余未分配的任务; nextBound:领取的扩容任务数  
 int nextIndex, nextBound;  
 if (--i >= bound || finishing)  
 advance = false;  
 //全部任务已经被分配出去  
 else if ((nextIndex = transferIndex) <= 0) {  
 i = -1;  
 advance = false;  
 }  
 //transferIndex（初值为最后一个桶），表示从transferIndex开始到后面所有的桶的迁移任务已经被分配出去了。  
 // 所以每次线程领取扩容任务，则需要对该属性进行CAS的减操作，即一般是transferIndex-stride。  
 else if (*U*.compareAndSwapInt  
 (this, *TRANSFERINDEX*, nextIndex,  
 nextBound = (nextIndex > stride ?  
 nextIndex - stride : 0))) {  
 bound = nextBound;  
 i = nextIndex - 1;  
 advance = false;  
 }  
 }  
 if (i < 0 || i >= n || i + n >= nextn) {  
 int sc;  
 if (finishing) {  
 //如果所有的节点都已经完成复制工作 就把nextTab赋值给table 清空过渡对象nextTable  
 nextTable = null;  
 table = nextTab;  
 sizeCtl = (n << 1) - (n >>> 1);//1.5\*n 扩容阈值设置为原来容量的1.5倍 依然相当于现在容量的0.75倍  
 return;  
 }  
 //参与扩容的线程未全部结束时,已经结束扩容任务的线程的处理逻辑  
 //每个线程完成自己的扩容操作后就对sizeCtl-1  
 if (*U*.compareAndSwapInt(this, *SIZECTL*, sc = sizeCtl, sc - 1)) {  
 //判断是否全部任务已经完成,sizeCtl初始值为一个负值=(rs << RESIZE\_STAMP\_SHIFT) + 2)  
 if ((sc - 2) != *resizeStamp*(n) << *RESIZE\_STAMP\_SHIFT*)  
 return;  
 finishing = advance = true;  
 i = n; // recheck before commit  
 }  
 }  
 //如果需要处理的槽位边界中没有节点，则通过CAS插入在上面初始化的ForwardingNode节点，用于告诉其它线程该槽位已经处理过了  
 else if ((f = *tabAt*(tab, i)) == null)  
 advance = *casTabAt*(tab, i, null, fwd);  
 //如果槽位边界已经被其他线程处理了，那么当前线程处理到这个节点时，取到该节点的hash值应该为MOVED，则直接跳过，继续处理下一个槽位的节点  
 else if ((fh = f.hash) == *MOVED*)  
 advance = true; // already processed  
 else {  
 synchronized (f) {  
 if (*tabAt*(tab, i) == f) {  
 //处理当前槽位的节点,构建两个node:ln/hn,分别放在低位和高位  
 //ln:原位置; hn:i+n位置  
 Node<K,V> ln, hn;  
 //如果这个位置是Node(链表)节点（fh>=0）  
 if (fh >= 0) {  
 //使用fn&n快速把链表中的元素分成两份  
 //原槽位i上的节点会分为两类，设原表长度为n=2^x，根据槽位计算公式i=(n-1)&hash，在表扩张后，  
 // 若hash在第x位是1，那么槽位就是(n-1)&hash+2^x，若hash在第x为是0，那么槽位不变。  
 int runBit = fh & n;  
 //最下一个节点  
 Node<K,V> lastRun = f;  
 //获取当前处理节点f的最下一个节点  
 for (Node<K,V> p = f.next; p != null; p = p.next) {  
 int b = p.hash & n;  
 if (b != runBit) {  
 runBit = b;  
 lastRun = p;  
 }  
 }  
 if (runBit == 0) {  
 ln = lastRun;  
 hn = null;  
 }  
 else {  
 hn = lastRun;  
 ln = null;  
 }  
 for (Node<K,V> p = f; p != lastRun; p = p.next) {  
 int ph = p.hash; K pk = p.key; V pv = p.val;  
 //在原位置  
 if ((ph & n) == 0)  
 ln = new Node<K,V>(ph, pk, pv, ln);  
 //i+n位置  
 else  
 hn = new Node<K,V>(ph, pk, pv, hn);  
 }  
 //nextTab的i位置插入一个链表  
 *setTabAt*(nextTab, i, ln);  
 //nextTab的i+n位置插入一个链表  
 *setTabAt*(nextTab, i + n, hn);  
 //在table的i位置上插入forwardNode节点 表示已经处理过该节点  
 *setTabAt*(tab, i, fwd);  
 advance = true;  
 }  
 */\*\*  
 \* 如果该槽位是红黑树结构，则构造树节点lo和hi，遍历红黑树中的节点，同样根据hash&tab.length算法，  
 \* 把节点分为两类，分别插入到lo和hi为头的链表中，根据lo和hi链表中的元素个数分别生成ln和hn节点，  
 \* 其中ln节点的生成逻辑如下：  
 \*（1）如果lo链表的元素个数小于等于UNTREEIFY\_THRESHOLD，默认为6，则通过untreeify方法把树节点链表转化成普通节点链表  
 \*（2）否则判断hi链表中的元素个数是否等于0：如果等于0，表示lo链表中包含了所有原始节点，则设置原始红黑树给ln，  
 \* 否则根据lo链表重新构造红黑树。  
 \*/* else if (f instanceof TreeBin) {  
 //转为头结点  
 TreeBin<K,V> t = (TreeBin<K,V>)f;  
 TreeNode<K,V> lo = null, loTail = null;  
 TreeNode<K,V> hi = null, hiTail = null;  
 int lc = 0, hc = 0;  
 //  
 for (Node<K,V> e = t.first; e != null; e = e.next) {  
 int h = e.hash;  
 TreeNode<K,V> p = new TreeNode<K,V>  
 (h, e.key, e.val, null, null);  
 //原位置  
 if ((h & n) == 0) {  
 if ((p.prev = loTail) == null)  
 lo = p;  
 else  
 loTail.next = p;  
 loTail = p;  
 ++lc;  
 }  
 //i+n位置  
 else {  
 if ((p.prev = hiTail) == null)  
 hi = p;  
 else  
 hiTail.next = p;  
 hiTail = p;  
 ++hc;  
 }  
 }  
 //如果扩容后已经不再需要tree的结构 反向转换为链表结构  
 ln = (lc <= *UNTREEIFY\_THRESHOLD*) ? *untreeify*(lo) :  
 (hc != 0) ? new TreeBin<K,V>(lo) : t;  
 hn = (hc <= *UNTREEIFY\_THRESHOLD*) ? *untreeify*(hi) :  
 (lc != 0) ? new TreeBin<K,V>(hi) : t;  
 *setTabAt*(nextTab, i, ln);  
 *setTabAt*(nextTab, i + n, hn);  
 *setTabAt*(tab, i, fwd);  
 advance = true;  
 }  
 }  
 }  
 }  
 }  
}

说明:

* 1. 线程是靠sizeCtl的值来感知其他线程是否也在进行扩容操作, 它初始值是一个负值=(rs << RESIZE\_STAMP\_SHIFT) + 2)，每当一个线程参与进来执行迁移工作，则该值进行CAS自增，该线程的任务执行完毕要退出时对该值进行CAS自减操作，所以当sizeCtl的值等于上述初值则说明了此时没有其他线程还在执行迁移工作，可以去执行收尾工作了
  2. 任务分片: stride的计算依赖于CPU的核数，如果只有1个核，那么此时就不用分片，即stride=n,其他情况就是 (n >>> 3) / NCPU
  3. ConcurrentHashMap含有一个属性transferIndex（初值为最后一个桶），表示从transferIndex开始到后面所有的桶的迁移任务已经被分配出去了。所以每次线程领取扩容任务，则需要对该属性进行CAS的减操作，即一般是transferIndex-stride。
  4. 线程处理分片任务时,第一个获取到分片的线程会创建一个新的Node<K,V>[],即nextTab,长度为原来的2倍,容量(sizeCtl)为原来的1.5倍(还是0.75\*n).然后遍历自己所分到的任务:



* + - 桶中元素不存在,则通过cas操作设置桶中的第一个元素为ForwardingNode,其hash值为MOVED(-1),同时该元素含有新的数组引用
    - 此时若其它线程进行put操作,发现第一个元素的hash值为-1即正在进行扩容操作（并且表明该桶已经完成扩容操作了，可以直接在新的数组中重新进行hash和插入操作），该线程就可以去参与进去，或者没有任务则不用参与，此时可以去直接操作新的数组了
    - 桶中为链表或者红黑树结构，则需要获取桶锁，防止其他线程对该桶进行put操作，然后处理方式同HashMap的处理方式一样，对桶中元素分为2类，分别代表当前桶中和要迁移到新桶中的元素。设置完毕后代表桶迁移工作已经完成，旧数组中该桶可以设置成ForwardingNode了

###### get(key)

public V get(Object key) {  
 Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;  
 //计算hash值  
 int h = *spread*(key.hashCode());  
 //通过tabAt找到元素位置  
 if ((tab = table) != null && (n = tab.length) > 0 &&  
 (e = *tabAt*(tab, (n - 1) & h)) != null) {  
 if ((eh = e.hash) == h) {  
 if ((ek = e.key) == key || (ek != null && key.equals(ek)))  
 return e.val;  
 }  
 //小于0可能为红黑树或者fwd节点  
 else if (eh < 0)  
 return (p = e.find(h, key)) != null ? p.val : null;  
 while ((e = e.next) != null) {  
 if (e.hash == h &&  
 ((ek = e.key) == key || (ek != null && key.equals(ek))))  
 return e.val;  
 }  
 }  
 return null;  
}

**说明**: 具体看源码

###### size()

*/\*\*  
 \* Base counter value, used mainly when there is no contention,  
 \* but also as a fallback during table initialization  
 \* races. Updated via CAS.  
 \*/*private transient volatile long baseCount;

*/\*\*  
 \* Table of counter cells. When non-null, size is a power of 2.  
 \* 保存部分元素的变化个数  
 \* size = baseCount + CounterCell[i].value  
 \*/*private transient volatile CounterCell[] counterCells;

public long mappingCount() {  
 long n = sumCount();  
 return (n < 0L) ? 0L : n; // ignore transient negative values  
}

public int size() {  
 long n = sumCount();  
 return ((n < 0L) ? 0 :  
 (n > (long)Integer.*MAX\_VALUE*) ? Integer.*MAX\_VALUE* :  
 (int)n);  
}

final long sumCount() {  
 CounterCell[] as = counterCells; CounterCell a;  
 long sum = baseCount;  
 if (as != null) {  
 for (int i = 0; i < as.length; ++i) {  
 if ((a = as[i]) != null)  
 sum += a.value;  
 }  
 }  
 return sum;  
}

**说明**: 对于ConcurrentHashMap来说，这个table里到底装了多少东西其实是个不确定的数量，因为不可能在调用size()方法的时候像GC的“stop the world”一样让其他线程都停下来让你去统计，因此只能说这个数量是个估计值。对于这个估计值，ConcurrentHashMap也是大费周章才计算出来的。提供了两个计算size的方法,mappingCount和size,都是调用的sumCount计算,可以看出在sumCount中,总数的计算是通过baseCount+counterCells[i].value 计算出来的.

#### ConcurrentSkipListMap

##### 概述

ConcurrentSkipListMap是线程安全的有序的哈希表，适用于高并发的场景。  
ConcurrentSkipListMap和TreeMap，它们虽然都是有序的哈希表。但是，第一，它们的线程安全机制不同，TreeMap是非线程安全的，而ConcurrentSkipListMap是线程安全的。第二，ConcurrentSkipListMap是通过跳表实现的，而TreeMap是通过红黑树实现的。  
关于跳表(Skip List)，它是平衡树的一种替代的数据结构，但是和红黑树不相同的是，跳表对于树的平衡的实现是基于一种随机化的算法的，这样也就是说跳表的插入和删除的工作是比较简单的。

##### 函数列表

// 构造一个新的空映射，该映射按照键的自然顺序进行排序。  
ConcurrentSkipListMap()  
// 构造一个新的空映射，该映射按照指定的比较器进行排序。  
ConcurrentSkipListMap(Comparator<? super K> comparator)  
// 构造一个新映射，该映射所包含的映射关系与给定映射包含的映射关系相同，并按照键的自然顺序进行排序。  
ConcurrentSkipListMap(Map<? extends K,? extends V> m)  
// 构造一个新映射，该映射所包含的映射关系与指定的有序映射包含的映射关系相同，使用的顺序也相同。  
ConcurrentSkipListMap(SortedMap<K,? extends V> m)  
  
// 返回与大于等于给定键的最小键关联的键-值映射关系；如果不存在这样的条目，则返回 null。  
Map.Entry<K,V> ceilingEntry(K key)  
// 返回大于等于给定键的最小键；如果不存在这样的键，则返回 null。  
K ceilingKey(K key)  
// 从此映射中移除所有映射关系。  
void clear()  
// 返回此 ConcurrentSkipListMap 实例的浅表副本。  
ConcurrentSkipListMap<K,V> clone()  
// 返回对此映射中的键进行排序的比较器；如果此映射使用键的自然顺序，则返回 null。  
Comparator<? super K> comparator()  
// 如果此映射包含指定键的映射关系，则返回 true。  
boolean containsKey(Object key)  
// 如果此映射为指定值映射一个或多个键，则返回 true。  
boolean containsValue(Object value)  
// 返回此映射中所包含键的逆序 NavigableSet 视图。  
NavigableSet<K> descendingKeySet()  
// 返回此映射中所包含映射关系的逆序视图。  
ConcurrentNavigableMap<K,V> descendingMap()  
// 返回此映射中所包含的映射关系的 Set 视图。  
Set<Map.Entry<K,V>> entrySet()  
// 比较指定对象与此映射的相等性。  
boolean equals(Object o)  
// 返回与此映射中的最小键关联的键-值映射关系；如果该映射为空，则返回 null。  
Map.Entry<K,V> firstEntry()  
// 返回此映射中当前第一个（最低）键。  
K firstKey()  
// 返回与小于等于给定键的最大键关联的键-值映射关系；如果不存在这样的键，则返回 null。  
Map.Entry<K,V> floorEntry(K key)  
// 返回小于等于给定键的最大键；如果不存在这样的键，则返回 null。  
K floorKey(K key)  
// 返回指定键所映射到的值；如果此映射不包含该键的映射关系，则返回 null。  
V get(Object key)  
// 返回此映射的部分视图，其键值严格小于 toKey。  
ConcurrentNavigableMap<K,V> headMap(K toKey)  
// 返回此映射的部分视图，其键小于（或等于，如果 inclusive 为 true）toKey。  
ConcurrentNavigableMap<K,V> headMap(K toKey, boolean inclusive)  
// 返回与严格大于给定键的最小键关联的键-值映射关系；如果不存在这样的键，则返回 null。  
Map.Entry<K,V> higherEntry(K key)  
// 返回严格大于给定键的最小键；如果不存在这样的键，则返回 null。  
K higherKey(K key)  
// 如果此映射未包含键-值映射关系，则返回 true。  
boolean isEmpty()  
// 返回此映射中所包含键的 NavigableSet 视图。  
NavigableSet<K> keySet()  
// 返回与此映射中的最大键关联的键-值映射关系；如果该映射为空，则返回 null。  
Map.Entry<K,V> lastEntry()  
// 返回映射中当前最后一个（最高）键。  
K lastKey()  
// 返回与严格小于给定键的最大键关联的键-值映射关系；如果不存在这样的键，则返回 null。  
Map.Entry<K,V> lowerEntry(K key)  
// 返回严格小于给定键的最大键；如果不存在这样的键，则返回 null。  
K lowerKey(K key)  
// 返回此映射中所包含键的 NavigableSet 视图。  
NavigableSet<K> navigableKeySet()  
// 移除并返回与此映射中的最小键关联的键-值映射关系；如果该映射为空，则返回 null。  
Map.Entry<K,V> pollFirstEntry()  
// 移除并返回与此映射中的最大键关联的键-值映射关系；如果该映射为空，则返回 null。  
Map.Entry<K,V> pollLastEntry()  
// 将指定值与此映射中的指定键关联。  
V put(K key, V value)  
// 如果指定键已经不再与某个值相关联，则将它与给定值关联。  
V putIfAbsent(K key, V value)  
// 从此映射中移除指定键的映射关系（如果存在）。  
V remove(Object key)  
// 只有目前将键的条目映射到给定值时，才移除该键的条目。  
boolean remove(Object key, Object value)  
// 只有目前将键的条目映射到某一值时，才替换该键的条目。  
V replace(K key, V value)  
// 只有目前将键的条目映射到给定值时，才替换该键的条目。  
boolean replace(K key, V oldValue, V newValue)  
// 返回此映射中的键-值映射关系数。  
int size()  
// 返回此映射的部分视图，其键的范围从 fromKey 到 toKey。  
ConcurrentNavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)  
// 返回此映射的部分视图，其键值的范围从 fromKey（包括）到 toKey（不包括）。  
ConcurrentNavigableMap<K,V> subMap(K fromKey, K toKey)  
// 返回此映射的部分视图，其键大于等于 fromKey。  
ConcurrentNavigableMap<K,V> tailMap(K fromKey)  
// 返回此映射的部分视图，其键大于（或等于，如果 inclusive 为 true）fromKey。  
ConcurrentNavigableMap<K,V> tailMap(K fromKey, boolean inclusive)  
// 返回此映射中所包含值的 Collection 视图。  
Collection<V> values();

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*以下为1.8新增方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
  
/\*\*对于指定key做remappingFunction函数调用，remappingFunction函数返回值即为新的value，  
如果返回值为null，则从map中删除对应的key。compute返回key更新后的值（remappingFunction函数返回值）\*/*public V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction);  
  
*/\*\*如果指定的key不存在，对该key做mappingFunction函数操作，mappingFunction函数返回值不为null，则将对应的k-v放到map中，否则不操作。  
如果key存在返回key对应的value（此时mappingFunction不会调用）\*/*public V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction);  
*/\*\*类似与computeIfAbsent，仅对已经存在的key才计算新value。同样，如果remappingFunction返回值为null，会删除对应的k-v。\*/*public V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction);  
*/\*\*当key不存在，直接插入对应value，remappingFunction不会被调用；否则，对oldValue与value做remappingFunction函数，结果作为新的newValue插入到map中。  
 同样null结果会删除对应的k-v。\*/*public V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction);

##### 数据结构







**说明**：

先以数据“7,14,21,32,37,71,85”序列为例，来对跳表进行简单说明。

跳表分为许多层(level)，每一层都可以看作是数据的索引，这些索引的意义就是加快跳表查找数据速度。每一层的数据都是有序的，上一层数据是下一层数据的子集，并且第一层(level 1)包含了全部的数据；层次越高，跳跃性越大，包含的数据越少。  
跳表包含一个表头，它查找数据时，是从上往下，从左往右进行查找。现在“需要找出值为32的节点”为例，来对比说明跳表和普遍的链表。

**情况1：链表中查找“32”节点**  
路径如下图1-02所示：

[http://images.cnitblog.com/blog/497634/201312/30222059-f01604b1b24d429e940f7a002947c143.jpg](http://images.cnitblog.com/blog/497634/201312/30222059-f01604b1b24d429e940f7a002947c143.jpg)

需要4步(红色部分表示路径)。

**情况2：跳表中查找“32”节点**  
路径如下图1-03所示：

[](http://images.cnitblog.com/blog/497634/201312/30222128-045c88b7e992443395a540ba2eb740f3.jpg)

忽略索引垂直线路上路径的情况下，只需要2步(红色部分表示路径)。

下面说说Java中ConcurrentSkipListMap的数据结构。  
(01) ConcurrentSkipListMap继承于AbstractMap类，也就意味着它是一个哈希表。  
(02) Index是ConcurrentSkipListMap的内部类，它与“跳表中的索引相对应”。HeadIndex继承于Index，ConcurrentSkipListMap中含有一个HeadIndex的对象head，head是“跳表的表头”。  
(03) Index是跳表中的索引，它包含“右索引的指针(right)”，“下索引的指针(down)”和“哈希表节点node”。node是Node的对象，Node也是ConcurrentSkipListMap中的内部类。

###### Node<K,V>

static final class Node<K,V> {  
 final K key;  
 volatile Object value;  
 volatile Node<K,V> next;  
  
 */\*\*  
 \* Creates a new regular node.  
 \*/* Node(K key, Object value, Node<K,V> next) {  
 this.key = key;  
 this.value = value;  
 this.next = next;  
 }  
  
 */\*\*  
 \* Creates a new marker node. A marker is distinguished by  
 \* having its value field point to itself. Marker nodes also  
 \* have null keys, a fact that is exploited in a few places,  
 \* but this doesn't distinguish markers from the base-level  
 \* header node (head.node), which also has a null key.  
 \*/* Node(Node<K,V> next) {  
 this.key = null;  
 this.value = this;  
 this.next = next;  
 }

//以下代码省略

}

**说明**: 表示节点数据,存储K-V,持有下一个节点的引用

###### Index<K,V>

static class Index<K,V> {  
 final Node<K, V> node;  
 final Index<K, V> down;  
 volatile Index<K, V> right;  
  
 */\*\*  
 \* Creates index node with given values.  
 \*/* Index(Node<K, V> node, Index<K, V> down, Index<K, V> right) {  
 this.node = node;  
 this.down = down;  
 this.right = right;  
 }  
 //以下代码省略  
}

**说明:** 跳表中的索引,包含了右指针(right),向下的索引down和node节点的引用

###### HeadIndex<K,V>

*/\*\*  
 \* Nodes heading each level keep track of their level.  
 \* 表示跳表的表头,内部标识了跳表的层级level  
 \*/*static final class HeadIndex<K,V> extends Index<K,V> {  
 final int level;  
 HeadIndex(Node<K,V> node, Index<K,V> down, Index<K,V> right, int level) {  
 super(node, down, right);  
 this.level = level;  
 }  
}

说明: 跳表的表头header,内部标识了当前的层级

##### 源码解析

###### 构造函数

public ConcurrentSkipListMap() {  
 this.comparator = null;  
 initialize();  
}  
  
*/\*\*  
 \* Constructs a new, empty map, sorted according to the specified  
 \* comparator.  
 \*  
 \** ***@param*** *comparator the comparator that will be used to order this map.  
 \* If {****@code*** *null}, the {****@linkplain*** *Comparable natural  
 \* ordering} of the keys will be used.  
 \*/*public ConcurrentSkipListMap(Comparator<? super K> comparator) {  
 this.comparator = comparator;  
 initialize();  
}  
  
*/\*\*  
 \* Constructs a new map containing the same mappings as the given map,  
 \* sorted according to the {****@linkplain*** *Comparable natural ordering} of  
 \* the keys.  
 \*  
 \** ***@param*** *m the map whose mappings are to be placed in this map  
 \** ***@throws*** *ClassCastException if the keys in {****@code*** *m} are not  
 \* {****@link*** *Comparable}, or are not mutually comparable  
 \** ***@throws*** *NullPointerException if the specified map or any of its keys  
 \* or values are null  
 \*/*public ConcurrentSkipListMap(Map<? extends K, ? extends V> m) {  
 this.comparator = null;  
 initialize();  
 putAll(m);  
}  
  
*/\*\*  
 \* Constructs a new map containing the same mappings and using the  
 \* same ordering as the specified sorted map.  
 \*  
 \** ***@param*** *m the sorted map whose mappings are to be placed in this  
 \* map, and whose comparator is to be used to sort this map  
 \** ***@throws*** *NullPointerException if the specified sorted map or any of  
 \* its keys or values are null  
 \*/*public ConcurrentSkipListMap(SortedMap<K, ? extends V> m) {  
 this.comparator = m.comparator();  
 initialize();  
 buildFromSorted(m);  
}  
*/\*\*  
 \* Initializes or resets state. Needed by constructors, clone,  
 \* clear, readObject. and ConcurrentSkipListSet.clone.  
 \* (Note that comparator must be separately initialized.)  
 \*/*private void initialize() {  
 keySet = null;  
 entrySet = null;  
 values = null;  
 descendingMap = null;  
 head = new HeadIndex<K,V>(new Node<K,V>(null, *BASE\_HEADER*, null),  
 null, null, 1);  
}

**说明:** 可以看到构造函数都是调用了initialize()方法,该方法在构造器、clone、clear、readObject和ConcurrentSkipListSet.clone中都有用到,注意在构造函数中,comparator需单独初始化。

###### put(K,V)

public V put(K key, V value) {  
 if (value == null)  
 throw new NullPointerException();  
 return doPut(key, value, false);  
}  
private V doPut(K key, V value, boolean onlyIfAbsent) {  
 Node<K,V> z; // added node  
 if (key == null)  
 throw new NullPointerException();  
 Comparator<? super K> cmp = comparator;  
 //自旋  
 outer: for (;;) {  
 //找到最底层(base-level)节点链中key的上一个节点(比key小(左边))的节点b,n为当前节点  
 for (Node<K,V> b = findPredecessor(key, cmp), n = b.next;;) {  
 if (n != null) {  
 Object v; int c;  
 Node<K,V> f = n.next;  
 if (n != b.next) //读不一致,跳出重试 // inconsistent read  
 break;  
 if ((v = n.value) == null) { // n is deleted  
 //帮助删除n节点  
 n.helpDelete(b, f);  
 break;  
 }  
 if (b.value == null || v == n) // b is deleted  
 break;  
 if ((c = *cpr*(cmp, key, n.key)) > 0) {//当前key大于n.key,把之前节点往后移动  
 b = n;  
 n = f;  
 continue;  
 }  
 if (c == 0) {//key相同处理  
 if (onlyIfAbsent || n.casValue(v, value)) {  
 @SuppressWarnings("unchecked") V vv = (V)v;  
 return vv;  
 }  
 break; // restart if lost race to replace value  
 }  
 // else c < 0; fall through  
 }  
 //新建一个节点  
 z = new Node<K,V>(key, value, n);  
 //cas替换  
 if (!b.casNext(n, z))  
 break; // restart if lost race to append to b  
 break outer;  
 }  
 }  
 //在跳表上层添加index  
 int rnd = ThreadLocalRandom.*nextSecondarySeed*();  
 if ((rnd & 0x80000001) == 0) { //生成的32位的rnd最高位和最低位不为1时执行(极大概率) test highest and lowest bits  
 int level = 1, max;  
 //计算跳表level  
 while (((rnd >>>= 1) & 1) != 0)//判断从右到左有多少个连续的1  
 ++level;  
 Index<K,V> idx = null;  
 HeadIndex<K,V> h = head;  
 if (level <= (max = h.level)) {  
 for (int i = 1; i <= level; ++i)  
 //从下到上依次赋值,节点持有新节点z和down节点idx的引用  
 idx = new Index<K,V>(z, idx, null);  
 }  
 else { // try to grow by one level  
 level = max + 1; // hold in array and later pick the one to use  
 //构建一个level+1长度的Index数组  
 @SuppressWarnings("unchecked")Index<K,V>[] idxs =  
 (Index<K,V>[])new Index<?,?>[level+1];  
 //从下到到上构建Index  
 for (int i = 1; i <= level; ++i)  
 idxs[i] = idx = new Index<K,V>(z, idx, null);  
 //自旋  
 for (;;) {  
 h = head;  
 //保存head之前的层级  
 int oldLevel = h.level;  
 if (level <= oldLevel) // lost race to add level  
 break;  
  
 HeadIndex<K,V> newh = h;  
 Node<K,V> oldbase = h.node;  
 //为每层生成一个头节点,每层的右节点为idxs[j]  
 for (int j = oldLevel+1; j <= level; ++j)  
 newh = new HeadIndex<K,V>(oldbase, newh, idxs[j], j);  
 //cas替换头节点  
 if (casHead(h, newh)) {  
 h = newh;  
 //idx赋值为之前层级的头结点，并将level赋值为之前的层级  
 idx = idxs[level = oldLevel];  
 break;  
 }  
 }  
 }  
 // find insertion points and splice in  
 //为每一层插入数据  
 splice: for (int insertionLevel = level;;) {  
 //获取新跳表headIndex的层级  
 int j = h.level;  
 //q:新跳表; r:q.right; t:未操作之前跳表  
 for (Index<K,V> q = h, r = q.right, t = idx;;) {  
 if (q == null || t == null)  
 break splice;  
 if (r != null) {  
 Node<K,V> n = r.node;  
 // compare before deletion check avoids needing recheck  
 int c = *cpr*(cmp, key, n.key);  
 if (n.value == null) {// 结点的值为空，表示需要删除  
 if (!q.unlink(r))// 删除q的right结点  
 break;  
 r = q.right;  
 continue;  
 }  
 if (c > 0) {// key大于结点的key,向右寻找  
 q = r;  
 r = r.right;  
 continue;  
 }  
 }  
  
 if (j == insertionLevel) {  
 if (!q.link(r, t))//r结点插入q与t之间  
 break; // restart  
 if (t.node.value == null) {  
 findNode(key);  
 break splice;  
 }  
 if (--insertionLevel == 0)//到达最底层  
 break splice;  
 }  
  
 if (--j >= insertionLevel && j < level)//移动到下一层level  
 t = t.down;  
 q = q.down;  
 r = q.right;  
 }  
 }  
 }  
 return null;

说明:put调用了doput方法,主要分两个个步骤:

1. 找到最底层(base-level)需要插入的节点位置,在最底层插入node,此处用findPredecessor方法找到最底层比给定key小(在给定节点左边)的节点,源码如下

*/\*\*  
 \* Returns a base-level node with key strictly less than given key,  
 \* or the base-level header if there is no such node. Also  
 \* unlinks indexes to deleted nodes found along the way. Callers  
 \* rely on this side-effect of clearing indices to deleted nodes.  
 \** ***@param*** *key the key  
 \** ***@return*** *a predecessor of key  
 \* 返回最底层(base-level)节点链中比给定key小(在“给定节点”左边)的节点，  
 \* 如果没找到，那么返回底层链的头节点。  
 \* 在查找过程中会帮助删除一些标记为删除的节点。  
 \*/*private Node<K,V> findPredecessor(Object key, Comparator<? super K> cmp) {  
 if (key == null)  
 throw new NullPointerException(); // don't postpone errors  
 for (;;) {  
 //q:head节点; r:右节点; d:下节点  
 for (Index<K,V> q = head, r = q.right, d;;) {  
 //右节点不为空  
 if (r != null) {  
 Node<K,V> n = r.node;  
 K k = n.key;  
 //判断r.node是否被删除  
 if (n.value == null) {  
 //如果n已经删除,尝试以CAS更新q的右节点为r.right  
 if (!q.unlink(r))  
 break; // restart  
 r = q.right; // reread r  
 continue;  
 }  
 //比较key和k  
 if (*cpr*(cmp, key, k) > 0) {  
 //继续向右循环  
 q = r;  
 r = r.right;  
 continue;  
 }  
 }  
 //right节点为空,则向下找  
 if ((d = q.down) == null)  
 //下节点为空,返回当前q节点的node  
 return q.node;  
 q = d;  
 r = d.right;  
 }  
 }  
}

1. 在跳表上层添加index
   1. 生成随机层级level,level小于最大层级直接从下到上构建index;大于最大层级时首先构建head(最上层headIndex)节点
   2. 通过双层嵌套循环为每一层添加index,这个操作中如果最上层Index(idx)的value为空,证明节点可能被删除了,就调用findNode()方法消除删除的节点,源码如下:

private Node<K,V> findNode(Object key) {  
 if (key == null)  
 throw new NullPointerException(); // don't postpone errors  
 Comparator<? super K> cmp = comparator;  
 outer: for (;;) {  
 for (Node<K,V> b = findPredecessor(key, cmp), n = b.next;;) {  
 Object v; int c;  
 if (n == null)  
 break outer;  
 Node<K,V> f = n.next;  
 if (n != b.next) // inconsistent read  
 break;  
 if ((v = n.value) == null) { // n is deleted  
 n.helpDelete(b, f);  
 break;  
 }  
 if (b.value == null || v == n) // b is deleted  
 break;  
 if ((c = *cpr*(cmp, key, n.key)) == 0)  
 return n;  
 if (c < 0)  
 break outer;  
 b = n;  
 n = f;  
 }  
 }  
 return null;  
}

###### remove(K)

public V remove(Object key) {  
 return doRemove(key, null);  
}  
final V doRemove(Object key, Object value) {  
 if (key == null)  
 throw new NullPointerException();  
 Comparator<? super K> cmp = comparator;  
 outer: for (;;) {  
 //b:比key对应的node小(左侧)的节点  
 for (Node<K,V> b = findPredecessor(key, cmp), n = b.next;;) {  
 Object v; int c;  
 if (n == null)  
 break outer;  
 Node<K,V> f = n.next;  
 if (n != b.next) // inconsistent read  
 break;  
 if ((v = n.value) == null) { // n is deleted  
 n.helpDelete(b, f);  
 break;  
 }  
 if (b.value == null || v == n) // b is deleted  
 break;  
 if ((c = *cpr*(cmp, key, n.key)) < 0)  
 break outer;  
 if (c > 0) {  
 //继续往右寻找  
 b = n;  
 n = f;  
 continue;  
 }  
 if (value != null && !value.equals(v))  
 break outer;  
 if (!n.casValue(v, null))//当前节点的value设为空  
 break;  
 //n替换为f,b的next替换为f  
 if (!n.appendMarker(f) || !b.casNext(n, f))  
 findNode(key); // retry via findNode  
 else {  
 //删除n节点对应的index  
 findPredecessor(key, cmp); // clean index  
 if (head.right == null)  
 //减少跳表层级  
 tryReduceLevel();  
 }  
 @SuppressWarnings("unchecked") V vv = (V)v;  
 return vv;  
 }  
 }  
 return null;  
}

**说明:** 从源码中可以看出,remove方法调用了doRemove(). 首先找到需要删除节点的前节点,如果中途发现有已经删除的节点,就帮助清除,然后删除base-level的node,再用findPredecessor()的辅助作用删除node节点对应的index,最后判断head是否还有right节点,没有的话就需要通过tryReduceLevel()减少跳表层级,tryReduceLevel()源码如下:

private void tryReduceLevel() {  
 HeadIndex<K,V> h = head;  
 HeadIndex<K,V> d;  
 HeadIndex<K,V> e;  
 if (h.level > 3 &&  
 (d = (HeadIndex<K,V>)h.down) != null &&  
 (e = (HeadIndex<K,V>)d.down) != null &&  
 e.right == null &&  
 d.right == null &&  
 h.right == null &&  
 casHead(h, d) && // try to set  
 h.right != null) // recheck  
 casHead(d, h); // try to backout  
}

###### get(K)

public V get(Object key) {  
 return doGet(key);  
}  
private V doGet(Object key) {  
 if (key == null)  
 throw new NullPointerException();  
 Comparator<? super K> cmp = comparator;  
 outer: for (;;) {  
 //找到最底层比key小(左边)的node  
 for (Node<K,V> b = findPredecessor(key, cmp), n = b.next;;) {  
 Object v; int c;  
 if (n == null)  
 break outer;  
 Node<K,V> f = n.next;  
 if (n != b.next) // inconsistent read  
 break;  
 if ((v = n.value) == null) { // n is deleted  
 n.helpDelete(b, f);  
 break;  
 }  
 if (b.value == null || v == n) // b is deleted  
 break;  
 if ((c = *cpr*(cmp, key, n.key)) == 0) {  
 @SuppressWarnings("unchecked") V vv = (V)v;  
 return vv;  
 }  
 if (c < 0)  
 break outer;  
 b = n;  
 n = f;  
 }  
 }  
 return null;  
}

**说明:** 很简单,看源码

###### size()

public int size() {  
 long count = 0;  
 for (Node<K,V> n = findFirst(); n != null; n = n.next) {  
 if (n.getValidValue() != null)  
 ++count;  
 }  
 return (count >= Integer.*MAX\_VALUE*) ? Integer.*MAX\_VALUE* : (int) count;  
}

**说明:** 找到首节点,依次向后遍历.用到了findFirst(),源码如下:

final Node<K,V> findFirst() {  
 for (Node<K,V> b, n;;) {  
 if ((n = (b = head.node).next) == null)  
 return null;  
 if (n.value != null)  
 return n;  
 n.helpDelete(b, n.next);  
 }  
}

#### ConcurrentSkipListSet

##### 概述

ConcurrentSkipListSet是线程安全的有序的集合，适用于高并发的场景。  
ConcurrentSkipListSet和TreeSet，它们虽然都是有序的集合。但是，

第一，它们的线程安全机制不同，TreeSet是非线程安全的，而ConcurrentSkipListSet是线程安全的。

第二，ConcurrentSkipListSet是通过ConcurrentSkipListMap实现的，而TreeSet是通过TreeMap实现的。

##### 函数列表

// 构造一个新的空 set，该 set 按照元素的自然顺序对其进行排序。  
ConcurrentSkipListSet()  
// 构造一个包含指定 collection 中元素的新 set，这个新 set 按照元素的自然顺序对其进行排序。  
ConcurrentSkipListSet(Collection<? extends E> c)  
// 构造一个新的空 set，该 set 按照指定的比较器对其元素进行排序。  
ConcurrentSkipListSet(Comparator<? super E> comparator)  
// 构造一个新 set，该 set 所包含的元素与指定的有序 set 包含的元素相同，使用的顺序也相同。  
ConcurrentSkipListSet(SortedSet<E> s)  
  
// 如果此 set 中不包含指定元素，则添加指定元素。  
boolean add(E e)  
// 返回此 set 中大于等于给定元素的最小元素；如果不存在这样的元素，则返回 null。  
E ceiling(E e)  
// 从此 set 中移除所有元素。  
void clear()  
// 返回此 ConcurrentSkipListSet 实例的浅表副本。  
ConcurrentSkipListSet<E> clone()  
// 返回对此 set 中的元素进行排序的比较器；如果此 set 使用其元素的自然顺序，则返回 null。  
Comparator<? super E> comparator()  
// 如果此 set 包含指定的元素，则返回 true。  
boolean contains(Object o)  
// 返回在此 set 的元素上以降序进行迭代的迭代器。  
Iterator<E> descendingIterator()  
// 返回此 set 中所包含元素的逆序视图。  
NavigableSet<E> descendingSet()  
// 比较指定对象与此 set 的相等性。  
boolean equals(Object o)  
// 返回此 set 中当前第一个（最低）元素。  
E first()  
// 返回此 set 中小于等于给定元素的最大元素；如果不存在这样的元素，则返回 null。  
E floor(E e)  
// 返回此 set 的部分视图，其元素严格小于 toElement。  
NavigableSet<E> headSet(E toElement)  
// 返回此 set 的部分视图，其元素小于（或等于，如果 inclusive 为 true）toElement。  
NavigableSet<E> headSet(E toElement, boolean inclusive)  
// 返回此 set 中严格大于给定元素的最小元素；如果不存在这样的元素，则返回 null。  
E higher(E e)  
// 如果此 set 不包含任何元素，则返回 true。  
boolean isEmpty()  
// 返回在此 set 的元素上以升序进行迭代的迭代器。  
Iterator<E> iterator()  
// 返回此 set 中当前最后一个（最高）元素。  
E last()  
// 返回此 set 中严格小于给定元素的最大元素；如果不存在这样的元素，则返回 null。  
E lower(E e)  
// 获取并移除第一个（最低）元素；如果此 set 为空，则返回 null。  
E pollFirst()  
// 获取并移除最后一个（最高）元素；如果此 set 为空，则返回 null。  
E pollLast()  
// 如果此 set 中存在指定的元素，则将其移除。  
boolean remove(Object o)  
// 从此 set 中移除包含在指定 collection 中的所有元素。  
boolean removeAll(Collection<?> c)  
// 返回此 set 中的元素数目。  
int size()  
// 返回此 set 的部分视图，其元素范围从 fromElement 到 toElement。  
NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)  
// 返回此 set 的部分视图，其元素从 fromElement（包括）到 toElement（不包括）。  
NavigableSet<E> subSet(E fromElement, E toElement)  
// 返回此 set 的部分视图，其元素大于等于 fromElement。  
NavigableSet<E> tailSet(E fromElement)  
// 返回此 set 的部分视图，其元素大于（或等于，如果 inclusive 为 true）fromElement。  
NavigableSet<E> tailSet(E fromElement, boolean inclusive);

##### 数据结构



**说明**：  
(01) ConcurrentSkipListSet继承于AbstractSet。因此，它本质上是一个集合。  
(02) ConcurrentSkipListSet实现了NavigableSet接口。因此，ConcurrentSkipListSet是一个有序的集合。  
(03) ConcurrentSkipListSet是通过ConcurrentSkipListMap实现的。它包含一个ConcurrentNavigableMap对象m，而m对象实际上是ConcurrentNavigableMap的实现类ConcurrentSkipListMap的实例。ConcurrentSkipListMap中的元素是key-value键值对；而ConcurrentSkipListSet是集合，它只用到了ConcurrentSkipListMap中的key

##### 源码解析

见ConcurrentSkipListMap源码

#### ArrayBlockingQueue

##### 概述

ArrayBlockingQueue是数组实现的线程安全的有界的阻塞队列。  
线程安全是指，ArrayBlockingQueue内部通过“互斥锁”保护竞争资源，实现了多线程对竞争资源的互斥访问。而有界，则是指ArrayBlockingQueue对应的数组是有界限的。 阻塞队列，是指多线程访问竞争资源时，当竞争资源已被某线程获取时，其它要获取该资源的线程需要阻塞等待；而且，ArrayBlockingQueue是按 FIFO（先进先出）原则对元素进行排序，元素都是从尾部插入到队列，从头部开始返回。

注意：ArrayBlockingQueue不同于ConcurrentLinkedQueue，ArrayBlockingQueue是数组实现的，并且是有界限的；而ConcurrentLinkedQueue是链表实现的，是无界限的。

##### 函数列表

public ArrayBlockingQueue(int capacity);  
//创建ArrayBlockingQueue,指定锁类型  
public ArrayBlockingQueue(int capacity, boolean fair);  
//指定Collection元素  
public ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends E> c);  
//添加元素到尾部,队列已满抛出IllegalStateException  
public boolean add(E e);  
//添加元素到尾部,队列已满返回false  
public boolean offer(E e);  
//添加元素到尾部,队列已满等待timeout后无空间返回false  
public boolean offer(E e, long timeout, TimeUnit unit);  
//添加元素到尾部,队列已满就等待队列可用  
public void put(E e);  
//获取并移除队列头数据  
public E poll();  
//获取并移除队列头数据,等待指定时间  
public E poll(long timeout, TimeUnit unit);  
//获取并移除队列头数据,无数据一直等待  
public E take();  
//获取不移除队列都数据  
public E peek();  
//返回队列元素数量  
public int size();  
//返回剩余空间  
public int remainingCapacity();  
//移除元素  
public boolean remove(Object o);  
//返回是否包含o元素  
public boolean contains(Object o);  
//返回队列的元素数组  
public Object[] toArray();  
//返回队列的元素数组,指定类型  
public <T> T[] toArray(T[] a);  
//清空队列  
public void clear();  
//移除队列所有可用元素,并添加到c中  
public int drainTo(Collection<? super E> c);  
//移除队列中指定数量的元素,并添加到c中  
public int drainTo(Collection<? super E> c, int maxElements);

##### 数据结构



**说明:**1. 内部存储通过一个Object数组items实现2.线程安全是通过ReentrantLock和Condition实现的,内部持有ReentrantLock和两个Condition( notEmpty和notFull )的引用,分别控制队列的插入和取出,构造时也可以指定锁性质(fair或者nofair),源码如下:

//存储元素的数组  
final Object[] items;  
  
*/\*\* items index for next take, poll, peek or remove \*/*//下一个要取的元素索引  
int takeIndex;  
  
*/\*\* items index for next put, offer, or add \*/*//下一个要插入的元素索引  
int putIndex;  
  
*/\*\* Number of elements in the queue \*/*//队列中的项  
int count;  
  
/\*  
 \* Concurrency control uses the classic two-condition algorithm  
 \* found in any textbook.  
 \*/  
  
*/\*\* Main lock guarding all access \*/*final ReentrantLock lock;  
  
*/\*\* Condition for waiting takes \*/*//等待获取条件  
private final Condition notEmpty;  
  
*/\*\* Condition for waiting puts \*/*//等待插入条件  
private final Condition notFull;  
  
*/\*\*  
 \* Shared state for currently active iterators, or null if there  
 \* are known not to be any. Allows queue operations to update  
 \* iterator state.  
 \*/*//当前活动迭代器共享状态,没有的话为null.允许队列操作时更新迭代器状态  
transient Itrs itrs = null;

##### 源码解析

###### 构造函数

public ArrayBlockingQueue(int capacity) {  
 this(capacity, false);  
}  
public ArrayBlockingQueue(int capacity, boolean fair) {  
 if (capacity <= 0)  
 throw new IllegalArgumentException();  
 this.items = new Object[capacity];  
 lock = new ReentrantLock(fair);  
 notEmpty = lock.newCondition();  
 notFull = lock.newCondition();  
}  
public ArrayBlockingQueue(int capacity, boolean fair,  
 Collection<? extends E> c) {  
 this(capacity, fair);  
  
 final ReentrantLock lock = this.lock;  
 lock.lock(); // Lock only for visibility, not mutual exclusion  
 try {  
 int i = 0;  
 try {  
 for (E e : c) {  
 *checkNotNull*(e);  
 items[i++] = e;  
 }  
 } catch (ArrayIndexOutOfBoundsException ex) {  
 throw new IllegalArgumentException();  
 }  
 count = i;  
 putIndex = (i == capacity) ? 0 : i;  
 } finally {  
 lock.unlock();  
 }  
}

**说明:** 构造一个指定数组大小,指定锁类型的ArrayBlockingQueue,默认为非公平锁.

###### 添加(入列)

ArrayBlockingQueue的插入元素方法包括:add(E),offer(E),put(E),源码如下:

*/\*\*添加元素到尾部,队列已满抛出IllegalStateException\*/*public boolean add(E e) {  
 return super.add(e);  
}  
public boolean add(E e) {  
 if (offer(e))  
 return true;  
 else  
 throw new IllegalStateException("Queue full");  
}

*/\*\*添加元素到尾部,队列已满返回false\*/*public boolean offer(E e) {  
 *checkNotNull*(e);  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 if (count == items.length)  
 return false;  
 else {  
 enqueue(e);  
 return true;  
 }  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*插入元素到队列尾部\*/*private void enqueue(E x) {  
 // assert lock.getHoldCount() == 1;  
 // assert items[putIndex] == null;  
 final Object[] items = this.items;  
 items[putIndex] = x;  
 if (++putIndex == items.length)  
 putIndex = 0;  
 count++;  
 notEmpty.signal();  
}

*/\*\*添加元素到尾部,队列已满就一直等待可用\*/*public void put(E e) throws InterruptedException {  
 *checkNotNull*(e);  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 while (count == items.length)  
 notFull.await();  
 enqueue(e);  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*添加元素到队列尾部,队列已满等待timeout后无空间返回false\*/*public boolean offer(E e, long timeout, TimeUnit unit)  
 throws InterruptedException {  
  
 *checkNotNull*(e);  
 long nanos = unit.toNanos(timeout);  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 while (count == items.length) {  
 if (nanos <= 0)  
 return false;  
 nanos = notFull.awaitNanos(nanos);  
 }  
 enqueue(e);  
 return true;  
 } finally {  
 lock.unlock();  
 }  
}

**说明:** p

1. add内部调用了offer,不同的是add在队列已满的时候会抛出IllegalStateException.
2. Put插入元素时,如果队列已满,则会一直等待队列可用,并且lock是响应中断的.

###### 获取(出列)

获取元素的方法包括:poll(),take(),peek(),drainTo(Collection)源码如下:

*/\*\*获取并移除列表头元素\*/*public E poll() {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 return (count == 0) ? null : dequeue();  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*获取并移除列表头元素,无元素一直等待\*/*public E take() throws InterruptedException {  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 while (count == 0)  
 notEmpty.await();  
 return dequeue();  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*获取并移除列表头元素,无元素等待timeout后返回null\*/*public E poll(long timeout, TimeUnit unit) throws InterruptedException {  
 long nanos = unit.toNanos(timeout);  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 while (count == 0) {  
 if (nanos <= 0)  
 return null;  
 nanos = notEmpty.awaitNanos(nanos);  
 }  
 return dequeue();  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*获取但不移除头元素\*/*public E peek() {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 return itemAt(takeIndex); // null when queue is empty  
 } finally {  
 lock.unlock();  
 }  
}

**说明:**

1. poll()会获取并移除头部元素
2. take()获取并移除头部元素,在队列为空时一直等待队列可用
3. peek()获取元素,但不移除
4. 元素移除时都调用dequeue()方法,源码如下:

//出队列  
private E dequeue() {  
 // assert lock.getHoldCount() == 1;  
 // assert items[takeIndex] != null;  
 final Object[] items = this.items;  
 @SuppressWarnings("unchecked")  
 E x = (E) items[takeIndex];  
 items[takeIndex] = null;  
 if (++takeIndex == items.length)  
 takeIndex = 0;  
 count--;  
 if (itrs != null)  
 itrs.elementDequeued();  
 notFull.signal();//唤醒notFull  
 return x;  
}

###### remove(O)

*/\*\*移除o元素\*/*public boolean remove(Object o) {  
 if (o == null) return false;  
 final Object[] items = this.items;  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 if (count > 0) {  
 final int putIndex = this.putIndex;  
 int i = takeIndex;  
 do {  
 if (o.equals(items[i])) {  
 removeAt(i);  
 return true;  
 }  
 if (++i == items.length)  
 i = 0;  
 } while (i != putIndex);  
 }  
 return false;  
 } finally {  
 lock.unlock();  
 }  
}  
void removeAt(final int removeIndex) {  
 // assert lock.getHoldCount() == 1;  
 // assert items[removeIndex] != null;  
 // assert removeIndex >= 0 && removeIndex < items.length;  
 final Object[] items = this.items;  
 if (removeIndex == takeIndex) {//移除元素的索引为取的索引,直接进行出队列操作  
 // removing front item; just advance  
 items[takeIndex] = null;  
 if (++takeIndex == items.length)  
 takeIndex = 0;  
 count--;  
 if (itrs != null)  
 itrs.elementDequeued();  
 } else {  
 // an "interior" remove  
  
 // slide over all others up through putIndex.  
 final int putIndex = this.putIndex;  
 for (int i = removeIndex;;) {  
 int next = i + 1;  
 if (next == items.length)  
 next = 0;  
 if (next != putIndex) {//移除的元素不是队尾  
 items[i] = items[next];  
 i = next;  
 } else {  
 items[i] = null;  
 this.putIndex = i;  
 break;  
 }  
 }  
 count--;  
 if (itrs != null)  
 itrs.removedAt(removeIndex);  
 }  
 notFull.signal();  
}

**说明:** 找到移除对象的索引,通过removeAt(i)移除对象,移除之后释放等待notFull条件的线程

###### drainTo(Collection)

*/\*\*移除队列所有可用元素,并添加到Collection中\*/*public int drainTo(Collection<? super E> c) {  
 return drainTo(c, Integer.*MAX\_VALUE*);  
}  
*/\*\*移除队列指定数量的可用元素,并添加到Collection中\*/*public int drainTo(Collection<? super E> c, int maxElements) {  
 *checkNotNull*(c);  
 if (c == this)  
 throw new IllegalArgumentException();  
 if (maxElements <= 0)  
 return 0;  
 final Object[] items = this.items;  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 int n = Math.*min*(maxElements, count);  
 int take = takeIndex;  
 int i = 0;  
 try {  
 while (i < n) {  
 @SuppressWarnings("unchecked")  
 E x = (E) items[take];  
 c.add(x);  
 items[take] = null;  
 if (++take == items.length)  
 take = 0;  
 i++;  
 }  
 return n;  
 } finally {  
 // Restore invariants even if c.add() threw  
 //当collection添加元素出错时修复队列属性  
 if (i > 0) {  
 count -= i;  
 takeIndex = take;  
 if (itrs != null) {  
 if (count == 0)  
 itrs.queueIsEmpty();//队列置空  
 else if (i > take)  
 itrs.takeIndexWrapped();//出列索引置为0时调用,更新迭代器状态  
 }  
 for (; i > 0 && lock.hasWaiters(notFull); i--)  
 notFull.signal();  
 }  
 }  
 } finally {  
 lock.unlock();  
 }  
}

说明: 移除指定数量的可用元素,并放入指定collection中,操作完成之后循环唤醒等待notFull条件的线程.

#### LinkedBlockingQueue

##### 概述

LinkedBlockingQueue是一个单向链表实现的阻塞队列。该队列按 FIFO（先进先出）排序元素，新元素插入到队列的尾部，并且队列获取操作会获得位于队列头部的元素。链接队列的吞吐量通常要高于基于数组的队列，但是在大多数并发应用程序中，其可预知的性能要低。

此外，LinkedBlockingQueue还是可选容量的(防止过度膨胀)，即可以指定队列的容量。如果不指定，默认容量大小等于Integer.MAX\_VALUE。

##### 函数列表

public LinkedBlockingQueue() {  
 this(Integer.*MAX\_VALUE*);  
}  
public LinkedBlockingQueue(int capacity)

public LinkedBlockingQueue(Collection<? extends E> c)

//队列长度  
public int size()

//剩余空间  
public int remainingCapacity()

//尾部插入节点,队列满时会一直等待可用,响应中断  
public void put(E e)

*/\*\*插入节点,成功返回true,等待timeout时间后无空间返回false\*/*public boolean offer(E e, long timeout, TimeUnit unit)

*/\*\*插入节点\*/*public boolean offer(E e)

*/\*\*获取并消除头节点,会一直等待队列可用,响应中断\*/*public E take()

*/\*\*获取并消除头节点,等待timeout时间后无数据返回null,响应中断\*/*public E poll(long timeout, TimeUnit unit)

*/\*\*获取并消除头节点,忽略中断\*/*public E poll()

*/\*\*获取头节点,不消除\*/*public E peek()

*/\*\*移除o元素\*/*public boolean remove(Object o)

*/\*\*返回是否包含o元素\*/*public boolean contains(Object o)

*/\*\*返回元素列表(数组)\*/*public Object[] toArray()

*/\*\*返回元素列表,指定类型\*/*public <T> T[] toArray(T[] a)

*/\*\*清空队列\*/*public void clear()

*/\*\*移除指定数量的元素,并添加到collection中\*/*public int drainTo(Collection<? super E> c, int maxElements)

##### 数据结构



static class Node<E> {  
 E item;  
  
 */\*\*  
 \* One of:  
 \* - the real successor Node  
 \* - this Node, meaning the successor is head.next  
 \* - null, meaning there is no successor (this is the last node)  
 \*/* Node<E> next;  
  
 Node(E x) { item = x; }  
}  
  
*/\*\* The capacity bound, or Integer.MAX\_VALUE if none \*/*//容量  
private final int capacity;  
  
*/\*\* Current number of elements \*/*//元素个数  
private final AtomicInteger count = new AtomicInteger();  
  
*/\*\*  
 \* Head of linked list.  
 \* Invariant: head.item == null  
 \*/*//链表头  
transient Node<E> head;  
  
*/\*\*  
 \* Tail of linked list.  
 \* Invariant: last.next == null  
 \*/*//链表尾  
private transient Node<E> last;  
  
*/\*\* Lock held by take, poll, etc \*/*//出列锁  
private final ReentrantLock takeLock = new ReentrantLock();  
  
*/\*\* Wait queue for waiting takes \*/*//等待获取(出队)条件  
private final Condition notEmpty = takeLock.newCondition();  
  
*/\*\* Lock held by put, offer, etc \*/*//入列锁  
private final ReentrantLock putLock = new ReentrantLock();  
  
*/\*\* Wait queue for waiting puts \*/*//等待插入(入列)条件  
private final Condition notFull = putLock.newCondition();

**说明**：  
1. LinkedBlockingQueue继承于AbstractQueue，它本质上是一个FIFO(先进先出)的队列。  
2. LinkedBlockingQueue实现了BlockingQueue接口，它支持多线程并发。当多线程竞争同一个资源时，某线程获取到该资源之后，其它线程需要阻塞等待。  
3. LinkedBlockingQueue是通过单链表实现的。  
(01) head是链表的表头。取出数据时，都是从表头head处插入。  
(02) last是链表的表尾。新增数据时，都是从表尾last处插入。  
(03) count是链表的实际大小，即当前链表中包含的节点个数。  
(04) capacity是列表的容量，它是在创建链表时指定的。  
(05) putLock是插入锁，takeLock是取出锁；notEmpty是“非空条件”，notFull是“未满条件”。通过它们对链表进行并发控制。  
       LinkedBlockingQueue在实现“多线程对竞争资源的互斥访问”时，对于“插入”和“取出(删除)”操作分别使用了不同的锁。对于插入操作，通过“插入锁putLock”进行同步；对于取出操作，通过“取出锁takeLock”进行同步。  
       此外，插入锁putLock和“非满条件notFull”相关联，取出锁takeLock和“非空条件notEmpty”相关联。通过notFull和notEmpty更细腻的控制锁。

-- 若某线程(线程A)要取出数据时，队列正好为空，则该线程会执行notEmpty.await()进行等待；当其它某个线程(线程B)向队列中插入了数据之后，会调用notEmpty.signal()唤醒“notEmpty上的等待线程”。此时，线程A会被唤醒从而得以继续运行。 此外，线程A在执行取操作前，会获取takeLock，在取操作执行完毕再释放takeLock。

-- 若某线程(线程H)要插入数据时，队列已满，则该线程会它执行notFull.await()进行等待；当其它某个线程(线程I)取出数据之后，会调用notFull.signal()唤醒“notFull上的等待线程”。此时，线程H就会被唤醒从而得以继续运行。 此外，线程H在执行插入操作前，会获取putLock，在插入操作执行完毕才释放putLock。

##### 源码解析

###### 构造函数

public LinkedBlockingQueue() {  
 this(Integer.*MAX\_VALUE*);  
}  
public LinkedBlockingQueue(int capacity) {  
 if (capacity <= 0) throw new IllegalArgumentException();  
 this.capacity = capacity;  
 last = head = new Node<E>(null);  
}

public LinkedBlockingQueue(Collection<? extends E> c) {  
 this(Integer.*MAX\_VALUE*);  
 final ReentrantLock putLock = this.putLock;  
 putLock.lock(); // Never contended, but necessary for visibility  
 try {  
 int n = 0;  
 for (E e : c) {  
 if (e == null)  
 throw new NullPointerException();  
 if (n == capacity)  
 throw new IllegalStateException("Queue full");  
 enqueue(new Node<E>(e));  
 ++n;  
 }  
 count.set(n);  
 } finally {  
 putLock.unlock();  
 }  
}

**说明:** 链表实现,可选容量

###### 添加(入列)

插入方法包括:put(E),offer(E),源码如下:

//尾部插入节点,队列满时会一直等待可用,响应中断  
public void put(E e) throws InterruptedException {  
 if (e == null) throw new NullPointerException();  
 // Note: convention in all put/take/etc is to preset local var  
 // holding count negative to indicate failure unless set.  
 int c = -1;  
 Node<E> node = new Node<E>(e);  
 final ReentrantLock putLock = this.putLock;  
 final AtomicInteger count = this.count;  
 putLock.lockInterruptibly();  
 try {  
 /\*  
 \* Note that count is used in wait guard even though it is  
 \* not protected by lock. This works because count can  
 \* only decrease at this point (all other puts are shut  
 \* out by lock), and we (or some other waiting put) are  
 \* signalled if it ever changes from capacity. Similarly  
 \* for all other uses of count in other wait guards.  
 \*/  
 while (count.get() == capacity) {  
 notFull.await();  
 }  
 enqueue(node);  
 c = count.getAndIncrement();  
 if (c + 1 < capacity)  
 notFull.signal();  
 } finally {  
 putLock.unlock();  
 }  
 if (c == 0)  
 signalNotEmpty();  
}

*/\*\*插入节点,成功返回true,等待timeout时间后无空间返回false,响应中断\*/*public boolean offer(E e, long timeout, TimeUnit unit)  
 throws InterruptedException {  
  
 if (e == null) throw new NullPointerException();  
 long nanos = unit.toNanos(timeout);  
 int c = -1;  
 final ReentrantLock putLock = this.putLock;  
 final AtomicInteger count = this.count;  
 putLock.lockInterruptibly();  
 try {  
 while (count.get() == capacity) {  
 if (nanos <= 0)  
 return false;  
 nanos = notFull.awaitNanos(nanos);  
 }  
 enqueue(new Node<E>(e));  
 c = count.getAndIncrement();  
 if (c + 1 < capacity)  
 notFull.signal();  
 } finally {  
 putLock.unlock();  
 }  
 if (c == 0)  
 signalNotEmpty();  
 return true;  
}

*/\*\*插入节点\*/*public boolean offer(E e) {  
 if (e == null) throw new NullPointerException();  
 final AtomicInteger count = this.count;  
 if (count.get() == capacity)  
 return false;  
 int c = -1;  
 Node<E> node = new Node<E>(e);  
 final ReentrantLock putLock = this.putLock;  
 putLock.lock();  
 try {  
 if (count.get() < capacity) {  
 enqueue(node);  
 c = count.getAndIncrement();  
 if (c + 1 < capacity)  
 notFull.signal();  
 }  
 } finally {  
 putLock.unlock();  
 }  
 if (c == 0)  
 signalNotEmpty();  
 return c >= 0;  
}

**说明:** put和offer都是在队列尾插入节点,区别是:

1. Put在队列满时会一直等待队列可用,lock是响应中断的
2. Offer分了两个方法,有等待时间和无等待时间,无等待时间的在队列满时直接返回false,lock是忽略中断的.

###### 获取(出列)

获取方法包括:take(),poll(),peek().源码如下:

*/\*\*获取并消除头节点,会一直等待队列可用,响应中断\*/*public E take() throws InterruptedException {  
 E x;  
 int c = -1;  
 final AtomicInteger count = this.count;  
 final ReentrantLock takeLock = this.takeLock;  
 takeLock.lockInterruptibly();  
 try {  
 while (count.get() == 0) {  
 notEmpty.await();  
 }  
 x = dequeue();  
 c = count.getAndDecrement();  
 if (c > 1)  
 notEmpty.signal();  
 } finally {  
 takeLock.unlock();  
 }  
 if (c == capacity)  
 signalNotFull();  
 return x;  
}  
*/\*\*获取并消除头节点,等待timeout时间后无数据返回null,响应中断\*/*public E poll(long timeout, TimeUnit unit) throws InterruptedException {  
 E x = null;  
 int c = -1;  
 long nanos = unit.toNanos(timeout);  
 final AtomicInteger count = this.count;  
 final ReentrantLock takeLock = this.takeLock;  
 takeLock.lockInterruptibly();  
 try {  
 while (count.get() == 0) {  
 if (nanos <= 0)  
 return null;  
 nanos = notEmpty.awaitNanos(nanos);  
 }  
 x = dequeue();  
 c = count.getAndDecrement();  
 if (c > 1)  
 notEmpty.signal();  
 } finally {  
 takeLock.unlock();  
 }  
 if (c == capacity)  
 signalNotFull();  
 return x;  
}  
*/\*\*获取并消除头节点,忽略中断\*/*public E poll() {  
 final AtomicInteger count = this.count;  
 if (count.get() == 0)  
 return null;  
 E x = null;  
 int c = -1;  
 final ReentrantLock takeLock = this.takeLock;  
 takeLock.lock();  
 try {  
 if (count.get() > 0) {  
 x = dequeue();  
 c = count.getAndDecrement();  
 if (c > 1)  
 notEmpty.signal();  
 }  
 } finally {  
 takeLock.unlock();  
 }  
 if (c == capacity)  
 signalNotFull();  
 return x;  
}  
  
*/\*\*获取头节点,不消除\*/*public E peek() {  
 if (count.get() == 0)  
 return null;  
 final ReentrantLock takeLock = this.takeLock;  
 takeLock.lock();  
 try {  
 Node<E> first = head.next;  
 if (first == null)  
 return null;  
 else  
 return first.item;  
 } finally {  
 takeLock.unlock();  
 }  
}

**说明:**

1. take()获取并移除头部元素,在队列为空时一直等待队列可用,lock响应中断
2. poll()会获取并移除头部元素
3. peek()获取元素,但不移除

###### remove(O)

*/\*\*移除o元素\*/*public boolean remove(Object o) {  
 if (o == null) return false;  
 fullyLock();  
 try {  
 for (Node<E> trail = head, p = trail.next;  
 p != null;  
 trail = p, p = p.next) {  
 if (o.equals(p.item)) {  
 unlink(p, trail);//移除p节点  
 return true;  
 }  
 }  
 return false;  
 } finally {  
 fullyUnlock();  
 }  
}

**说明:**   
找到o元素所在的节点,移除节点调用了unlink方法,源码如下:

*/\*\*  
 \* Unlinks interior Node p with predecessor trail.  
 \*/*//移除p节点,trail:p的前节点  
void unlink(Node<E> p, Node<E> trail) {  
 // assert isFullyLocked();  
 // p.next is not changed, to allow iterators that are  
 // traversing p to maintain their weak-consistency guarantee.  
 p.item = null;  
 trail.next = p.next;  
 if (last == p)  
 last = trail;  
 if (count.getAndDecrement() == capacity)  
 notFull.signal();  
}

###### drainTo(C,int)

*/\*\*移除指定数量的元素,并添加到collection中\*/*public int drainTo(Collection<? super E> c, int maxElements) {  
 if (c == null)  
 throw new NullPointerException();  
 if (c == this)  
 throw new IllegalArgumentException();  
 if (maxElements <= 0)  
 return 0;  
 boolean signalNotFull = false;  
 final ReentrantLock takeLock = this.takeLock;  
 takeLock.lock();  
 try {  
 int n = Math.*min*(maxElements, count.get());  
 // count.get provides visibility to first n Nodes  
 Node<E> h = head;  
 int i = 0;  
 try {  
 while (i < n) {  
 Node<E> p = h.next;  
 c.add(p.item);  
 p.item = null;  
 h.next = h;  
 h = p;  
 ++i;  
 }  
 return n;  
 } finally {  
 // Restore invariants even if c.add() threw  
 if (i > 0) {  
 // assert h.item == null;  
 head = h;  
 signalNotFull = (count.getAndAdd(-i) == capacity);  
 }  
 }  
 } finally {  
 takeLock.unlock();  
 if (signalNotFull)  
 signalNotFull();  
 }  
}

很简单 不说明了

#### LinkedBlockingDeque

##### 概述

LinkedBlockingDeque是双向链表实现的双向并发阻塞队列。该阻塞队列同时支持FIFO和FILO两种操作方式，即可以从队列的头和尾同时操作(插入/删除)；并且，该阻塞队列是支持线程安全。

此外，LinkedBlockingDeque还是可选容量的(防止过度膨胀)，即可以指定队列的容量。如果不指定，默认容量大小等于Integer.MAX\_VALUE。

##### 函数列表

*/\*\*插入到队列头,队列满时抛出IllegalStateException\*/*public void addFirst(E e)  
*/\*\*插入到队列尾,队列满时抛出IllegalStateException\*/*public void addLast(E e)

*/\*\*插入到队列头,失败返回false\*/*public boolean offerFirst(E e)

*/\*\*插入到队列尾,失败返回false\*/*public boolean offerLast(E e)

*/\*\*插入到队列头,队列满时一直等待可用\*/*public void putFirst(E e) throws InterruptedException

*/\*\*插入到队列尾,队列满时一直等待可用\*/*public void putLast(E e)

*/\*\*插入队列头,等待timeout时间后若队列无空间返回false,lock响应中断\*/*public boolean offerFirst(E e, long timeout, TimeUnit unit)

*/\*\*插入队列尾,等待timeout时间后若队列无空间返回false,lock响应中断\*/*public boolean offerLast(E e, long timeout, TimeUnit unit)

*/\*\*获取并移除头节点,队列为空抛出NoSuchElementException\*/*public E removeFirst()

*/\*\*获取并移除尾节点,队列为空抛出NoSuchElementException\*/*public E removeLast()

*/\*\*获取并移除头节点,成功返回节点,失败返回null\*/*public E pollFirst()

*/\*\*获取并移除尾节点,成功返回节点,失败返回null\*/*public E pollLast()

*/\*\*获取并移除头节点,队列为空则一直等待队列可用\*/*public E takeFirst()

*/\*\*获取并移除尾节点,队列为空则一直等待队列可用\*/*public E takeLast()

*/\*\*获取并移除头节点,等待timeout时间,lock响应中断\*/*public E pollFirst(long timeout, TimeUnit unit)

*/\*\*获取并移除尾节点,等待timeout时间,lock响应中断\*/*public E pollLast(long timeout, TimeUnit unit)

*/\*\*获取头节点,不移除,队列为空抛出NoSuchElementException\*/*public E getFirst()

*/\*\*获取尾节点,不移除,队列为空抛出NoSuchElementException\*/*public E getLast()

*/\*\*获取头节点,不移除,队列为空返回null\*/*public E peekFirst()

*/\*\*获取尾节点,不移除,队列为空返回null\*/*public E peekLast()

*/\*\*从队列头开始找,移除指定元素节点\*/*public boolean removeFirstOccurrence(Object o)

*/\*\*从队列尾开始找,移除指定元素节点\*/*public boolean removeLastOccurrence(Object o)

*/\*\*插入节点到队列尾{****@link*** *#addLast}\*/*public boolean add(E e)

*/\*\*插入节点到队列尾{****@link*** *#offerLast}\*/*public boolean offer(E e)

*/\*\*插入节点到队列尾{****@link*** *#putLast}\*/*public void put(E e)

*/\*\*插入节点到队列尾{****@link*** *#offerLast}\*/*public boolean offer(E e, long timeout, TimeUnit unit)

*/\*\*获取并移除头节点{****@link*** *#removeFirst}\*/*public E remove()

*/\*\*获取并移除头节点{****@link*** *#pollFirst}\*/*public E poll()

*/\*\*获取并移除头节点{****@link*** *#takeFirst}\*/*public E take()

*/\*\*获取并移除头节点{****@link*** *#pollFirst}\*/*public E poll(long timeout, TimeUnit unit)

*/\*\*获取头节点,不移除{****@link*** *#getFirst}\*/*public E element()

*/\*\*获取头节点,不移除{****@link*** *#peekFirst}\*/*public E peek()

*/\*\*返回剩余容量\*/*public int remainingCapacity()

*/\*\*移除指定数量的元素,并放入指定collection中,返回移除成功的元素数\*/*public int drainTo(Collection<? super E> c, int maxElements)

*/\*\*插入节点到队列头{****@link*** *#addFirst}\*/*public void push(E e)

*/\*\*获取并移除队列头{****@link*** *#removeFirst}\*/*public E pop()

*/\*\*从队列头开始找,移除指定元素节点{****@link*** *#removeFirstOccurrence(Object)}\*/*public boolean remove(Object o)

*/\*\*返回队列元素数量\*/*public int size()

*/\*\*返回是否包含指定元素\*/*public boolean contains(Object o)

*/\*\*返回元素列表\*/*public Object[] toArray()

*/\*\*返回元素列表,指定类型\*/*public <T> T[] toArray(T[] a)

*/\*\*清空队列\*/*public void clear()

##### 数据结构



*/\*\* Doubly-linked list node class \*/*static final class Node<E> {  
 */\*\*  
 \* The item, or null if this node has been removed.  
 \*/* E item;  
  
 */\*\*  
 \* One of:  
 \* - the real predecessor Node  
 \* - this Node, meaning the predecessor is tail  
 \* - null, meaning there is no predecessor  
 \*/* Node<E> prev;  
  
 */\*\*  
 \* One of:  
 \* - the real successor Node  
 \* - this Node, meaning the successor is head  
 \* - null, meaning there is no successor  
 \*/* Node<E> next;  
  
 Node(E x) {  
 item = x;  
 }  
}  
  
*/\*\*  
 \* Pointer to first node.  
 \* Invariant: (first == null && last == null) ||  
 \* (first.prev == null && first.item != null)  
 \*/*transient Node<E> first;  
  
*/\*\*  
 \* Pointer to last node.  
 \* Invariant: (first == null && last == null) ||  
 \* (last.next == null && last.item != null)  
 \*/*transient Node<E> last;  
  
*/\*\* Number of items in the deque \*/*private transient int count;  
  
*/\*\* Maximum number of items in the deque \*/*private final int capacity;  
  
*/\*\* Main lock guarding all access \*/*final ReentrantLock lock = new ReentrantLock();  
  
*/\*\* Condition for waiting takes \*/*private final Condition notEmpty = lock.newCondition();  
  
*/\*\* Condition for waiting puts \*/*private final Condition notFull = lock.newCondition();

**说明**：  
1. LinkedBlockingDeque继承于AbstractQueue，它本质上是一个支持FIFO和FILO的双向的队列。  
2. LinkedBlockingDeque实现了BlockingDeque接口，它支持多线程并发。当多线程竞争同一个资源时，某线程获取到该资源之后，其它线程需要阻塞等待。  
3. LinkedBlockingDeque是通过双向链表实现的。  
first是双向链表的表头。  
last是双向链表的表尾。  
count是LinkedBlockingDeque的实际大小，即双向链表中当前节点个数。  
capacity是LinkedBlockingDeque的容量，它是在创建LinkedBlockingDeque时指定的。  
lock是控制对LinkedBlockingDeque的互斥锁，当多个线程竞争同时访问LinkedBlockingDeque时，某线程获取到了互斥锁lock，其它线程则需要阻塞等待，直到该线程释放lock，其它线程才有机会获取lock从而获取cpu执行权。  
notEmpty和notFull分别是“非空条件”和“未满条件”。通过它们能够更加细腻进行并发控制。

4. LinkedBlockingDeque包含两个迭代器Itr和DescendingItr, Itr是正向迭代,从头开始; DescendingItr与之相反.

##### 源码解析

###### 构造函数

*/\*\*  
 \* Creates a {****@code*** *LinkedBlockingDeque} with a capacity of  
 \* {****@link*** *Integer#MAX\_VALUE}.  
 \*/*public LinkedBlockingDeque() {  
 this(Integer.*MAX\_VALUE*);  
}  
  
*/\*\*  
 \* Creates a {****@code*** *LinkedBlockingDeque} with the given (fixed) capacity.  
 \*  
 \** ***@param*** *capacity the capacity of this deque  
 \** ***@throws*** *IllegalArgumentException if {****@code*** *capacity} is less than 1  
 \*/*public LinkedBlockingDeque(int capacity) {  
 if (capacity <= 0) throw new IllegalArgumentException();  
 this.capacity = capacity;  
}  
  
*/\*\*  
 \* Creates a {****@code*** *LinkedBlockingDeque} with a capacity of  
 \* {****@link*** *Integer#MAX\_VALUE}, initially containing the elements of  
 \* the given collection, added in traversal order of the  
 \* collection's iterator.  
 \*  
 \** ***@param*** *c the collection of elements to initially contain  
 \** ***@throws*** *NullPointerException if the specified collection or any  
 \* of its elements are null  
 \*/*public LinkedBlockingDeque(Collection<? extends E> c) {  
 this(Integer.*MAX\_VALUE*);  
 final ReentrantLock lock = this.lock;  
 lock.lock(); // Never contended, but necessary for visibility  
 try {  
 for (E e : c) {  
 if (e == null)  
 throw new NullPointerException();  
 if (!linkLast(new Node<E>(e)))  
 throw new IllegalStateException("Deque full");  
 }  
 } finally {  
 lock.unlock();  
 }  
}

**说明:** 可选容量

###### 添加(入列)

插入方法包括:  
addFirst(E),addLast(E),offerFirst(E),offerLast(E),putFirst(E),putLast(E),add(E),put(E),offer(E),  
push(E).

源码都是基于linkFirst(node) linkLast(node)实现,具体方法说明见函数列表.这里挑一个插入方法进行分析:

*/\*\*插入节点到队列尾{****@link*** *#addLast}\*/*public boolean add(E e) {  
 addLast(e);  
 return true;  
}

*/\*\*插入到队列尾,队列满时抛出IllegalStateException\*/*public void addLast(E e) {  
 if (!offerLast(e))  
 throw new IllegalStateException("Deque full");  
}

*/\*\*  
 \** ***@throws*** *NullPointerException {****@inheritDoc****}  
 \*/  
/\*\*插入到队列尾,失败返回false\*/*public boolean offerLast(E e) {  
 if (e == null) throw new NullPointerException();  
 Node<E> node = new Node<E>(e);  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 return linkLast(node);  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*插入节点到队列头\*/*private boolean linkFirst(Node<E> node) {  
 // assert lock.isHeldByCurrentThread();  
 if (count >= capacity)  
 return false;  
 Node<E> f = first;  
 node.next = f;  
 first = node;  
 if (last == null)  
 last = node;  
 else  
 f.prev = node;  
 ++count;  
 notEmpty.signal();  
 return true;  
}  
  
*/\*\*  
 \* Links node as last element, or returns false if full.  
 \*/  
/\*\*插入节点到队列尾\*/*private boolean linkLast(Node<E> node) {  
 // assert lock.isHeldByCurrentThread();  
 if (count >= capacity)  
 return false;  
 Node<E> l = last;  
 node.prev = l;  
 last = node;  
 if (first == null)  
 first = node;  
 else  
 l.next = node;  
 ++count;  
 notEmpty.signal();  
 return true;  
}

**说明:** 由于是双向链表,所以LinkedBlockingDeque提供了插入头和尾的方法,分别是由linkFirst和linkLast实现:

首先对队列加锁,判断队列容量是否超出,然后插入节点,修改前节点(prev)或后节点(next),最后唤醒等待notEmpty条件的线程.

###### 获取(出列)

获取方法包括:

removeFirst(),removeLast(),pollFirst(),pollLast(),takeFirst(),takeLast(),getFirst(),getLast(),peekFirst(),peekLast(),remove(),poll(),take(),element(),peek(),pop()

源码都是基于unlinkFirst()和unlinkLast()实现,具体方法说明看函数列表,这里挑一个获取方法进行说明:

*/\*\*获取并移除头节点{****@link*** *#pollFirst}\*/*public E poll(long timeout, TimeUnit unit) throws InterruptedException {  
 return pollFirst(timeout, unit);  
}

*/\*\*获取并移除头节点,等待timeout时间,lock响应中断\*/*public E pollFirst(long timeout, TimeUnit unit)  
 throws InterruptedException {  
 long nanos = unit.toNanos(timeout);  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 E x;  
 while ( (x = unlinkFirst()) == null) {  
 if (nanos <= 0)  
 return null;  
 nanos = notEmpty.awaitNanos(nanos);  
 }  
 return x;  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*移除头结点,成功返回节点,队列空时返回null\*/*private E unlinkFirst() {  
 // assert lock.isHeldByCurrentThread();  
 Node<E> f = first;  
 if (f == null)  
 return null;  
 Node<E> n = f.next;  
 E item = f.item;  
 f.item = null;  
 f.next = f; // help GC  
 first = n;  
 if (n == null)  
 last = null;  
 else  
 n.prev = null;  
 --count;  
 notFull.signal();  
 return item;  
}  
  
*/\*\*  
 \* Removes and returns last element, or null if empty.  
 \*/  
/\*\*移除尾结点,成功返回节点,队列空时返回null\*/*private E unlinkLast() {  
 // assert lock.isHeldByCurrentThread();  
 Node<E> l = last;  
 if (l == null)  
 return null;  
 Node<E> p = l.prev;  
 E item = l.item;  
 l.item = null;  
 l.prev = l; // help GC  
 last = p;  
 if (p == null)  
 first = null;  
 else  
 p.next = null;  
 --count;  
 notFull.signal();  
 return item;  
}

**说明:** 提供了获取头结点和尾节点的方法,都是由unlinkFirst和unlinkLast来实现:

首先对队列加锁,判断队列是否为空,移除头或尾节点,重新定义头或尾节点,然后唤醒等待notFull条件的线程,最后返回获取到(出列)的节点.

###### remove\*(O)

移除方法包括:  
removeFirstOccurrence(O), removeLastOccurrence(),remove(O)

public boolean remove(Object o) {  
 return removeFirstOccurrence(o);  
}

*/\*\*从队列头开始找,移除指定元素节点\*/*public boolean removeFirstOccurrence(Object o) {  
 if (o == null) return false;  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 for (Node<E> p = first; p != null; p = p.next) {  
 if (o.equals(p.item)) {  
 unlink(p);  
 return true;  
 }  
 }  
 return false;  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*移除给定节点\*/*void unlink(Node<E> x) {  
 // assert lock.isHeldByCurrentThread();  
 Node<E> p = x.prev;  
 Node<E> n = x.next;  
 if (p == null) {  
 unlinkFirst();  
 } else if (n == null) {  
 unlinkLast();  
 } else {  
 p.next = n;  
 n.prev = p;  
 x.item = null;  
 // Don't mess with x's links. They may still be in use by  
 // an iterator.  
 --count;  
 notFull.signal();  
 }  
}

**说明:** 移除给定元素所在的节点,都是由unlink(E)方法实现:  
首先对队列加锁,找到元素所在的节点,通过unlink(E)方法移除节点,最后唤醒等待notFull条件的线程.

###### drainTo(C,int)

*/\*\*移除指定数量的元素,并放入指定collection中,返回移除成功的元素数\*/*public int drainTo(Collection<? super E> c, int maxElements) {  
 if (c == null)  
 throw new NullPointerException();  
 if (c == this)  
 throw new IllegalArgumentException();  
 if (maxElements <= 0)  
 return 0;  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 int n = Math.*min*(maxElements, count);  
 for (int i = 0; i < n; i++) {  
 c.add(first.item); // In this order, in case add() throws.  
 unlinkFirst();  
 }  
 return n;  
 } finally {  
 lock.unlock();  
 }  
}

**说明:** 从头部开始移除,通过unlinkFirst()实现.

#### ConcurrentLinkedQueue

##### 概述

ConcurrentLinkedQueue是基于单项链表的线程安全的并发队列;与LinkedBlockingQueue就是并发和阻塞的区别, ConcurrentLinkedQueue使用cas实现并发, LinkedBlockingQueue使用ReentrantLock的机制实现阻塞.适用场景:  
单生产,单消费: LinkedBlockingQueue  
多生产,单消费: LinkedBlockingQueue  
单生产,多消费: ConcurrentLinkedQueue  
单生产,多消费: ConcurrentLinkedQueue

##### 数据结构



private static class Node<E> {  
 volatile E item;  
 volatile Node<E> next;  
  
 */\*\*  
 \* Constructs a new node. Uses relaxed write because item can  
 \* only be seen after publication via casNext.  
 \*/* Node(E item) {  
 *UNSAFE*.putObject(this, *itemOffset*, item);  
 }  
  
 boolean casItem(E cmp, E val) {  
 return *UNSAFE*.compareAndSwapObject(this, *itemOffset*, cmp, val);  
 }  
  
 void lazySetNext(Node<E> val) {  
 *UNSAFE*.putOrderedObject(this, *nextOffset*, val);  
 }  
  
 boolean casNext(Node<E> cmp, Node<E> val) {  
 return *UNSAFE*.compareAndSwapObject(this, *nextOffset*, cmp, val);  
 }  
  
 // Unsafe mechanics  
  
 private static final sun.misc.Unsafe *UNSAFE*;  
 private static final long *itemOffset*;  
 private static final long *nextOffset*;  
  
 static {  
 try {  
 *UNSAFE* = sun.misc.Unsafe.*getUnsafe*();  
 Class<?> k = Node.class;  
 *itemOffset* = *UNSAFE*.objectFieldOffset  
 (k.getDeclaredField("item"));  
 *nextOffset* = *UNSAFE*.objectFieldOffset  
 (k.getDeclaredField("next"));  
 } catch (Exception e) {  
 throw new Error(e);  
 }  
 }  
}

*/\*\*  
 \* A node from which the first live (non-deleted) node (if any)  
 \* can be reached in O(1) time.  
 \* Invariants:  
 \* - all live nodes are reachable from head via succ()  
 \* - head != null  
 \* - (tmp = head).next != tmp || tmp != head  
 \* Non-invariants:  
 \* - head.item may or may not be null.  
 \* - it is permitted for tail to lag behind head, that is, for tail  
 \* to not be reachable from head!  
 \*/  
/\*\*  
 \* 在执行方法之前和之后，head 必须保持的不变式：  
 所有“活着”的节点（指未删除节点），都能从 head 通过调用 succ() 方法遍历可达。  
 head 不能为 null。  
 head 节点的 next 域不能引用到自身。  
 在执行方法之前和之后，head 的可变式：  
 head 节点的 item 域可能为 null，也可能不为 null。  
 允许 tail 滞后（lag behind）于 head，也就是说：从 head 开始遍历队列，不一定能到达 tail。  
 \*/*private transient volatile Node<E> head;  
  
*/\*\*  
 \* A node from which the last node on list (that is, the unique  
 \* node with node.next == null) can be reached in O(1) time.  
 \* Invariants:  
 \* - the last node is always reachable from tail via succ()  
 \* - tail != null  
 \* Non-invariants:  
 \* - tail.item may or may not be null.  
 \* - it is permitted for tail to lag behind head, that is, for tail  
 \* to not be reachable from head!  
 \* - tail.next may or may not be self-pointing to tail.  
 \*/  
/\*\*  
 \* 在执行方法之前和之后，tail 必须保持的不变式：  
 通过 tail 调用 succ() 方法，最后节点总是可达的。  
 tail 不能为 null。  
 在执行方法之前和之后，tail 的可变式：  
 tail 节点的 item 域可能为 null，也可能不为 null。  
 允许 tail 滞后于 head，也就是说：从 head 开始遍历队列，不一定能到达 tail。  
 tail 节点的 next 域可以引用到自身。  
 \*/*private transient volatile Node<E> tail;

**说明：**在后面的源代码分析中，我们将会看到队列有时会处于不一致状态。为此，ConcurrentLinkedQueue 使用三个不变式 ( 基本不变式，head 的不变式和 tail 的不变式 )，来约束队列中方法的执行。通过这三个不变式来维护非阻塞算法的正确性。  
**基本不变式**

在执行方法之前和之后，队列必须要保持的不变式：

* 当入队插入新节点之后，队列中有一个 next 域为 null 的（最后）节点。
* 从 head 开始遍历队列，可以访问所有 item 域不为 null 的节点。

**head 的不变式和可变式**

在执行方法之前和之后，head 必须保持的不变式：

* 所有“活着”的节点（指未删除节点），都能从 head 通过调用 succ() 方法遍历可达。
* head 不能为 null。
* head 节点的 next 域不能引用到自身。

在执行方法之前和之后，head 的可变式：

* head 节点的 item 域可能为 null，也可能不为 null。
* 允许 tail 滞后（lag behind）于 head，也就是说：从 head 开始遍历队列，不一定能到达 tail。

**tail 的不变式和可变式**

在执行方法之前和之后，tail 必须保持的不变式：

* 通过 tail 调用 succ() 方法，最后节点总是可达的。
* tail 不能为 null。

在执行方法之前和之后，tail 的可变式：

* tail 节点的 item 域可能为 null，也可能不为 null。
* 允许 tail 滞后于 head，也就是说：从 head 开始遍历队列，不一定能到达 tail。
* tail 节点的 next 域可以引用到自身。

##### 函数列表

public ConcurrentLinkedQueue() {  
 head = tail = new Node<E>(null);  
}  
public ConcurrentLinkedQueue(Collection<? extends E> c)  
*/\*\*添加节点到队列尾,优于队列是无界的,所以不会返回false或者抛出IllegalStateException\*/*public boolean add(E e) {  
 return offer(e);  
}  
*/\*\*添加节点到队列尾\*/*public boolean offer(E e)  
*/\*\*获取队列头节点\*/*public E poll()  
*/\*\*获取队列头节点,不移除\*/*public E peek()

*/\*\*返回队列是否为空\*/*public boolean isEmpty()

*/\*\*返回队列元素个数(不一定准确)\*/*public int size()

*/\*\*返回是否包含元素o\*/*public boolean contains(Object o)

*/\*\*移除元素\*/*public boolean remove(Object o)

*/\*\*把Collection的元素添加到队列尾\*/*public boolean addAll(Collection<? extends E> c)

*/\*\*返回队列元素列表\*/*public Object[] toArray()

*/\*\*按指定类型返回队列元素列表\*/*public <T> T[] toArray(T[] a)

##### 源码解析

###### add(E)

*/\*\*添加节点到队列尾,由于队列是无界的,所以不会返回false或者抛出IllegalStateException\*/*public boolean add(E e) {  
 return offer(e);  
}  
*/\*\*添加节点到队列尾\*/*public boolean offer(E e) {  
 *checkNotNull*(e);  
 final Node<E> newNode = new Node<E>(e);  
 //自旋,t:尾节点  
 for (Node<E> t = tail, p = t;;) {  
 Node<E> q = p.next;  
 if (q == null) {//p为尾节点  
 // p is last node  
 if (p.casNext(null, newNode)) {//cas替换p的next节点为新节点  
 // Successful CAS is the linearization point  
 // for e to become an element of this queue,  
 // and for newNode to become "live".  
 if (p != t) // hop two nodes at a time跳两个节点时才修改tail  
 casTail(t, newNode); // Failure is OK.cas替换尾节点  
 return true;  
 }  
 // Lost CAS race to another thread; re-read next  
 }  
 else if (p == q)//p=p.next  
 // We have fallen off list. If tail is unchanged, it  
 // will also be off-list, in which case we need to  
 // jump to head, from which all live nodes are always  
 // reachable. Else the new tail is a better bet.  
 //如果尾节点变了,返回p=t,否则p=head,继续循环  
 p = (t != (t = tail)) ? t : head;  
 else  
 // Check for tail updates after two hops.  
 //重新寻找tail节点  
 p = (p != t && t != (t = tail)) ? t : q;  
 }  
}

**说明:** 插入节点到队列尾调用了offer(E)方法，由于队列是无界的，所以插入时不会返回false或者抛出IllegalStateException,首先找到尾节点自旋,定义q为尾节点的下一个节点,这里循环中分三种情况:  
1、q为空，证明p就是尾节点，通过cas把p的next替换为新的节点，成功后判断p!=t，这里为了尽量减少cas次数，只有在跳两个节点时才通过cas替换尾节点，成功返回true；否则证明有其他线程修改了尾节点，继续循环。  
2、p==q，由第三种情况引起，如果尾节点改变返回p=t，否则（头节点改变）返回p=head继续循环。  
3、重新寻找tail节点，如果p==t返回p=q；否则先比较t是否还是尾节点，然后重新给t赋值tail，如果t!=tail，返回p=t(tail)继续循环。

**注意：为了尽量减少cas次数，在入列时，每两次入列才调用casTail更新tail节点，这个控制的实现就是通过p!=t来实现的**

###### poll()

*/\*\*获取并移除队列头节点\*/*public E poll() {  
 restartFromHead:  
 for (;;) {  
 //h：头结点  
 for (Node<E> h = head, p = h, q;;) {  
 E item = p.item;//获取头节点元素  
  
 if (item != null && p.casItem(item, null)) {//cas修改p节点的元素为空  
 // Successful CAS is the linearization point  
 // for item to be removed from this queue.  
 if (p != h) // hop two nodes at a time弹出两个节点时才修改head  
 //cas修改head节点  
 updateHead(h, ((q = p.next) != null) ? q : p);  
 return item;  
 }  
 else if ((q = p.next) == null) {//head.next为空  
 updateHead(h, p);//cas修改head   
 return null;  
 }  
 else if (p == q)  
 continue restartFromHead;//跳转到restartFromHead重新循环  
 else  
 p = q;  
 }  
 }  
}  
*/\*\*修改头节点\*/*final void updateHead(Node<E> h, Node<E> p) {  
 if (h != p && casHead(h, p))  
 h.lazySetNext(h);//设置h的next为自己（链接到自身，做为哨兵节点方便垃圾回收）  
}  
void lazySetNext(Node<E> val) {//惰性设置 next 域的值,可能延迟执行  
 *UNSAFE*.putOrderedObject(this, *nextOffset*, val);  
}

**说明：**获取并移除队列头节点，成功返回节点元素，失败返回null。使用了两个自旋操作，分四种情况：  
1. 找到head节点，如果节点元素不为空则通过cas修改节点元素为null，成功后再利用cas修改head节点（也是在弹出两个节点时才修改head），最后返回节点元素item；  
2. head.next为空，说明队列是只有一个head的的链表，更新head后返回null；  
3. p==q,head的next指向自己，跳转到第一个自旋重新循环；  
4. 设置p=p.next继续下一次循环，到步骤1，此时p=head.next，h=head，p!=h，执行cas函数updatHead，返回item。

**注意：为了尽量减少cas次数，在出列时，每两次弹出才调用casHead更新head节点，这个控制的实现就是通过p!=h来实现的。**

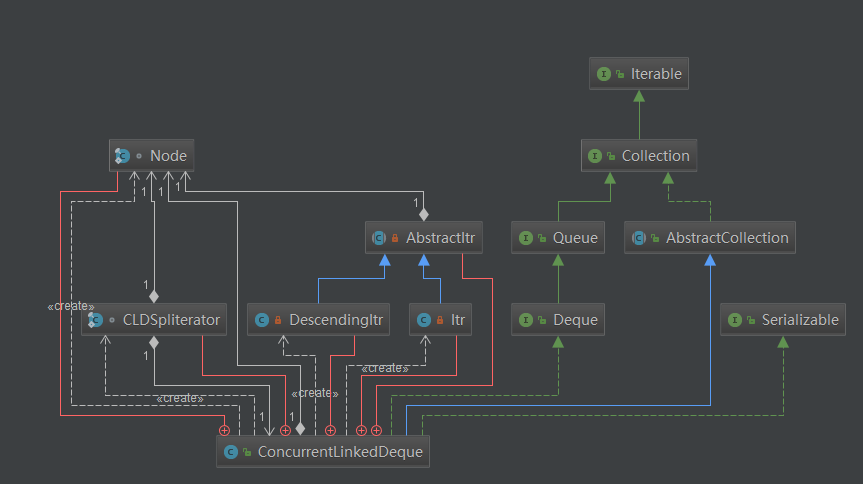
###### peek()

*/\*\*获取队列头节点,不移除\*/*public E peek() {  
 restartFromHead:  
 for (;;) {  
 for (Node<E> h = head, p = h, q;;) {  
 E item = p.item;  
 if (item != null || (q = p.next) == null) {  
 updateHead(h, p);//修改头节点  
 return item;  
 }  
 else if (p == q)//head的next指向自己  
 continue restartFromHead;  
 else  
 p = q;  
 }  
 }  
}

**说明：**获取队列头节点的元素，不移除头节点。具体分析见上小节poll()

#### ConcurrentLinkedDeque

##### 概述



Iterator spliterator

# Java8

## Lambda

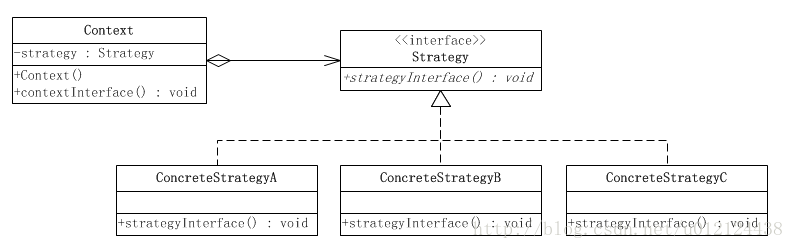
## Java.util.function

# Java9

# 设计模式

## 策略模式

**UML类图**



这个模式涉及到三个角色：

环境(Context)角色：持有一个Strategy的引用。

抽象策略(Strategy)角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。

具体策略(ConcreteStrategy)角色：包装了相关的算法或行为。

项目实例:

AbstractProcessHandler.doProcess0

## 责任链模式

使多个对象都有机会处理请求，从而避免请求的发送者和接受者之间的耦合关系，

将这个对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理他为止。

角色

抽象处理者角色(Handler)：定义出一个处理请求的接口。如果需要，接口可以定义 出一个方法以设定和返回对下家的引用。这个角色通常由一个Java抽象类或者Java接口实现。

具体处理者角色(ConcreteHandler)：具体处理者接到请求后，可以选择将请求处理掉，或者将请求传给下家。由于具体处理者持有对下家的引用，因此，如果需要，具体处理者可以访问下家。

实例: netty **ChannelPipeline-ChannelHandler**

## 反应器(reactor)/观察者(observer)模式

[**Java**](http://lib.csdn.net/base/java)NIO非堵塞技术实际是采取反应器模式，或者说是观察者(observer)模式为我们监察I/O端口，如果有内容进来，会自动通知我们，这样，我们就不必开启多个线程死等，从外界看，实现了流畅的I/O读写，不堵塞了。

同步和异步区别：有无通知（是否轮询）  
堵塞和非堵塞区别：操作结果是否等待（是否马上有返回值），只是设计方式的不同

NIO 有一个主要的类Selector，这个类似一个观察者，只要我们把需要探知的socketchannel告诉Selector，我们接着做别的事情，当有事件发生时，他会通知我们，传回一组SelectionKey，我们读取这些Key，就会获得我们刚刚注册过的socketchannel，然后，我们从这个Channel中读取数据，接着我们可以处理这些数据。

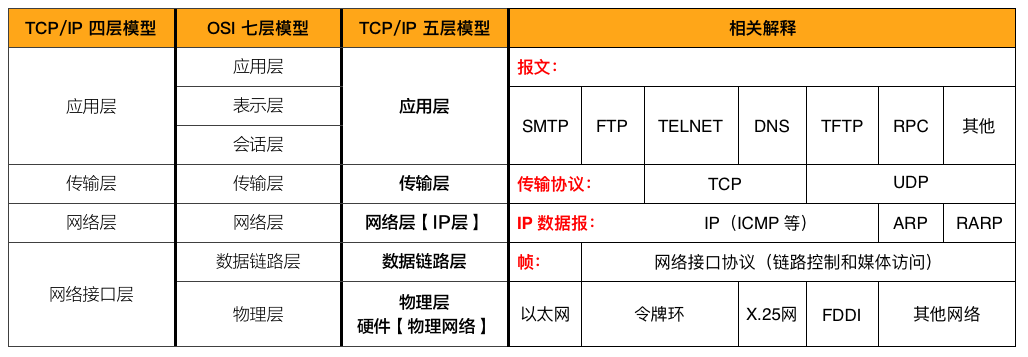
反应器模式与观察者模式在某些方面极为相似：当一个主体发生改变时，所有依属体都得到通知。不过，观察者模式与单个事件源关联，而反应器模式则与多个事件源关联 。

# OpenResty(nginx+lua)

编程式负载均衡, 配置指定服务器接口

# Tcp/Ip Http Socket篇

## 1. 简介



## HTTP「Hypertext Transfer Protocol」超文本传输协议

作用：规定客户端和服务器之间的数据传输格式  
特点：

* 简单快速：HTTP协议简单，HTTP服务器程序小，通信速度快
* 允许传输各种各样的数据
* HTTP 0.9 和 1.0使用 非持续连接，限制每次连接只处理一个请求，请求做出响应后，马上断开，节省传输时间

### 1）发送请求的方法「不同的方法对资源有不同的操作方式」

* PUT：增，给服务器添加资源
* DELETE：删，给服务器删除资源
* POST：改，修改服务器资源
* GET：查，查找服务器资源

### 2）请求参数

传递给服务器的具体数据，比如：帐号、密码  
浏览器和服务器对URL的长度有限制，一般不超过 1K

GET请求参数

* 格式：URL +?+ 参数1=值1 +&+ 参数2=值2「参数间用 & 隔开」  
  多值参数，格式：URL +?+ 参数1=值1 +&+ 参数1=值2
* 安全性差「直接将请求暴露在 URL 里」  
  GET请求的URL一般会记录在服务器的访问日志里，服务器的访问日志是黑客攻击的重点对象之一
* 仅仅是查询数据使用

POST请求参数

* 服务器参数全部放在 **请求体** 中
* 理论上，没有大小限制。实际上，取决于服务器的处理能力
* **安全性好** 即便是 POST也要加密后提交
* 增、删、改服务器数据使用

### 3）HTTP 通信过程 - 请求

I. 请求头：对客户端环境，请求信息的描述

* Host: 120.25.226.186:32812 格式：服务器主机地址: 端口
* GET/minion.png HTTP/1.1 格式：请求方法**/**请求资源路径 HTTP**/**协议版本
* User-Agent: Mozilla/5.0 客户端的类型，客户端的软件环境
* Accept: text/html, \*/\* 客户端所能接收的数据类型
* Accept-Language: zh-cn 客户端的语言环境
* Accept-Encoding: gzip 客户端支持的数据压缩格式

II. 请求体：客户端发给服务器的具体数据，比如：文件数据「POST才会有」

### 4）HTTP 通信过程 - 响应

I. 响应头：服务器的描述，返回数据的描述

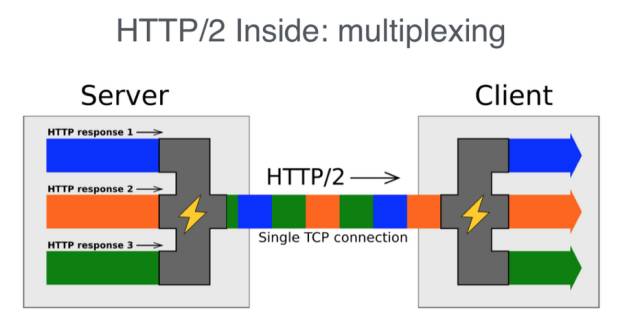
* HTTP/1.1 200 OK 格式：HTTP协议版本 **状态码** 状态名称
* Server: Apache-Coyote/1.1 服务器的类型
* Content-Type: image/jpeg 返回数据的类型
* Content-Length: 56811 返回数据的长度
* Date: Mon, 23 Jun 2014 12:54:52 GMT 响应的时间

II. 响应体：服务器返回客户端的具体数据，比如：文件数据  
III. 常见的响应状态码

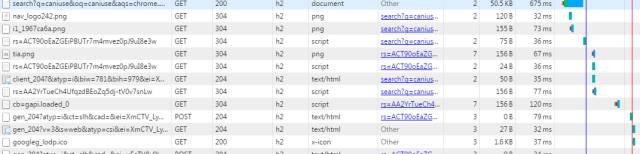
| **状态码** | **英文名称** | **中文描述** |
| --- | --- | --- |
| 200 | OK | 请求成功 |
| 300 |  | 重定向 |
| 400 | Bad Request | 客户端请求语法错误，服务器无法解析 |
| 404 | NotFound | 根据客户端的请求，找不到资源 |
| 500 | Internal Server Error | 服务器内部错误无法完成请求 |

### 5 ) HTTP 2.0

* **新的二进制格式**（Binary Format），HTTP1.x的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认0和1的组合。基于这种考虑HTTP2.0的协议解析决定采用二进制格式，实现方便且健壮。
* HTTP 2.0 二进制分帧层，封装HTTP 消息并在客户端与服务器之间传输
* HTTP2.0 将所有传输的信息分割为更小的消息和帧，并对它们采用二进制格式的编码。
* 注：HTTPS 是二进制分帧的另一个典型示例：所有HTTP 消息都以透明的方式为我们编码和解码，不必对应用进行任何修改。HTTP2.0工作原理有点类似
* **多路复用**（MultiPlexing），即连接共享，即每一个request都是是用作连接共享机制的。一个request对应一个id，这样一个连接上可以有多个request，每个连接的request可以随机的混杂在一起，接收方可以根据request的 id将request再归属到各自不同的服务端请求里面。**多路复用原理图**：



* **header压缩，**如上文中所言，对前面提到过HTTP1.x的header带有大量信息，而且每次都要重复发送，HTTP2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小。
* **服务端\*\***（server push），同SPDY一样，HTTP2.0也具有server push功能。目前，有大多数网站已经启用HTTP2.0，例如YouTuBe，淘宝网等网站，利用chrome控制台可以查看是否启用H2：



### 6 ) HTTPS

1. HTTPS协议需要到ca申请证书，一般免费证书很少，需要交费。
2. HTTP是超文本传输协议，信息是明文传输，HTTPS 则是具有安全性的ssl加密传输协议。
3. HTTP和HTTPS使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
4. HTTPS的连接很简单，HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

## TCP Transmission Control Protocol」传输控制协议

简介

* 面向连接的通信协议，通过**三次握手**建立连接，通讯完成时要拆除连接，**四次挥手**断开连接
* TCP 负责发现传输的问题，一有问题就发出信号，要求重新传输，直到所有数据安全正确地传输到目的地
* 使用 TCP 的协议  
  FTP（文件传输协议）、Telnet（远程登录协议）、SMTP（简单邮件传输协议）  
  POP3（和 SMTP 相对，用于接收邮件）、HTTP 协议等

特点

* 每条传输连接只能有两个端点，只能进行点对点的连接，不支持多播和广播的传输方式
* 连接中可以进行大数据传输「数据不受限制」
* 通信双方可以同时发数据和接收数据
* 每次发送请求使用 TCP 协议  
  必须建立连接「效率会稍低」，是**可靠协议**，送达安全

### I. 建立连接-三次握手

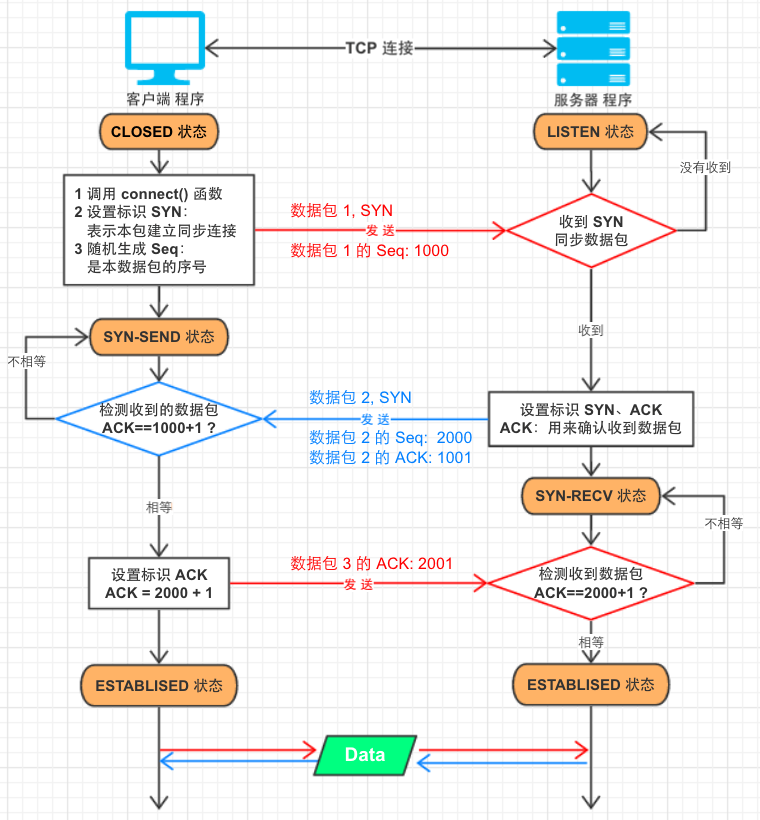
建立连接非常重要，是数据正确传输的前提

模拟步骤

* [Shake 1] 套接字A：“你好，套接字 B，我这里有数据要传送给你，建立连接吧。”
* [Shake 2] 套接字B：“好的，我这边已准备就绪。”
* [Shake 3] 套接字A：“谢谢你受理我的请求。”

具体步骤

* CLOSED 状态：没有建立连接，客户端已经调用 socket() 函数创建了套接字
* LISTEN 状态：服务器端调用 listen() 函数后，开始监听客户端请求  
  服务器没有回传 ACK 包，客户端会重新发送，直到服务器回传 ACK 包
* SYN-SEND 状态：向服务器端发送过了数据包
* SYN-RECV 状态：服务器已经将数据包发出
* ESTABLISED 状态：连接已经成功建立



必要性

* 在「两次握手」的情形下，假设 Client 想跟 Server 建立连接，但是却因为中途连接请求的数据报丢失了，故Client端不得不重新发送一遍，这个时候 Server 端仅收到一个连接请求，因此可以正常的建立连接
* 有时候 Client 端重新发送请求不是因为数据报丢失了，而是有可能数据传输过程因为网络并发量很大在某结点被阻塞了，这种情形下Server端将先后收到 2 次请求，并持续等待两个 Client 请求向他发送数据...
* 问题就在这里，Cient端实际上只有一次请求，而 Server 端却有 2 个响应，极端的情况可能由于 Client 端多次重新发送请求数据而导致 Server端最后建立了 N 多个响应在等待，因而造成极大的资源浪费！

### II. 断开连接-四次挥手

释放不再使用的资源  
不能正常断开，会造成数据传输错误，套接字不能关闭，持续占用资源，如果并发量高，服务器压力堪忧

模拟步骤

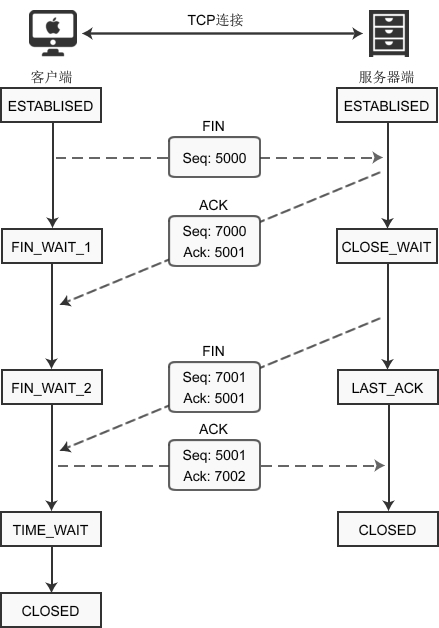
* [Shake 1] 套接字A：“任务处理完毕，我希望断开连接。”
* [Shake 2] 套接字B：“哦，是吗？请稍等，我准备一下。”
* 等待片刻后……
* [Shake 3] 套接字B：“我准备好了，可以断开连接了。”
* [Shake 4] 套接字A：“好的，谢谢合作。”

具体步骤

* FIN 状态：Finish，表示断开连接
* TIME\_WAIT 状态：TIME\_WAIT 要等待 2MSL 才会进入 CLOSED 状态，如果 2MSL 后还未收到服务器重传的 FIN 包，就说明服务器已经收到了 ACK 包  
  客户端最后一次发送 ACK 包时，若服务器收不到，服务器会再次发送 FIN 包  
  如果这时客户端完全关闭了连接，那么服务器无论如何也收不到 ACK 包了，所以客户端需要等待片刻、确认后，才能进入 CLOSED 状态

报文最大生存时间（MSL，Maximum Segment Lifetime）  
数据包在网络中是有生存时间的，超过这个时间还未到达目标主机就会被丢弃，并通知源主机

* CLOSED 状态：断开连接状态



## UDP「User Datagram Protocol」用户数据协议

简介

* 将数据及源和目的封装成数据包中，不需要建立连接
* 使用 UDP协议包括：TFTP（简单文件传输协议）、SNMP（简单网络管理协议）、DNS（域名解析协议）、NFS、BOOTP

特点

* 每个数据报的大小限制在 64K 之内
* 支持单播，组播，广播等多种通信方式
* 每次发送请求使用 UDP 协议  
  无需建立连接，是**不可靠协议**「效率高」

## socket

socket是在应用层和传输层之间的一个抽象层，它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用已实现进程在网络中通信。



* **Socket 是 应用层 和 传输层 之间的桥梁**

HTTP 与 Socket 的区别

1. HTTP 是基于 Socket 的实现；HTTP 应用层协议，主要解决如何包装数据
2. HTTP 传输的数据格式是规定好的，Socket 实现数据传输是最原始，Socket 实现的数据传输格式可自定义
3. Socket 是对 TCP/IP 协议的封装，Socket 本身并不是协议，而是一个调用接口「API」  
   通过 Socket 我们才能使用 TCP/IP 协议

## **HTTP tcp/ip对比**

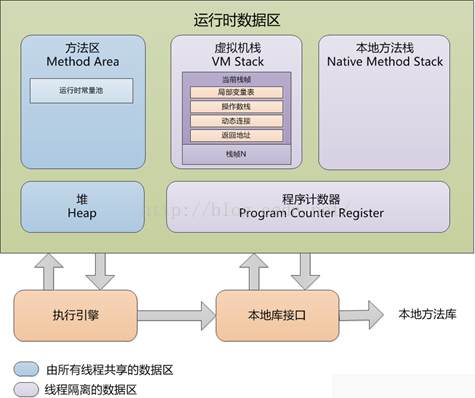
**TPC/IP协议是传输层协议，主要解决数据如何在网络中传输，而HTTP是应用层协议，主要解决如何包装数据。**关于TCP/IP和HTTP协议的关系，网络有一段比较容易理解的介绍：“我们在传输数据时，可以只使用（传输层）TCP/IP协议，但是那样的话，如果没有应用层，便无法识别数据内容，如果想要使传输的数据有意义，则必须使用到应用层协议，应用层协议有很多，比如HTTP、FTP、TELNET等，也可以自己定义应用层协议。WEB使用HTTP协议作应用层协议，以封装HTTP 文本信息，然后使用TCP/IP做传输层协议将它发到网络上。”

术语TCP/IP代表传输控制协议/网际协议，指的是一系列协议。“IP”代表网际协议，TCP和UDP使用该协议从一个网络传送数据包到另一个网络。把**IP想像成一种高速公路**，它允许其它协议在上面行驶并找到到其它电脑的出口。**TCP和UDP是高速公路上的“卡车”，它们携带的货物就是像HTTP**，文件传输协议FTP这样的协议等。

下面的图表试图显示不同的TCP/IP和其他的协议在最初OSI模型中的位置：

|  |  |  |
| --- | --- | --- |
| 7 | **应用层** | 例如[HTTP](http://zh.wikipedia.org/wiki/HTTP)、[SMTP](http://zh.wikipedia.org/wiki/SMTP)、[SNMP](http://zh.wikipedia.org/wiki/SNMP)、[FTP](http://zh.wikipedia.org/wiki/FTP)、[Telnet](http://zh.wikipedia.org/wiki/Telnet)、[SIP](http://zh.wikipedia.org/wiki/SIP)、[SSH](http://zh.wikipedia.org/wiki/SSH)、[NFS](http://zh.wikipedia.org/wiki/NFS)、[RTSP](http://zh.wikipedia.org/wiki/RTSP)、[XMPP](http://zh.wikipedia.org/wiki/XMPP)、[Whois](http://zh.wikipedia.org/wiki/Whois)、[ENRP](http://zh.wikipedia.org/w/index.php?title=ENRP&action=edit&redlink=1) |
| 6 | **表示层** | 例如[XDR](http://zh.wikipedia.org/w/index.php?title=External_Data_Representation&action=edit&redlink=1)、[ASN.1](http://zh.wikipedia.org/w/index.php?title=Abstract_Syntax_Notation_1&action=edit&redlink=1)、[SMB](http://zh.wikipedia.org/w/index.php?title=Server_message_block&action=edit&redlink=1)、[AFP](http://zh.wikipedia.org/w/index.php?title=Apple_Filing_Protocol&action=edit&redlink=1)、[NCP](http://zh.wikipedia.org/w/index.php?title=NetWare_Core_Protocol&action=edit&redlink=1) |
| 5 | **会话层** | 例如[ASAP](http://zh.wikipedia.org/w/index.php?title=Aggregate_Server_Access_Protocol&action=edit&redlink=1)、[TLS](http://zh.wikipedia.org/wiki/Transport_Layer_Security)、[SSH](http://zh.wikipedia.org/wiki/SSH)、ISO 8327 / CCITT X.225、[RPC](http://zh.wikipedia.org/w/index.php?title=Remote_procedure_call&action=edit&redlink=1)、[NetBIOS](http://zh.wikipedia.org/w/index.php?title=NetBIOS&action=edit&redlink=1)、[ASP](http://zh.wikipedia.org/w/index.php?title=AppleTalk&action=edit&redlink=1)、[Winsock](http://zh.wikipedia.org/w/index.php?title=Winsock&action=edit&redlink=1)、[BSD sockets](http://zh.wikipedia.org/wiki/Berkeley_sockets) |
| 4 | **传输层** | 例如[TCP](http://zh.wikipedia.org/wiki/TCP)、[UDP](http://zh.wikipedia.org/wiki/User_Datagram_Protocol)、[RTP](http://zh.wikipedia.org/w/index.php?title=Real-time_Transport_Protocol&action=edit&redlink=1)、[SCTP](http://zh.wikipedia.org/w/index.php?title=Stream_Control_Transmission_Protocol&action=edit&redlink=1)、[SPX](http://zh.wikipedia.org/w/index.php?title=Sequenced_packet_exchange&action=edit&redlink=1)、[ATP](http://zh.wikipedia.org/w/index.php?title=AppleTalk&action=edit&redlink=1)、[IL](http://zh.wikipedia.org/w/index.php?title=IL_Protocol&action=edit&redlink=1) |
| 3 | **网络层** | 例如[IP](http://zh.wikipedia.org/wiki/%E7%BD%91%E9%99%85%E5%8D%8F%E8%AE%AE)、[ICMP](http://zh.wikipedia.org/wiki/ICMP)、[IGMP](http://zh.wikipedia.org/wiki/IGMP)、[IPX](http://zh.wikipedia.org/wiki/IPX)、[BGP](http://zh.wikipedia.org/wiki/BGP)、[OSPF](http://zh.wikipedia.org/wiki/OSPF)、[RIP](http://zh.wikipedia.org/wiki/RIP)、[IGRP](http://zh.wikipedia.org/wiki/IGRP)、[EIGRP](http://zh.wikipedia.org/wiki/EIGRP)、[ARP](http://zh.wikipedia.org/wiki/ARP)、[RARP](http://zh.wikipedia.org/wiki/RARP)、 [X.25](http://zh.wikipedia.org/wiki/X.25) |
| 2 | **数据链路层** | 例如[以太网](http://zh.wikipedia.org/wiki/%E4%BB%A5%E5%A4%AA%E7%BD%91)、[令牌环](http://zh.wikipedia.org/wiki/%E4%BB%A4%E7%89%8C%E7%8E%AF)、[HDLC](http://zh.wikipedia.org/wiki/HDLC)、[帧中继](http://zh.wikipedia.org/wiki/%E5%B8%A7%E4%B8%AD%E7%BB%A7)、[ISDN](http://zh.wikipedia.org/wiki/ISDN)、[ATM](http://zh.wikipedia.org/wiki/%E5%BC%82%E6%AD%A5%E4%BC%A0%E8%BE%93%E6%A8%A1%E5%BC%8F)、[IEEE 802.11](http://zh.wikipedia.org/wiki/IEEE_802.11)、[FDDI](http://zh.wikipedia.org/wiki/FDDI)、[PPP](http://zh.wikipedia.org/wiki/PPP) |
| 1 | **物理层** | 例如[线路](http://zh.wikipedia.org/w/index.php?title=%E7%BA%BF%E8%B7%AF&action=edit&redlink=1)、[无线电](http://zh.wikipedia.org/wiki/%E6%97%A0%E7%BA%BF%E7%94%B5)、[光纤](http://zh.wikipedia.org/wiki/%E5%85%89%E7%BA%A4)、[信鸽](http://zh.wikipedia.org/wiki/%E4%BF%A1%E9%B8%BD) |

# JVM内存模型



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **名称** | **特征** | **作用** | **配置参数** | **异常** |
| 程序计数器 | 占用内存小，线程私有，  生命周期与线程相同 | 大致为字节码行号指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。 | 无 | 无 |
| 虚拟机栈 | 线程私有，生命周期与线程相同，使用连续的内存空间 | Java 方法执行的内存模型，存储局部变量表、操作栈、动态链接、方法出口等信息 | -Xss | StackOverflowError  OutOfMemoryError |
| java堆 | 线程共享，生命周期与虚拟机相同，可以不使用连续的内存地址 | 保存对象实例，所有对象实例（包括数组）都要在堆上分配 | -Xms  -Xsx  -Xmn | OutOfMemoryError |
| 方法区 | 线程共享，生命周期与虚拟机相同，可以不使用连续的内存地址 | 存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据 | -XX:PermSize:  16M  -XX:MaxPermSize  64M | OutOfMemoryError |
| 运行时常量池 | 方法区的一部分，具有动态性 | 存放字面量及符号引用 |  |  |

## 内存分配过程

1、JVM 会试图为相关Java对象在Eden Space中初始化一块内存区域。

2、当Eden空间足够时，内存申请结束；否则到下一步。

3、JVM 试图释放在Eden中所有不活跃的对象（这属于1或更高级的垃圾回收）。释放后若Eden空间仍然不足以放入新对象，则试图将部分Eden中活跃对象放入Survivor区。

4、Survivor区被用来作为Eden及Old的中间交换区域，当Old区空间足够时，Survivor区的对象会被移到Old区，否则会被保留在Survivor区。

5、当Old区空间不够时，JVM 会在Old区进行完全的垃圾收集（0级）。

6、完全垃圾收集后，若Survivor及Old区仍然无法存放从Eden复制过来的部分对象，导致JVM无法在Eden区为新对象创建内存区域，则出现“outofmemory”错误。

# 二进制及运算篇

<< 向左移位

>> 向右移位

>>> 无符号右移 ,忽略符号位并总是以”0”作为填充位

/\* 00000001 << 1 = 00000010 \*/  
 1 << 1 == 2   
   
/\* 00000001 << 3 = 00001000 \*/  
 1 << 3 == 8  
   
/\* 11111111 11111111 11111111 11110000 >> 4 = 11111111 11111111 11111111 11111111 \*/  
 0xFFFFFFF0 >> 4 == 0xFFFFFFFF   
   
/\* 00001111 11111111 11111111 11111111 >> 4 = 00000000 11111111 11111111 11111111 \*/  
 0x0FFFFFFF >> 4 == 0x00FFFFFF

/\* 10000000 00000000 00000000 00000000 >>> 1 = 01000000 00000000 00000000 00000000 \*/  
 0x80000000 >>> 1 == 0x40000000  
   
/\* 10000000 00000000 00000000 00000000 >> 1 = 11000000 00000000 00000000 00000000 \*/  
 0x80000000 >> 1 == 0xC0000000

**注意：**向右移位是有符号操作符。和许多语言一样，Java使用最高位来表示数值的正负，负数的最高位永远为1。一个以1开头的二进制数移位后还将以1开头，一个以0开头的二进制树移位后还将以0开头。所以**要小心**：Java是可以在整数中进行位运算的。

最大的用途之一是迅速求2的幂。1向左移位1位是2，移2位是4，移3位是8…… 相似的，向右移1位相当于是把该数除以2。

另一个用途便是创建掩码。位掩码可用于屏蔽或者修改一个二进制数中的某些指定位，下一部分会进行详细讲解。假如我们想要创建一个  
*00001000*的掩码，代码十分简单：

|  |  |
| --- | --- |
|  | int bitmask = 1 << 3; |

n<<1 = n\*2

n>>1 = n/2取整数

n<<m = n\*2m

n>>m = n/2m

~n = -(n+1) ~10 = -11

* ~ 按位取反
* ~1111 == 0000
* ~0011 == 1100
* & 按位与 取0
* 1010 & 0101 == 0000
* 1100 & 0110 == 0100
* ^ 按位异或 同0异1
* 1010 ^ 0101 == 1111
* 1100 ^ 0110 == 1010
* | 按位或 取1
* 1010 | 0101 == 1111
* 1100 | 0110 == 1110