## JVM &GC

### Import notes about GC(Garbage Collection) in Java

1. Objects are created on heap in Java irrespective of their scopes, e.g. local or member variables. Both **heap area** and **method area** are shared between different threads;
2. GC is a mechanism provided by JVM to reclaim heap space from objects which are eligible for garbage collection;
3. GC in java is carried by a daemon thread called **Garbage Collector**;
4. Before removing an object from memory garbage collector thread invokes finalized() method of that object and gives an opportunity to perform any sort of cleanup required;
5. Methods like **System.gc()** and **Runtime.gc()** are used to send request to Garbage collection to JVM but it’s not guaranteed that GC will happen. It will only be triggered when JVM thinks it needs based on heap size;

### When an Object becomes Eligible for Garbage Collection

An object becomes eligible for Garbage collection or GC if it’s not reachable from any live threads or by any static references. (Cyclic reference is not counted in), typical cases are:

1. All references of that object explicitly set to null e.g. object = null;
2. Object is created inside a block and reference goes out scope once control exit that block;
3. Parent object set to null, if an object holds reference of another object and when you set container object's reference null, child or contained object automatically becomes eligible for garbage collection;
4. If an object has only **live weak references** via  **WeakHashMap**  it will be eligible for garbage collection;

### Heap Generations for GC in java

1. **Young generation**

When an object first created in heap its gets created in new generation inside Eden space and after subsequent minor garbage collection if object survives its gets moved to survivor 1 and then survivor 2 before major garbage collection moved that object to old or tenured generation.

When objects disappear from this area, we say a "**minor GC**" has occurred.

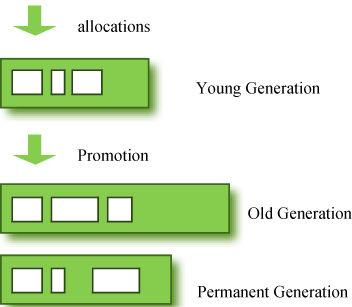
1. **Old/Tenured generation**

The objects that did not become unreachable and survived from the young generation are copied here. It is generally larger than the young generation. As it is bigger in size, the GC occurs less frequently than in the young generation. When objects disappear from the old generation, we say a "**major GC**" (or a "**full GC**") has occurred.

1. **Perm Area**

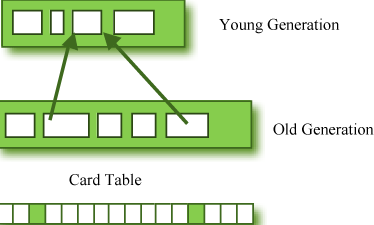
Also called the "**method area**” and it stores classes or interned character strings. So, this area is definitely not for objects that survived from the old generation to stay permanently. A GC may occur in this area. The GC that took place here is still counted as a major GC.

used to store String pool and various Meta data required by JVM related to Class, method and other java primitives**; In most of JVM default size of Perm Space is around "64MB".**



**“What if an object in the old generation need to reference an object in the young generation?”**

To handle these cases, there is something called the a "**card table**" in the old generation, which is a *512 byte chunk*. Whenever an object in the old generation references an object in the young generation, it is recorded in this table. When a GC is executed for the young generation, only this card table is searched to determine whether or not it is subject for GC, instead of checking the reference of all the objects in the old generation. This card table is managed with **write barrier**. This *write barrier* is a device that allows a faster performance for minor GC. Though a bit of overhead occurs because of this, the overall GC time is reduced.



### Cases and Principles of tuning JVM

1. If an application has too many short lived object then making Eden space wide enough or larger will reduces number of minor collections;
2. String pool is created in PermGen area of Heap, garbage collection can occur in perm space but depends upon JVM to JVM. By the way from JDK 1.7 update, String pool is moved to heap area where objects are created.
3. **Minor Garbage Collection** is used to move object from Eden space to Survivor 1 and Survivor 2 space and **Major Collection** is used to move object from young to tenured generation.
4. Whenever **Major Garbage Collection** occurs application threads stops during that period which will reduce application’s performance and throughput.
5. There are few performance improvement has been applied in garbage collection in java 6 and we usually use JRE 1.6.20 for running our application.
6. JVM command line options **–Xmx** and **-Xms** is used to setup starting and max size for Java Heap. Ideal ratio of this parameter is either 1:1 or 1:1.5 based upon my experience for example you can have either both –Xmx and –Xms as 1GB or –Xms 1.2 GB and 1.8 GB.

### JVM options

1. JVM Options that **begin with -X are non-standard** (they are not guaranteed to be supported on all JVM implementations), and are subject to change without notice in subsequent releases of the JDK.

JVM Options or parameters which are **specified with -XX are not stable and are not recommended for casual use**. These options are subject to change without notice also.

1. VM memory options related to java heap size
2. **-Xms**        set initial Java heap size

b)  **-Xmx** set maximum Java heap size

c) **-Xss**         set java thread stack size

1. JVM option to print gc details

a) **-verbose:gc**

logs garbage collector runs and how long they're taking. I generally use this as my first tool to investigate if GC is a bottleneck for a given application.

b) **-XX:+PrintGCDetails**

 includes the data from -verbose:gc but also adds information about the size of the new generation and more accurate timings.

c) **-XX:-PrintGCTimeStamps**

Print timestamps at garbage collection.

1. JVM parameters to specify Java Garbage collector

-XX:+UseParallelGC      Use parallel garbage collection for scavenges

-XX:-UseConcMarkSweepGC Use concurrent mark-sweep collection for the old generation. (Introduced in 1.4.1)

-XX:-UseSerialGC        Use serial garbage collection. (Introduced in 5.0.)

1. JVM options to change Perm Gen Size

-XX:PermSize and –XX:MaxPermSize

-XX:NewRatio=2  Ratio of new/old generation sizes.

-XX:MaxPermSize=64m     Size of the Permanent Generation.

### Deal with “java.lang.OutOfMemoryError”

Permanent generation of heap is used to store String pool and various Meta data required by JVM related to Class, method and other java primitives. Since in **most of JVM default size of Perm Space is around "64MB"** you can easily ran out of memory if you have too many classes or huge number of Strings in your project. Important point to remember is that it doesn't depends on –Xmx value so no matter how big your total heap size you can ran OutOfMemory in perm space. Good think is you can **specify size of permanent generation** using JVM options **"-XX:PermSize" and  "-XX:MaxPermSize"** based on your project need.

1. Java.lang.OutOfMemoryError: Java heap space

**-Xms1024m -Xmx1024m**

2)  Java.lang.OutOfMemoryError: PermGen space  
 **-XX:PermSize=64M -XX:MaxPermSize=256m**

### Solution of Tomcat: OutOfMemroyError

**Case: OutOfMemoryError: PermGen space in TOMCAT server**

PermGen Space of heap is used to store classes and Meta data about classes in Java. When a class is loaded by a class loader it got stored in **PermGen** space, it gets unloaded only when the class loader which loaded this class got garbage collected. If any object retains reference of class loader than its not garbage collected and Perm Gen Space is not freed up. This causes memory leak in **PermGen** Space and eventually cause *java.lang.OutOfMemoryError: PermGen space.* Another important point is that when you deploy your web application a new Class loader gets created and it loads the classes used by web application. So if Class loader doesn't get garbage collected when your web application stops you will have memory leak in tomcat.

1. Find the offending classes which are retaining reference of [*Classloader*](http://javarevisited.blogspot.com/2012/12/how-classloader-works-in-java.html)  and preventing it from being garbage collected. Tomcat provides memory leak detection functionality after tomcat 6 onwards which can help you to find when particular library, framework or class is causing memory leak in tomcat. Here are some of the common causes:
2. JDBC Drivers:

JDBC drivers are most common cause *of* java.lang.OutOfMemoryError: PermGen space in tomcat if web app doesn't unregister during stop. One hack to get around this problem is that JDBC driver to be loaded by common class loader than application classloader and you can do this by transferring driver's jar into tomcat lib instead of bundling it on web application's war file.

1. Logging framework:

Similar solution can be applied to prevent logging libraries like Log4j causing java.lang.OutOfMemoryError: PermGen space in tomcat.

1. Application Threads which have not stopped.

Check your code carefully if you are leaving your [thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) unattended and running in while loop that can retain classloader's reference and cause java.lang.OutOfMemoryError: PermGen space in tomcat web server. Another common culprit is ThreadLocal, avoid using it until you need it absolutely, if do you make sure to set them null or free any object [ThreadLocal variables](http://javarevisited.blogspot.com/2012/05/how-to-use-threadlocal-in-java-benefits.html) are holding.

1. Another Simple Solution is to increase PermGen [heap size](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html) in catalina.bat or catalina.sh of tomcat server; this can give you some breathing space but eventually this will also return in java.lang.OutOfMemoryError: PermGen space after some time.

a) Go to Tomcat installation directory i.e C:\Program Files\Apache Software Foundation\Apache Tomcat 7.0.14\bin in Windows and something similar in linux.

b) Add JAVA\_OPTS in your catalina.bat or Catalina.sh

In Windows:

set JAVA\_OPTS="-Xms1024m -Xmx10246m -XX:NewSize=256m -XX:MaxNewSize=356m -XX:PermSize=256m -XX:MaxPermSize=356m"

In linux:

export JAVA\_OPTS="-Xms1024m -Xmx10246m -XX:NewSize=256m -XX:MaxNewSize=356m -XX:PermSize=256m -XX:MaxPermSize=356m"

You can change the actual heap size and PermGen Space as per your requirement.

1. Restart Tomcat.

As I said earlier increasing PermGen space can prevent java.lang.OutOfMemoryError: PermGen in tomcat only for some time and it will eventually occur based on how many times you redeploy your web application, its best to find the offending class which is causing memory leak in tomcat and fix it.

## Reference Types in Java & GC

There are four kind of reference in Java: Strong reference, Weak Reference, Soft Reference and Phantom Reference, along with the concept of ReferenceQueue.

#### Strong reference

Strong Reference is most simple as we use it in our day to day programming life e.g. in the code, String s = "abc" , reference variable s has strong reference to String object "abc". Any object which has Strong reference attached to it is not eligible for garbage collection. Obviously these are objects which is needed by Java program.

#### Weak Reference

Weak Reference are represented using java.lang.ref.WeakReference class and you can create Weak Reference by using following code :

**Counter** counter **=** **new** **Counter**(); // strong reference - line 1

**WeakReference**<**Counter**> weakCounter **=** **new** **WeakReference**<**Counter**>(counter); //weak reference

counter **=** **null**; // now Counter object is eligible for garbage collection

Now as soon as you make strong reference counter = null, counter object created on line 1 becomes eligible for garbage collection; because it doesn't have any more Strong reference and Weak reference by reference variable weakCounter can't prevent Counter object from being garbage collected.

One convenient example of WeakReference is WeakHashMap, which is another implementation of Map interface like HashMap or TreeMap but with one unique feature. WeakHashMap wraps keys as WeakReference which means once strong reference to actual object removed,  WeakReference present internally on WeakHashMap doesn't prevent them from being Garbage collected.

#### Soft Reference

For the example above, on the other hand, had that been Soft Reference, Counter object is not garbage collected until JVM absolutely needs memory. Soft reference in Java is represented using java.lang.ref.SoftReference class. You can use following code to create a SoftReference in Java:

**Counter** prime **=** **new** **Counter**(); // prime holds a strong reference—line 2

**SoftReference**<**Counter**> soft **=** **new** **SoftReference**<**Counter**>(prime) ; //soft reference variable has SoftReference to Counter Object created at line 2

prime **=** **null**; // now Counter object is eligible for garbage collection but only be collected when JVM absolutely needs memory

**Counter** prime2=soft.get();//**At any given point of time, if the garbage collector has not yet reclaimed the memory of the actual referent object, you can get a strong reference to the referent via the get method.**

After making strong reference null, Counter object created on line 2 only has one soft reference which can’t prevent it from being garbage collected but it can delay collection, which is eager in case of WeakReference. Due to this major *difference between SoftReference and WeakReference*, SoftReference are more suitable for **caches**and WeakReference are more suitable for storing**meta data**.

#### Phantom Reference

Phantom reference is third kind of reference type available in java.lang.ref package. Phantom reference is represented by java.lang.ref.PhantomReference class. Object which only has Phantom reference pointing them can be collected whenever Garbage Collector likes it. Similar to WeakReference and SoftReference you can create PhantomReference by using following code :

**DigitalCounter** digit **=** **new** **DigitalCounter**(); // digit reference variable has strong reference - line 3

**PhantomReference**<**DigitalCounter**> phantom **=** **new** **PhantomReference**<**DigitalCounter**>(digit); // phantom reference to object created at line 3

digit **=** **null**;

As soon as you remove Strong reference, DigitalCounter object created at line 3 can be garbage collected at any time as it only has one more PhantomReference pointing towards it, which can’t prevent it from GC'd.

A phantom reference is always associated with a references queue during construction time. This is because phantom references are enqueued in the queue only when the the object is about to be garbage collected, after the finalize method(if any) has been executed on it. Calling a get() on the Phantom reference always returns null. And that is quite appropriate because the finalize function has already run on the referent object.

[ 那虚引用到底有什么作用？其实虚引用主要被用来 跟踪对象被垃圾回收的状态，通过查看引用队列中是否包含对象所对应的虚引用来判断它是否 即将被垃圾回收，从而采取行动。它并不被期待用来取得目标对象的引用，而目标对象被回收前，它的引用会被放入一个 ReferenceQueue 对象中，从而达到跟踪对象垃圾回收的作用。]

#### ReferenceQueue

Apart from knowing WeakReference, SoftReference, PhantomReference and WeakHashMap, there is one more class called ReferenceQueue which is worth knowing.

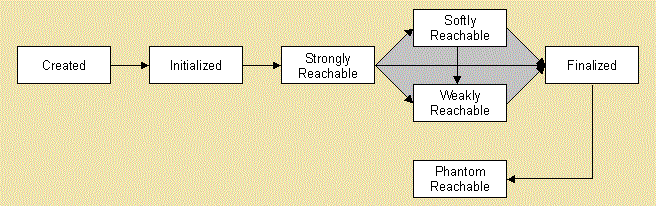
You can supply a ReferenceQueue instance while creating any WeakReference, SoftReference or PhantomReference as shown in following code :

Reference of instance will be appended to ReferenceQueue and you can use it to perform any clean-up by polling ReferenceQueue. An Object's life-cycle is nicely summed up by this diagram:

**ReferenceQueue** refQueue **=** **new** **ReferenceQueue**(); //reference will be stored in this queue for cleanup

**DigitalCounter** digit **=** **new** **DigitalCounter**();

**PhantomReference**<**DigitalCounter**> phantom **=** **new** **PhantomReference**<**DigitalCounter**>(digit, refQueue);



## GC Algorithms

（整理自<http://www.cnblogs.com/zuoxiaolong/p/jvm6.html> ）

1. 标记清除算法（ Mark & Sweep）、

**标记**：标记的过程其实就是，遍历所有的GC Roots，然后将所有GC Roots可达的对象标记为存活的对象。

**清除**：清除的过程将遍历堆中所有的对象，将没有标记的对象全部清除掉。

2．复制算法（copy）

复制算法将内存划分为两个区间，在任意时间点，所有动态分配的对象都只能分配在其中一个区间（称为活动区间），而另外一个区间（称为空闲区间）则是空闲的。

当有效内存空间耗尽时，JVM将暂停程序运行，开启复制算法GC线程。接下来GC线程会将活动区间内的存活对象，全部复制到空闲区间，且严格按照内存地址依次排列，与此同时，GC线程将更新存活对象的内存引用地址指向新的内存地址。

复制算法弥补了标记/清除算法中，内存布局混乱的缺点。不过与此同时，它的缺点也是相当明显的。

1、它浪费了一半的内存，这太要命了。

2、如果对象的存活率很高，我们可以极端一点，假设是100%存活，那么我们需要将所有对象都复制一遍，并将所有引用地址重置一遍。复制这一工作所花费的时间，在对象存活率达到一定程度时，将会变的不可忽视。

所以从以上描述不难看出，复制算法要想使用，最起码对象的存活率要非常低才行，而且最重要的是，我们必须要克服50%内存的浪费。

3． 标记整理算法

**标记**：它的第一个阶段与标记/清除算法是一模一样的，均是遍历GC Roots，然后将存活的对象标记。

**整理**：移动所有存活的对象，且按照内存地址次序依次排列，然后将末端内存地址以后的内存全部回收。因此，第二阶段才称为整理阶段

4．分代回收算法

分代回收算法是对以上算法的综合使用；

1. 对新生代，对象存活时间短，一次回收率可以达到90%，因此使用复制算法;
2. 对年老代和永久代，回收频率不频繁，因此使用标记整理算法；

（详细见 <http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/> ）

## ClassLoader in Java

#### Classloaders in General

There are three default class loader used in Java, Bootstrap , Extension and System or Application class loader. Every class loader has a predefined location, from where they loads class files. Except Bootstrap class loader, which is implemented in native language mostly in C,  all  Java class loaders are implemented using java.lang.ClassLoader.

1. Bootstrap class loader

Responsible for loading standard JDK class files from rt.jar and it is parent of all class loaders in Java. Bootstrap class loader don't have any parents, if you call String.class.getClassLoader() it will return null.

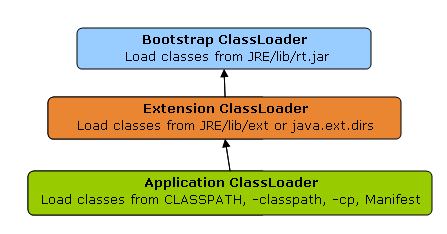
Bootstrap class loader is also known as**Primordial ClassLoader** in Java.

1. Extension class loader

Extension ClassLoader delegates class loading request to its parent, Bootstrap and if unsuccessful, loads class form jre/lib/ext directory or any other directory pointed by java.ext.dirs system property. Extension ClassLoader in JVM is implemented by  sun.misc.Launcher$ExtClassLoader.

1. System or Application class loader

responsible for loading application specific classes from CLASSPATH environment variable, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR. Application class loader is a child of Extension ClassLoader and it’s implemented by sun.misc.Launcher$AppClassLoader class.



#### How ClassLoader works in Java

1. Delegation principles

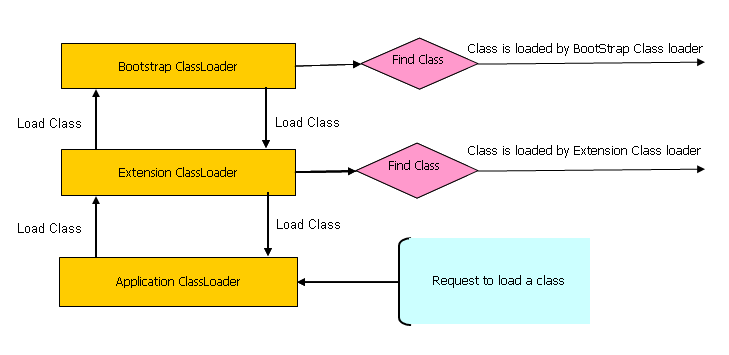
Suppose you have an application specific class called Abc.class, first request of loading this class will come to Application ClassLoader which will delegate to its parent Extension ClassLoader which further delegates to Primordial or Bootstrap class loader. Primordial will look for that class in rt.jar and since that class is not there, request comes to Extension class loader which looks on jre/lib/ext directory and tries to locate this class there, if class is found there than Extension class loader will load that class and Application class loader will never load that class but if it’s not loaded by extension class-loader than Application class loader loads it from Classpath in Java.

1. Visibility Principles

According to visibility principle, Child ClassLoader can see class loaded by Parent ClassLoader but vice-versa is not true. Which mean if class Abc is loaded by Application class loader than trying to load class ABC explicitly using extension ClassLoader will throw either java.lang.ClassNotFoundException.

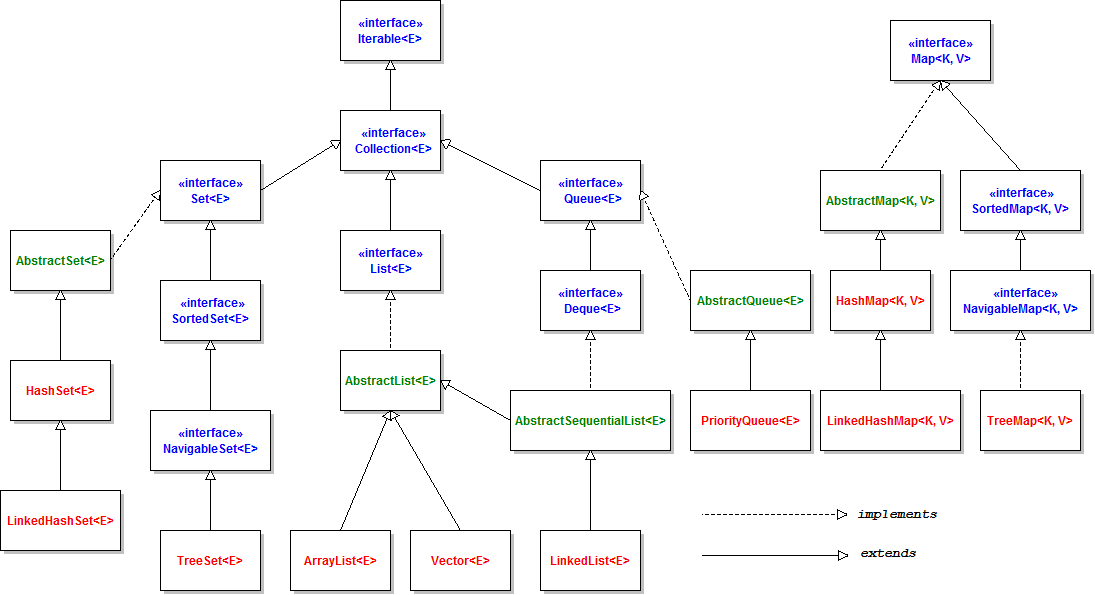
1. Uniqueness Principle

According to this principle a class loaded by Parent should not be loaded by Child ClassLoader again. Though its completely possible to write class loader which violates Delegation and Uniqueness principles and loads class by itself, its not something which is beneficial. You should follow all class loader principle while writing your own ClassLoader.



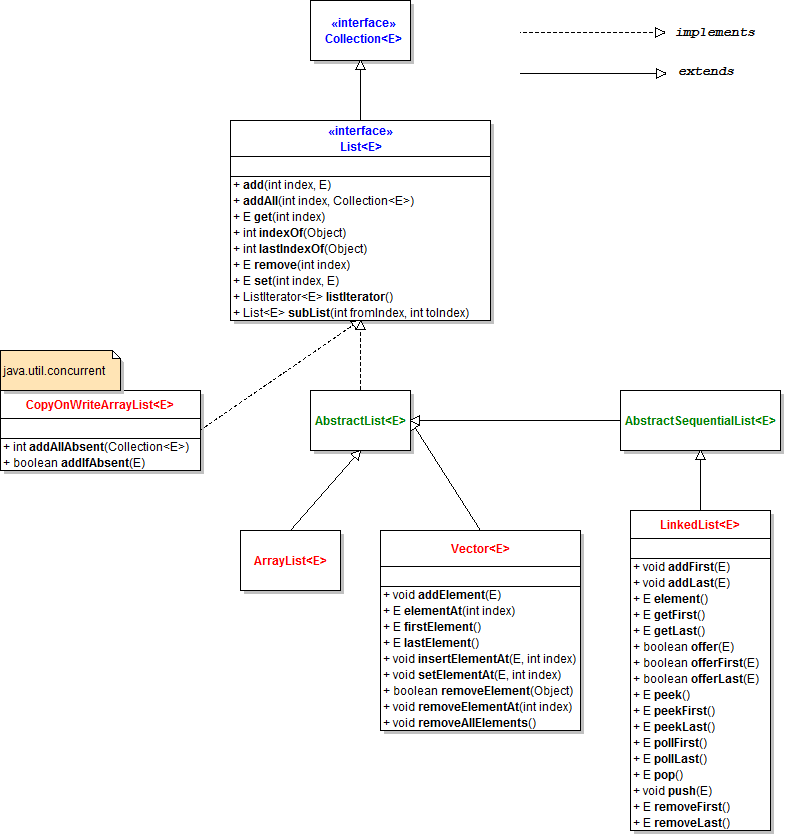
## Java Collection Framework

The following class diagram shows a brief overview of the Java Collections Framework which is divided into four groups**: List, Set, Map** and **Queue**. Only the principal, commonly-used interfaces and classes are listed.



There are also more detailed class diagrams for each group:

#### Class diagram of List API



List<E>is the base interface for all kinds of list. It defines general operations for a List type.

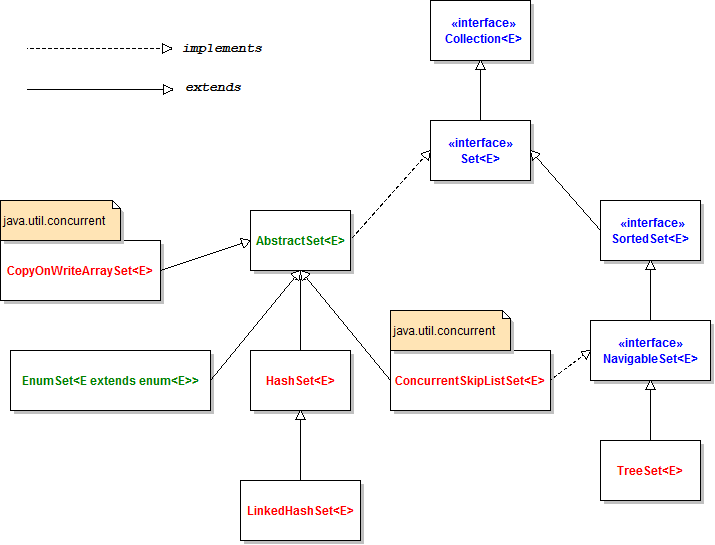
* Abstract subclasses: AbstractList<E> and AbstractSequentialList<E>
* Concrete implementation classes:

ArrayList<E>,Vector<E>,LinkedList<E> and CopyOnWriteArrayList<E>( java.util.concurrent.CopyOnWriteArrayList).

* Legacy collection: Vector<E> (Thread-safe)
* Implementation classes in JDK which are not members of Java Collections Framework:

AttributeList,RoleList,RoleUnresolvedList and Stack(java.utils.Stack).

#### Class Diagram of Set API



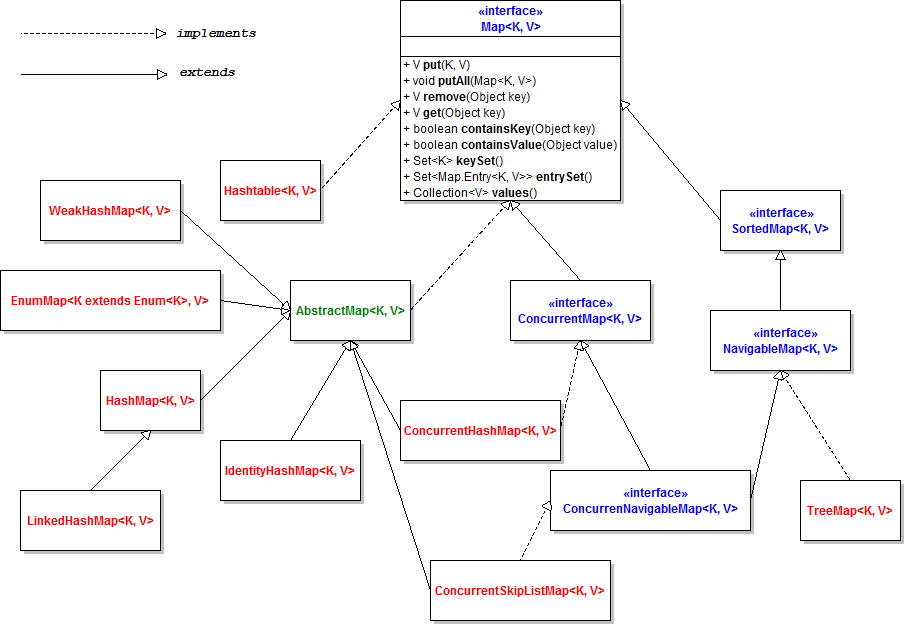
**Set<E>** is the base interface for all kinds of set. This interface extends all methods from the **Collection<E>** interface and does not define any new methods.

* Sub interfaces: **SortedSet<E>** and **NavigableSet<E>.**
* Abstract subclasses: **AbstractSet<E>** and **EnumSet<E extends enum<E>>.**
* Concrete implementation classes:

**HashSet<E>**, **LinkedHashSet<E>**, **TreeSet<E>**,**ConcurrentSkipListSet<E>** and **CopyOnWriteArraySet<E>**(these two last classes are under java.util.concurrent package).

* The **JobStateReasons**class extends **HashSet<E>**but it is not a member of Java Collections Framework.

#### Class Diagram of Map API

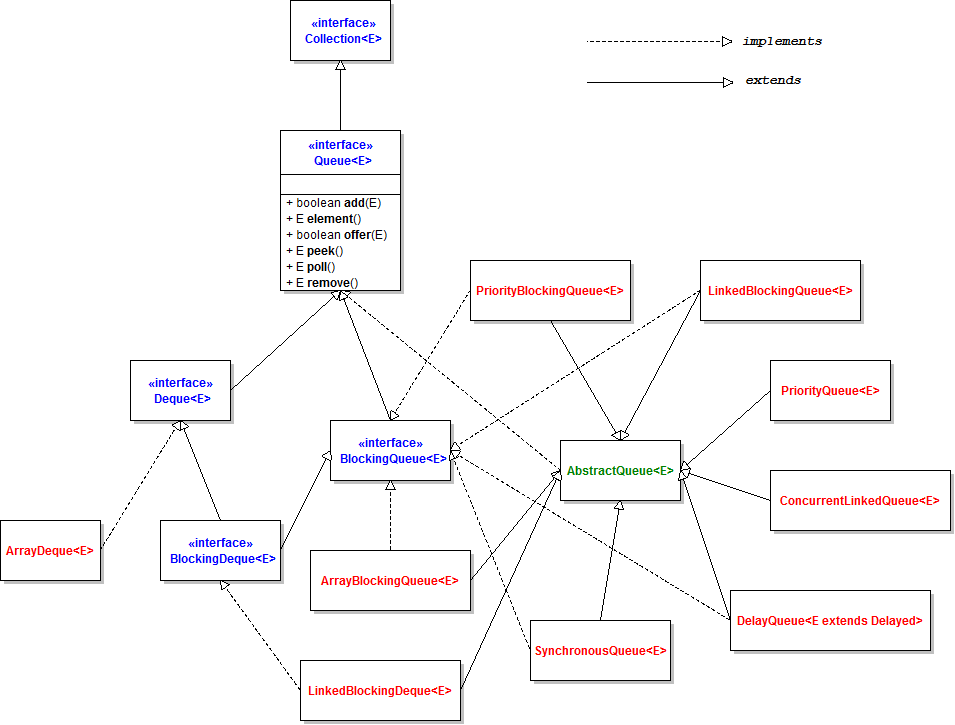


Map<K, V> is the base interface for all kind of maps.

* Sub interfaces: ConcurrentMap<K, V>, SortedMap<K, V>, NavigableMap<K, V> andConcurrentNavigableMap<K, V>.
* Abstract classes: AbstractMap<E>.
* Implementation classes:
  + Hashtable<K, V>
  + HashMap<K, V>
  + LinkedHashMap<K, V>
  + IdentityHashMap<K, V>
  + ConcurrentHashMap<K, V>
  + ConcurrentSkipListMap<K, V>
  + EnumMap<K extends Enum<K>, V>
  + WeakHashMap<K, V>
  + TreeMap<K, V>

#### Class Diagram of Queue API

Queue API is the most complex API in the family of Java Collections Framework. Queue<E> is the base interface for all kind of queues.



The following class diagram outlines the hierarchy of Queue API:

* Sub interfaces: BlockingQueue<E>, Deque<E> and BlockingDeque<E>.
* Abstract classes: AbstractQueue<E>.
* Implementation classes:
  + ArrayBlockingQueue<E>
  + ArrayDeque<E>
  + ConcurrentLinkedQueue<E>
  + DelayQueue<E extends Delayed>
  + LinkedBlockingDeque<E>
  + LinkedBlockingQueue<E>
  + LinkedList<E>
  + PriorityBlockingQueue<E>
  + PriorityQueue<E>
  + SynchronousQueue<E>

## Java Concurrent Framework

#### Overview of java.util.concurrent package in Java™ 7

Several java.util.concurrent.\* packages introduced with version 5.0 of the Java platform added high-level concurrency features to the Java and new concurrent data structures to the Java Collections Framework.

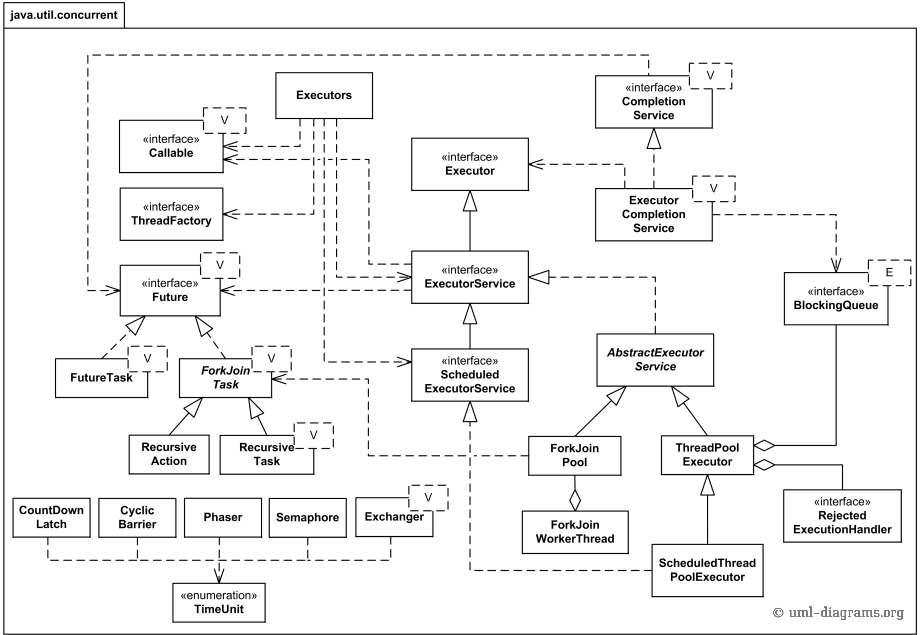


Figure 1 UML class diagram for the Java™ 7 executors and thread pool managers from the java.util.concurrent package.

**Concurrent collections** are also part of the java.util.concurrent package. These collections reduce the need for synchronization and designed to support concurrent access and modifications of the large collections of data.

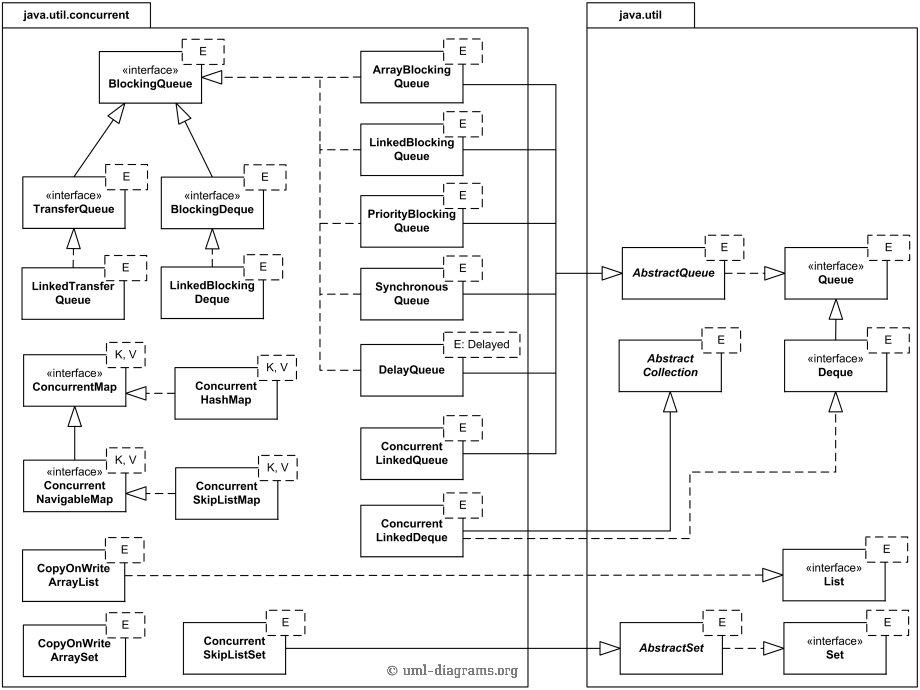
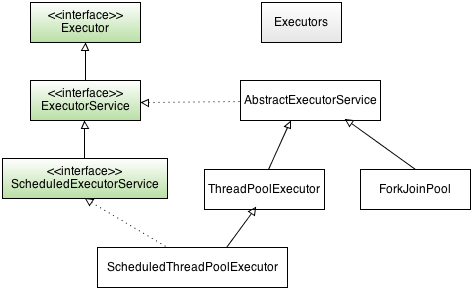


Figure 2 UML class diagram for concurrent collections from the Java 7 java.util.concurrent package

#### *Executor Framework*

Java 5 introduced a new framework to abstract thread management from the rest of the application. These set of API/objects are popularly known as Executors or Executor framework. Managing thread means; controlling its life cycle, its usage, and its scheduling etc. So through Executors you don't need to explicitly create threads. Thread creation and management will be taken care by Executors.

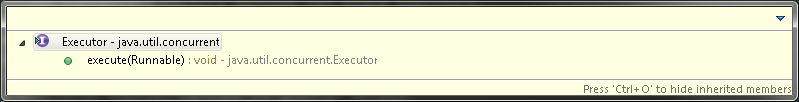
Below class diagram shows major classes which are part of Executor Framework.



1. Executor **interface**

Executor interface is the basis for a flexible and powerful framework for asynchronous task execution that supports a wide variety of task **execution policies**.

Executor is based on the **producer-consumer pattern**, where activities submitting tasks are the producers and the threads that execute tasks are the consumers.

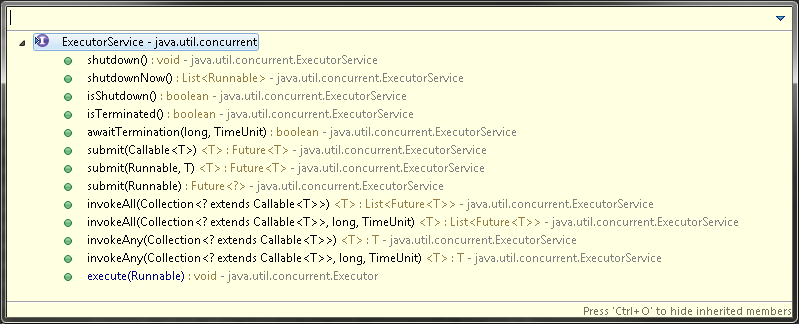


1. ExecutorService **interface**

To address the issue of execution service lifecycle, the ExecutorService interface extends Executor, adding a number of methods for lifecycle management (as well as some convenience methods or task submission).

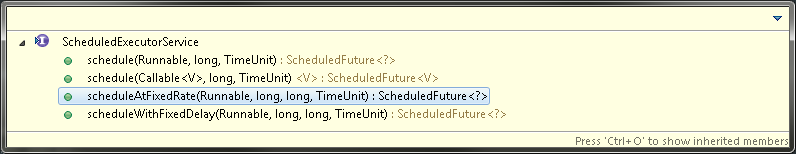
The lifecycle implied by ExecutorService has three states - running, shutting down, and terminated.

* ExecutorService are initially created in the running state;
* The shutdown() method initiates a graceful shutdown: no new tasks are accepted but previously submitted tasks are allowed to complete ‐ including those that have not yet begun execution;
* The shutdownNow() method initiates an abrupt shutdown: it attempts to cancel outstanding tasks and does not start any tasks that are queued but not begun.



1. ScheduledExecutorService **interface**

This interface extends further and adds methods related to scheduling. With this you can run your Runnable/Callable task after a fixed delay. So it can be used to execute an specified task repeatedly at fixed interval.



1. **Factory methods in** Executors

**Executors** define a high-level API for launching and managing threads to support large-scale applications mostly by adding **thread pool management** abilities. The java.util.concurrent package includes several thread pool management implementation classes.

Executors.newFixedThreadPool(10) : This is one of the most common type of thread pool. This type of pool always has a specified number of threads running. If a thread from the pool is terminated due to some reason then it automatically gets replaced with a new thread.

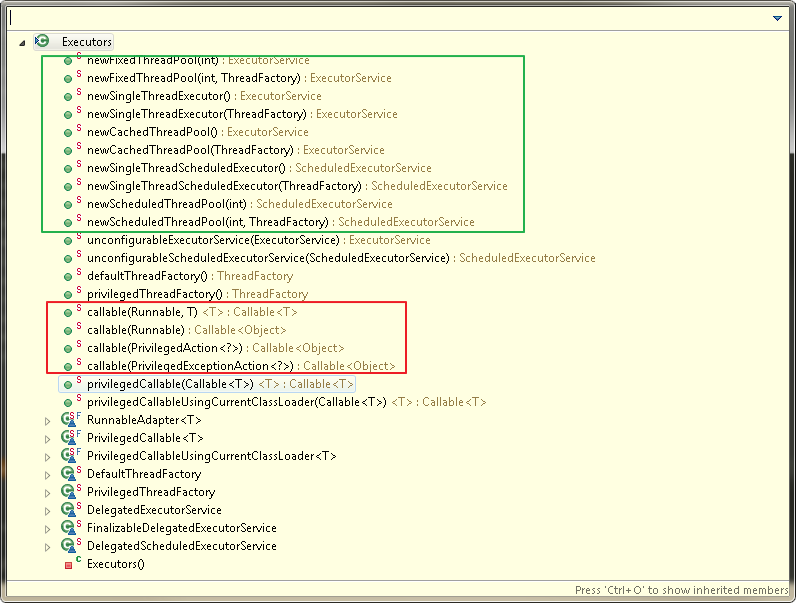
Executors.newCachedThreadPool() : Creates an expandable thread pool which reuses previously created threads if available. These pools will improve the performance of the program that creates many short-lived asynchronous tasks. Threads that have been not used for 60 seconds are terminated and removed from the cache. Thus if pool remains idle for longer time will not consume any resource.

Executors.newSingleThreadExecutor() : Creates a single-threaded executor. It's a single worker thread to process tasks, and replaces it if it dies due to some reason/error. So using this, tasks will be executed sequentially. So if you have a queue having 10 tasks, then tasks will get executed one after another depending on the order.

Executors.newScheduledThreadPool() : Creates a fixed size thread pool that supports delayed and periodic task execution.

|  |  |
| --- | --- |
| newCachedThreadPool() | -缓存型池，先查看池中有没有以前建立的线程，如果有，就 reuse.如果没有，就建一个新的线程加入池中 -缓存型池子通常用于执行一些**生存期很短的异步型任务**  因此在一些面向连接的daemon型SERVER中用得不多。但对于生存期短的异步任务，它是Executor的首选。 -能reuse的线程，必须是timeout IDLE内的池中线程，**缺省 timeout是60s**,超过这个IDLE时长，线程实例将被终止及移出池。 注意，放入CachedThreadPool的线程不必担心其结束，**超过TIMEOUT不活动，其会自动被终止。** |
| newFixedThreadPool(int) | -newFixedThreadPool与cacheThreadPool差不多，也是能reuse就用，但不能随时建新的线程 -其独特之处:任意时间点，最多只能有固定数目的活动线程存在，**此时如果有新的线程要建立，只能放在另外的队列中等待**，**直到当前的线程中某个线程终止直接被移出池子** -和cacheThreadPool不同，FixedThreadPool没有IDLE机制（可能也有，但既然文档没提，肯定非常长，类似依赖上层的TCP或UDP IDLE机制之类的），所以**FixedThreadPool多数针对一些很稳定很固定的正规并发线程，多用于服务器** -从方法的源代码看，cache池和fixed 池调用的是同一个底层池，只不过参数不同: **fixed池线程数固定，并且是0秒IDLE（无IDLE）** **cache池线程数支持0-Integer.MAX\_VALUE(显然完全没考虑主机的资源承受能力），60秒IDLE** |
| newScheduledThreadPool(int) | -调度型线程池 -这个池子里的线程可以按schedule依次delay执行，或周期执行 |
| SingleThreadExecutor() | -单线程，任意时间池中只能有一个线程 -**用的是和cache池和fixed池相同的底层池，但线程数目是1-1,0秒IDLE（无IDLE）** |

Executors also provides dedicated classes for creating thread pools specific to your needs if methods provided through Executors are not enough. These classes are ThreadPoolExecutor, ScheduledThreadPoolExecutor and ForkJoinPool as shown in above class diagram.



#### Future & Callable

Executor框架使用Runnable作为对(Task)任务的最基本抽象，但是Runnable接口无法返回任务执行结果或者抛出受检异常(Checked Exception), 虽然Runnable执行过程可以有side effect,例如生成日志，或者将结果保存到一个用于共享结果的数据结构中。

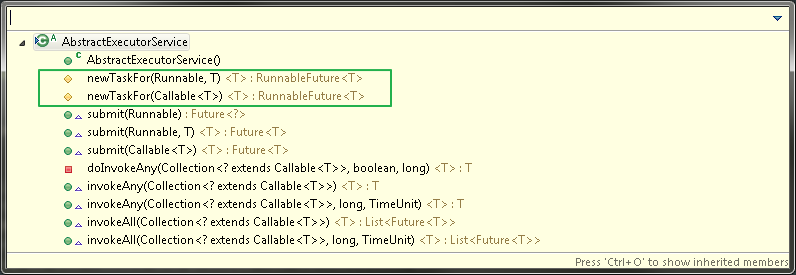
对于需要返回执行结果的任务，Callable接口是一个更好的抽象。Executors类提供一些静态方法用于将Runnable或者PrivileageAction转化为Callable。

Callable和Runnable都是对任务(Task)的抽象，任务作为提交到线程池执行和调度的最小单元。同时，任务具备四种状态created, submitted, started, completed. 对一个任务需要能查询当前的状态，查询运行结果（如果有结果），同时对于正在执行的任务需要能取消（取消一个任务取决于该任务的当前状态，如果还未started,可以直接取消，如果已经执行，需要interrupt()并由线程池处理任务抛出的InterruptedException来取消任务）。对于这部分功能，Java抽象为Future<T>。

The Future<V> interface represents the result of an asynchronous computation, where type V is the result type returned by the Future's get method. Methods of this interface allow waiting for the computation to complete, to cancel execution of the task, to check if the computation is complete or was cancelled, and to retrieve the result of the computation.

Java提供若干方式可以为任务（Runnable或者Callable）创建Future实例，用于查询任务的状态，取消任务，查看结果。

1. ExecutorService中的submit()方法；
2. 对给定的Runnable或Callable创建FutureTask实例，然后提交到Executor中执行；
3. 重写AbstractExecutorService中的newTaskFor()，AbstractExecutorService为ExecutorService的实现类，所有的具体实现类都继承AbstractExecutorService；



newTaskFor()默认实现返回RunnableFuture<T>的实现，默认为FutureTask.

The FutureTask class is an implementation of Future that implements java.lang.Runnable as required by RunnableFuture interface, and thus may be executed by an Executor.

protected <T> RunnableFuture<T> newTaskFor(Callable<T> task) {

return new FutureTask<T>(task);

}

AbstractExecutorService为提交的Runnable或者Callable创建好FutureTask之后直接提交给execute()方法执行，并直接返回该FutureTask对象。

public Future<?> submit(Runnable task) {

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

RunnableFuture<Void> ftask = newTaskFor(task, null);

execute(ftask);

**return ftask;**

}

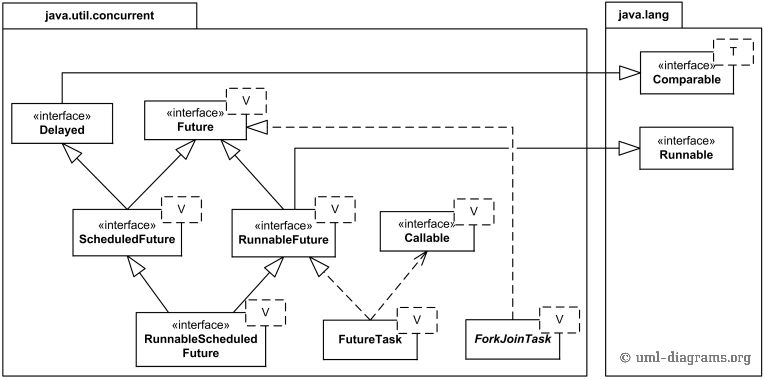


Figure 3 UML class diagram for asynchronous results (futures) from the Java 7 java.util.concurrent package.

The Delayed interface allows marking objects that should be acted upon after a given delay. ScheduledFuture<V> interface extends both Future<V> and Delayed, and is usually a result of scheduling a task with a ScheduledExecutorService.

#### ThreadPoolExecutor

（原文<http://www.infoq.com/cn/articles/executor-framework-thread-pool-task-execution-part-02> ）

##### 一．任务的提交

Executor接口中定义的execute(Runnable command)方法的作用就是执行提交的任务，该方法在抽象类AbstractExecutorService中没有实现，留到子类中实现。我们观察下子类ThreadPoolExecutor，使用最广泛的线程池如何来execute那些submit的任务的。这个方法看着比较简单，但是线程池什么时候创建新的作业线程来处理任务，什么时候只接收任务不创建作业线程，另外什么时候拒绝任务。线程池的接收任务、维护工作线程的策略都要在其中体现。

作为必要的预备知识，先补充下ThreadPoolExecutor有两个最重要的集合属性，分别是存储接收任务的任务队列和用来干活的作业集合。

//任务队列

private final BlockingQueue<Runnable> workQueue;

//作业线程集合

private final HashSet<Worker> workers = new HashSet<Worker>();

其中阻塞队列**workQueue**是来存储待执行的任务的，在构造线程池时可以选择满足该BlockingQueue 接口定义的SynchronousQueue、LinkedBlockingQueue或者DelayedWorkQueue等不同阻塞队列来实现不同特征的线程池。

关注下execute(Runnable command)方法中调用到的addIfUnderCorePoolSize，workQueue.offer(command) ， ensureQueuedTaskHandled(command)，addIfUnderMaximumPoolSize(command)这几个操作。尤其几个名字较长的private方法，把方法名的驼峰式的单词分开，加上对方法上下文的了解就能理解其功能。

因为前面说到的几个方法在里面即是操作，又返回一个布尔值，影响后面的逻辑，所以不大方便在方法体中为每条语句加注释来说明，需要大致关联起来看。所以首先需要把execute方法的主要逻辑说明下，再看其中各自方法的作用。

* 如果线程池的状态是RUNNING，线程池的大小小于配置的核心线程数，说明还可以创建新线程，则启动新的线程执行这个任务。
* 如果线程池的状态是RUNNING ，线程池的大小小于配置的最大线程数，并且任务队列已经满了，说明现有线程已经不能支持当前的任务了，并且线程池还有继续扩充的空间，就可以创建一个新的线程来处理提交的任务。
* 如果线程池的状态是RUNNING，当前线程池的大小大于等于配置的核心线程数，说明根据配置当前的线程数已经够用，不用创建新线程，只需把任务加入任务队列即可。如果任务队列不满，则提交的任务在任务队列中等待处理；如果任务队列满了则需要考虑是否要扩展线程池的容量。
* 当线程池已经关闭或者上面的条件都不能满足时，则进行拒绝策略，拒绝策略在RejectedExecutionHandler接口中定义，可以有多种不同的实现。

上面其实也是对最主要思路的解析，详细展开可能还会更复杂。简单梳理下思路：构建线程池时定义了一个额定大小，当线程池内工作线程数小于额定大小，有新任务进来就创建新工作线程，如果超过该阈值，则一般就不创建了，只是把接收任务加到任务队列里面。但是如果任务队列里的任务实在太多了，那还是要申请额外的工作线程来帮忙。如果还是不够用就拒绝服务。

public void execute(Runnable command) {

if (command == null)

throw new NullPointerException();

if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {

if (runState == RUNNING && workQueue.offer(command)) {

if (runState != RUNNING || poolSize == 0)

ensureQueuedTaskHandled(command);

}

else if (!addIfUnderMaximumPoolSize(command))

reject(command); // is shutdown or saturated

}

}

1. addIfUnderCorePoolSize方法检查如果当前线程池的大小小于配置的核心线程数，说明还可以创建新线程，则启动新的线程执行这个任务。

private boolean addIfUnderCorePoolSize(Runnable firstTask) {

Thread t = null;

//如果当前线程池的大小小于配置的核心线程数，说明还可以创建新线程

if (poolSize < corePoolSize && runState == RUNNING)

// 则启动新的线程执行这个任务

t = addThread(firstTask);

return t != null;

}

1. 和上一个方法类似，addIfUnderMaximumPoolSize检查如果线程池的大小小于配置的最大线程数，并且任务队列已经满了（就是execute方法试图把当前线程加入任务队列时不成功），说明现有线程已经不能支持当前的任务了，但线程池还有继续扩充的空间，就可以创建一个新的线程来处理提交的任务。

private boolean addIfUnderMaximumPoolSize(Runnable firstTask) {

// 如果线程池的大小小于配置的最大线程数，并且任务队列已经满了（就

是execute方法中试图把当前线程加入任务队列workQueue.offer(command)时候不成功

）,说明现有线程已经不能支持当前的任务了，但线程池还有继续扩充的空间

if (poolSize < maximumPoolSize && runState == RUNNING)

//就可以创建一个新的线程来处理提交的任务

t = addThread(firstTask);

return t != null;

}

1. 在ensureQueuedTaskHandled方法中，判断如果当前状态不是RUNING，则当前任务不加入到任务队列中，判断如果状态是停止，线程数小于允许的最大数，且任务队列还不空，则加入一个新的工作线程到线程池来帮助处理还未处理完的任务。

private void ensureQueuedTaskHandled(Runnable command) {

// 如果当前状态不是RUNING，则当前任务不加入到任务队列中，判断如

果状态是停止，线程数小于允许的最大数，且任务队列还不空

if (state < STOP &&

poolSize < Math.max(corePoolSize, 1) &&

!workQueue.isEmpty())

//则加入一个新的工作线程到线程池来帮助处理还未处理完的任务

t = addThread(null);

if (reject)

reject(command);

}

1. 在前面方法中都会调用adThread方法创建一个工作线程，差别是创建的有些工作线程上面关联接收到的任务firstTask，有些没有。该方法为当前接收到的任务firstTask创建Worker，并将Worker添加到作业集合HashSet<Worker> workers中，并启动作业。

private Thread addThread(Runnable firstTask) {

//为当前接收到的任务firstTask创建Worker

Worker w = new Worker(firstTask);

Thread t = threadFactory.newThread(w);

w.thread = t;

//将Worker添加到作业集合HashSet<Worker> workers中，并启动作业

workers.add(w);

t.start();

return t;

}

至此，任务提交过程简单描述完毕，并介绍了任务提交后ExecutorService框架下线程池的主要应对逻辑，其实就是接收任务，根据需要创建或者维护管理线程。

##### 二．任务的执行

ThreadPoolExecutor主要属性时提到其中有一个HashSet<Worker> workers的集合，我们有说明这里存储的就是线程池的工作队列的集合，队列的对象是Worker类型的工作线程，Worker是ThreadPoolExecutor的一个内部类，实现了Runnable接口：

private final class Worker implements Runnable

看作业线程干什么当然是看它的run方法在干什么。如我们所料，作业线程就是在一直调用getTask方法获取任务，然后调用 runTask(task)方法执行任务。看到没有，是在while循环里面，就是不干完不罢休的意思！

public void run() {

try {

Runnable task = firstTask;

//循环从线程池的任务队列获取任务

while (task != null || (task = getTask()) != null) {

//执行任务

runTask(task);

task = null;

}

} finally {

workerDone(this);

}

}

getTask()方法是ThreadPoolExecutor提供给其内部类Worker的方法。作用就是一个，从任务队列中取任务，源源不断地输出任务。有没有想到老大手里拿的总是满满当当的project，也是源源不断的。

Runnable getTask() {

for (;;) {

//从任务队列的头部取任务

r = workQueue.take();

return r;

}

}

runTask(Runnable task)是工作线程Worker真正处理拿到的每个具体任务, 需要注意的地方出现了，调用的其实是task的run()方法。

private void runTask(Runnable task) {

//调用任务的run方法，即在Worker线程中执行Task内定义内容。

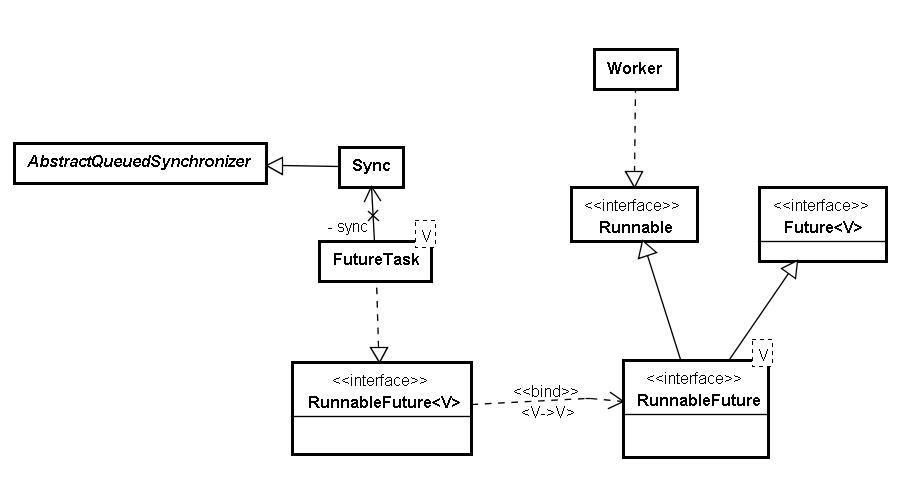
task.run();

}

这里插入一个FutureTask的类图。可以看到FutureTask实现了RunnableFuture接口，所以FutureTask即有Runnable接口的run方法来定义任务内容，也有Future接口中定义的get、cancel等方法来控制任务执行和获取执行结果。Runnable接口自不用说，Future接口的伟大设计，就是使得实现该接口的对象可以阻塞线程直到任务执行完毕，也可以取消任务执行，检测任务是执行完毕还是被取消了。想想在之前我们使用Thread.join()或者Thread.join(long millis)等待任务结束是多么苦涩。

FutureTask<T>的实现，以前基于AbstractQueueSynchronizer

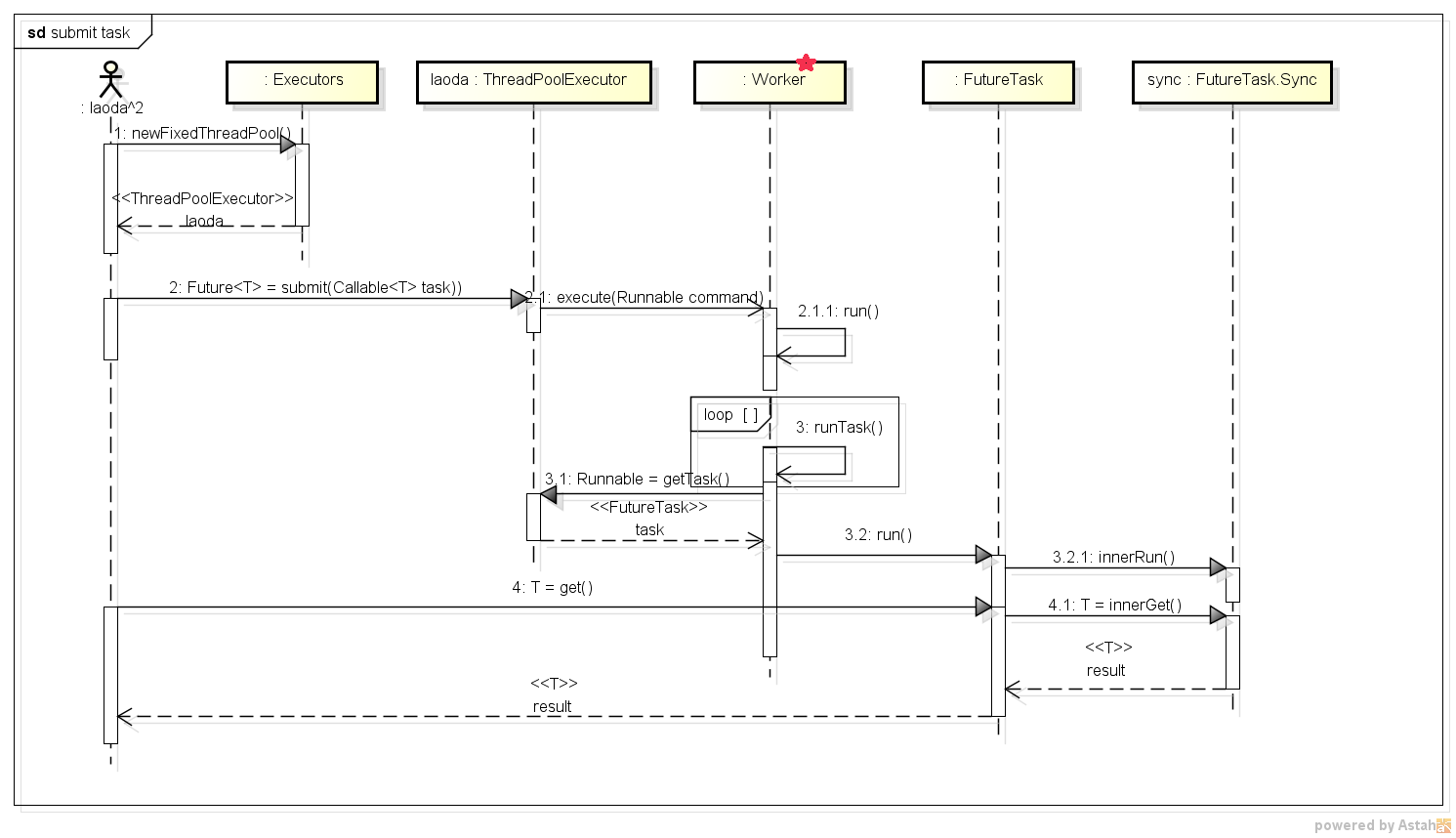
(介绍FutureTask如何利用AbstractQueueSynchronizer实现<http://www.infoq.com/cn/articles/executor-framework-thread-pool-task-execution-part-02> )



根据任务和角色分析如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **角色名** | **任务用户** | **任务管理者** | **任务执行者** |
| **角色属性** | 任务的甲方 | 任务的乙方 | 乙方的工具 |
| **角色说明** | 选择合适的任务执行服务，如可以根据需要选择ThreadPoolExecutor还是ScheduledThreadPoolExecutor，并定制ExecutorService的配置。 定义好任务的工作内容和结果类型，提交任务，等待任务的执行结果 | 接收提交的任务； 维护执行服务内部管理； 配置工作线程执行任务 | 每个工作线程一直从任务执行服务获取待执行的任务，保证任务完成后返回执行结果。 |
| **Executor中对应** | 创建获取ExecutorService、并提交Task的外部接口 | ExecutorService的各种实现。如经典的ThreadPoolExecutor，ScheduledThreadPoolExecutor | 执行服务内定义的配套的Worker线程。如ThreadPoolExecutor.Worker |
| **主要接口方法** | submit(Callable task) | execute(Runnable command) | runTask(Runnable task) |
| **现实角色映射** | 手里有活的大老大 | 领人干活的老大 | 真正干活的码农 |
| **主要工作伪代码** | taskService = createService() future=taskService.submitTask() future.get() | executeTask() { addTask() createThread() } | while(ture) { getTask() runTask() } |

1. 外面需要提交任务的角色（如例子中老大的老大），首先创建一个任务执行服务ExecutorService，一般使用工具类Executors的若干个工厂方法 创建不同特征的线程池ThreadPoolExecutor，例子中是使用newFixedThreadPool方法创建有n个固定工作线程的线程池。
2. 线程池是专门负责从外面接活的老大。把任务封装成一个FutureTask对象，并根据输入定义好要获得结果的类型，就可以submit任务了。
3. 线程池就像我们团队里管人管项目的老大，各个都有一套娴熟、有效的办法来对付输入的任务和手下干活的兄弟一样，内部有一套比较完整、细致的任务管理办法，工作线程管理办法，以便应付输入的任务。这些逻辑全部在其execute方法中体现。
4. 线程池接收输入的task，根据需要创建工作线程，启动工作线程来执行task。
5. 工作线程在其run方法中一直循环，从线程池领取可以执行的task，调用task的run方法执行task内定义的任务。
6. FutureTask的run方法中调用其内部类Sync的innerRun方法来执行封装的具体任务，并把任务的执行结果返回给FutureTask的result变量。
7. 当提及任务的角色调用FutureTask的get方法获取执行结果时，Sync的innerGet方法被调用。根据任务的执行状态判断，任务执行完毕则返回执行结果；未执行完毕则等待。



#### 线程安全集合(Thread safe Collection)

Java 5.0 improves on the synchronized collections by providing several concurrent collection classes. Synchronized collections achieve their thread safety by serializing all access to the collection's state. The cost of this approach is poor concurrency; when multiple threads contend for the collection‐wide lock, throughput suffers.

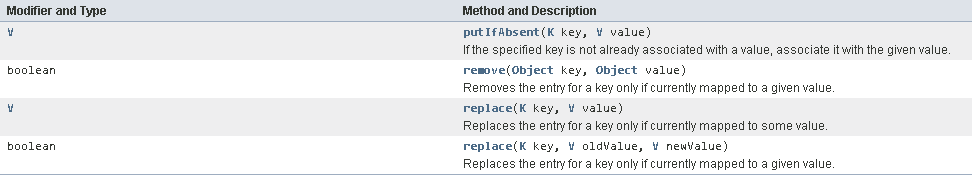
The concurrent collections, on the other hand, are designed for concurrent access from multiple threads. *[5.2.* ***Concurrent Collections*** *in “Java in Concurrency”]*

##### 接口ConcurrentMap<K,V>和ConcurrentHashMap<K, V>

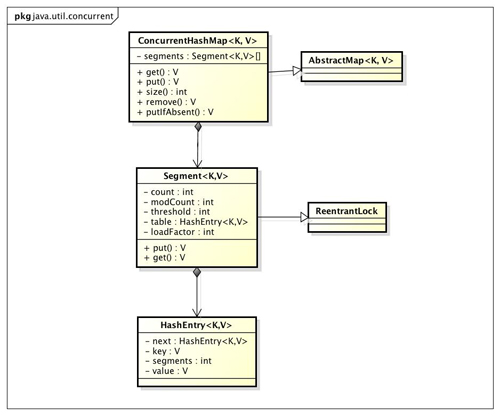
Implements Map<K,V>, and adds support for common compound actions such as put‐if‐absent, replace, and conditional remove.

**These compound operations such as put-if-absent, remove-if-equals and replace-if-equals are implemented as atomic operations and specified by ConcurrentMap interface.**

**Implementer Class**：ConcurrentHashMap<K,V>.



ConcurrentHashMap<K, V>继承了AbstractHashMap<K,V>抽象类，并实现了ConcurrentMap<K,V>接口。



###### 结构模型

ConcurrentHashMap 类中包含两个静态内部类 HashEntry 和 Segment。HashEntry 用来封装映射表的键 / 值对；Segment 是一种可重入锁ReentrantLock，用来充当锁的角色，每个 Segment 对象守护整个散列映射表的若干个桶。每个桶是由若干个 HashEntry 对象链接起来的链表。一个 ConcurrentHashMap 实例中包含由若干个 Segment 对象组成的数组。或者说，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。

HashEntry类

HashEntry 用来封装散列映射表中的键值对。在 HashEntry 类中，key，hash 和 next 域都被声明为 final 型，value 域被声明为 volatile 型。

**清单 1.HashEntry 类的定义**

static final class HashEntry<K,V> {

final K key; // 声明 key 为 final 型

final int hash; // 声明 hash 值为 final 型

volatile V value; // 声明 value 为 volatile 型

final HashEntry<K,V> next; // 声明 next 为 final 型

HashEntry(K key, int hash, HashEntry<K,V> next, V value) {

this.key = key;

this.hash = hash;

this.next = next;

this.value = value;

}

}

在 ConcurrentHashMap 中，在散列时如果产生“碰撞”，将采用“分离链接法”来处理“碰撞”：把“碰撞”的 HashEntry 对象链接成一个链表。由于 HashEntry 的 next 域为 final 型，所以新节点只能在链表的表头处插入。 下图是在一个空桶中依次插入 A，B，C 三个 HashEntry 对象后的结构图：

**图 1. 插入三个节点后桶的结构示意图：**

图 1. 插入三个节点后桶的结构示意图：

注意：由于只能在表头插入，所以链表中节点的顺序和插入的顺序相反。

Segment类

Segment 类继承于 ReentrantLock 类，从而使得 Segment 对象能充当锁的角色。每个 Segment 对象用来守护其（成员对象 table 中）包含的若干个桶。

table 是一个由 HashEntry 对象组成的数组。table 数组的每一个数组成员就是散列映射表的一个桶。

count 变量是一个计数器，它表示每个 Segment 对象管理的 table 数组（若干个 HashEntry 组成的链表）包含的 HashEntry 对象的个数。每一个 Segment 对象都有一个 count 对象来表示本 Segment 中包含的 HashEntry 对象的总数。注意，之所以在每个 Segment 对象中包含一个计数器，而不是在 ConcurrentHashMap 中使用全局的计数器，是为了避免出现“热点域”而影响ConcurrentHashMap 的并发性。

**清单 2.Segment 类的定义**

static final class Segment<K,V> extends ReentrantLock implements Serializable {

/\*\*

\* 在本 segment 范围内，包含的 HashEntry 元素的个数

\* 该变量被声明为 volatile 型

\*/

transient volatile int count;

/\*\*

\* table 被更新的次数

\*/

transient int modCount;

/\*\*

\* 当 table 中包含的 HashEntry 元素的个数超过本变量值时，触发 table 的再散列

\*/

transient int threshold;

/\*\*

\* table 是由 HashEntry 对象组成的数组

\* 如果散列时发生碰撞，碰撞的 HashEntry 对象就以链表的形式链接成一个链表

\* table 数组的数组成员代表散列映射表的一个桶

\* 每个 table 守护整个 ConcurrentHashMap 包含桶总数的一部分

\* 如果并发级别为 16，table 则守护 ConcurrentHashMap 包含的桶总数的 1/16

\*/

transient volatile HashEntry<K,V>[] table;

/\*\*

\* 装载因子

\*/

final float loadFactor;

Segment(int initialCapacity, float lf) {

loadFactor = lf;

setTable(HashEntry.<K,V>newArray(initialCapacity));

}

/\*\*

\* 设置 table 引用到这个新生成的 HashEntry 数组

\* 只能在持有锁或构造函数中调用本方法

\*/

void setTable(HashEntry<K,V>[] newTable) {

// 计算临界阀值为新数组的长度与装载因子的乘积

threshold = (int)(newTable.length \* loadFactor);

table = newTable;

}

/\*\*

\* 根据 key 的散列值，找到 table 中对应的那个桶（table 数组的某个数组成员）

\*/

HashEntry<K,V> getFirst(int hash) {

HashEntry<K,V>[] tab = table;

// 把散列值与 table 数组长度减 1 的值相“与”，

// 得到散列值对应的 table 数组的下标

// 然后返回 table 数组中此下标对应的 HashEntry 元素

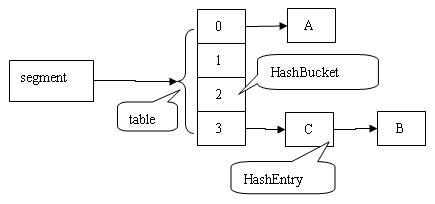
return tab[hash & (tab.length - 1)];

}

}

下图是依次插入 ABC 三个 HashEntry 节点后，Segment 的结构示意图。

**图 2. 插入三个节点后** Segment **的结构示意图：**



ConcurrentHashMap类

ConcurrentHashMap 在默认并发级别会创建包含 16 个 Segment 对象的数组。每个 Segment 的成员对象 table 包含若干个散列表的桶。每个桶是由 HashEntry 链接起来的一个链表。如果键能均匀散列，每个 Segment 大约守护整个散列表中桶总数的 1/16。

**清单 3.ConcurrentHashMap 类的定义**

public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>

implements ConcurrentMap<K, V>, Serializable {

/\*\*

\* 散列映射表的默认初始容量为 16，即初始默认为 16 个桶

\* 在构造函数中没有指定这个参数时，使用本参数

\*/

static final int DEFAULT\_INITIAL\_CAPACITY= 16;

/\*\*

\* 散列映射表的默认装载因子为 0.75，该值是 table 中包含的 HashEntry 元素的个数与

\* table 数组长度的比值

\* 当 table 中包含的 HashEntry 元素的个数超过了 table 数组的长度与装载因子的乘积时，

\* 将触发 再散列

\* 在构造函数中没有指定这个参数时，使用本参数

\*/

static final float DEFAULT\_LOAD\_FACTOR= 0.75f;

/\*\*

\* 散列表的默认并发级别为 16。该值表示当前更新线程的估计数

\* 在构造函数中没有指定这个参数时，使用本参数

\*/

static final int DEFAULT\_CONCURRENCY\_LEVEL= 16;

/\*\*

\* segments 的掩码值

\* key 的散列码的高位用来选择具体的 segment

\*/

final int segmentMask;

/\*\*

\* 偏移量

\*/

final int segmentShift;

/\*\*

\* 由 Segment 对象组成的数组

\*/

final Segment<K,V>[] segments;

/\*\*

\* 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。

\*/

public ConcurrentHashMap(int initialCapacity,

float loadFactor, int concurrencyLevel) {

if(!(loadFactor > 0) || initialCapacity < 0 ||

concurrencyLevel <= 0)

throw new IllegalArgumentException();

if(concurrencyLevel > MAX\_SEGMENTS)

concurrencyLevel = MAX\_SEGMENTS;

// 寻找最佳匹配参数（不小于给定参数的最接近的 2 次幂）

int sshift = 0;

int ssize = 1;

while(ssize < concurrencyLevel) {

++sshift;

ssize <<= 1;

}

segmentShift = 32 - sshift; // 偏移量值

segmentMask = ssize - 1; // 掩码值

this.segments = Segment.newArray(ssize); // 创建数组

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

int c = initialCapacity / ssize;

if(c \* ssize < initialCapacity)

++c;

int cap = 1;

while(cap < c)

cap <<= 1;

// 依次遍历每个数组元素

for(int i = 0; i < this.segments.length; ++i)

// 初始化每个数组元素引用的 Segment 对象

this.segments[i] = new Segment<K,V>(cap, loadFactor);

}

/\*\*

\* 创建一个带有默认初始容量 (16)、默认加载因子 (0.75) 和 默认并发级别 (16)

\* 的空散列映射表。

\*/

public ConcurrentHashMap() {

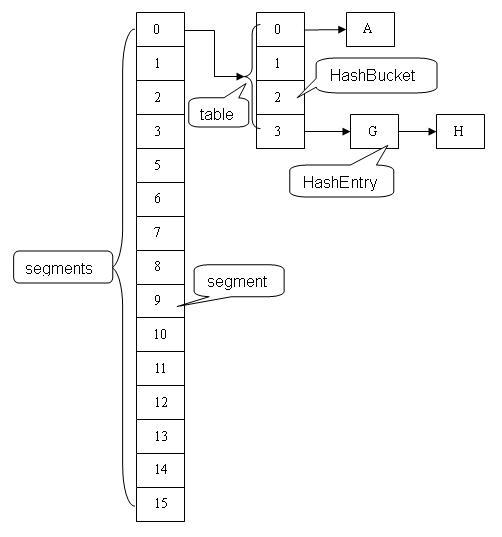
// 使用三个默认参数，调用上面重载的构造函数来创建空散列映射表

this(DEFAULT\_INITIAL\_CAPACITY, DEFAULT\_LOAD\_FACTOR, DEFAULT\_CONCURRENCY\_LEVEL);

}

}

图 3.ConcurrentHashMap 的结构示意图：



###### ConcurrentHashMap的并发总结

用分离锁实现多个线程间的并发写操作（put方法）

在 ConcurrentHashMap 中，线程对映射表做读操作时，一般情况下不需要加锁就可以完成，对容器做结构性修改的操作才需要加锁。下面以 put 操作为例说明对 ConcurrentHashMap 做结构性修改的过程。

由于put方法里需要对共享变量进行写入操作，所以为了线程安全，在操作共享变量时必须得加锁。Put方法首先定位到Segment，然后在Segment里进行插入操作。插入操作需要经历两个步骤，第一步判断是否需要对Segment里的HashEntry数组进行扩容，第二步定位添加元素的位置然后放在HashEntry数组里。

首先，根据 key 计算出对应的 hash 值：

清单 4.Put 方法的实现

public V put(K key, V value) {

if (value == null) //ConcurrentHashMap 中不允许用 null 作为映射值

throw new NullPointerException();

int hash = hash(key.hashCode()); // 计算键对应的散列码

// 根据散列码找到对应的 Segment

return segmentFor(hash).put(key, hash, value, false);

}

然后，根据 hash 值找到对应的Segment 对象：

清单 5.根据 hash 值找到对应的 Segment

/\*\*

\* 使用 key 的散列码来得到 segments 数组中对应的 Segment

\*/

final Segment<K,V> segmentFor(int hash) {

// 将散列值右移 segmentShift 个位，并在高位填充 0

// 然后把得到的值与 segmentMask 相“与”

// 从而得到 hash 值对应的 segments 数组的下标值

// 最后根据下标值返回散列码对应的 Segment 对象

return segments[(hash >>> segmentShift) & segmentMask];

}

最后，在这个 Segment 中执行具体的 put 操作：

1. 是否需要扩容。在插入元素前会先判断Segment里的HashEntry数组是否超过容量（threshold），如果超过阀值，数组进行扩容。值得一提的是，Segment的扩容判断比HashMap更恰当，因为HashMap是在插入元素后判断元素是否已经到达容量的，如果到达了就进行扩容，但是很有可能扩容之后没有新元素插入，这时HashMap就进行了一次无效的扩容。
2. 如何扩容。扩容的时候首先会创建一个两倍于原容量的数组，然后将原数组里的元素进行再hash后插入到新的数组里。为了高效ConcurrentHashMap不会对整个容器进行扩容，而只对某个segment进行扩容。

清单 6.在 Segment 中执行具体的 put 操作

V put(K key, int hash, V value, boolean onlyIfAbsent) {

lock(); // 加锁，这里是锁定某个 Segment 对象而非整个 ConcurrentHashMap

try {

int c = count;

if (c++ > threshold) // 如果超过再散列的阈值

rehash(); // 执行再散列，table 数组的长度将扩充一倍

HashEntry<K,V>[] tab = table;

// 把散列码值与 table 数组的长度减 1 的值相“与”

// 得到该散列码对应的 table 数组的下标值

int index = hash & (tab.length - 1);

// 找到散列码对应的具体的那个桶

HashEntry<K,V> first = tab[index];

HashEntry<K,V> e = first;

while (e != null && (e.hash != hash || !key.equals(e.key)))

e = e.next;

V oldValue;

if (e != null) { // 如果键 / 值对以经存在

oldValue = e.value;

if (!onlyIfAbsent)

e.value = value; // 设置 value 值

}

else { // 键 / 值对不存在

oldValue = null;

++modCount; // 要添加新节点到链表中，所以 modCont 要加 1

// 创建新节点，并添加到链表的头部

tab[index] = new HashEntry<K,V>(key, hash, first, value);

count = c; // 写 count 变量

}

return oldValue;

} finally {

unlock(); // 解锁

}

}

**注意：**这里的加锁操作是针对（键的 hash 值对应的）某个具体的 Segment，锁定的是该 Segment 而不是整个 ConcurrentHashMap。因为插入键 / 值对操作只是在这个 Segment 包含的某个桶中完成，不需要锁定整个ConcurrentHashMap。此时，其他写线程对另外 15 个Segment 的加锁并不会因为当前线程对这个 Segment 的加锁而阻塞。同时，所有读线程几乎不会因本线程的加锁而阻塞（除非读线程刚好读到这个 Segment 中某个 HashEntry 的 value 域的值为 null，此时需要加锁后重新读取该值）。

相比较于 HashTable 和由同步包装器包装的 HashMap每次只能有一个线程执行读或写操作，ConcurrentHashMap 在并发访问性能上有了质的提高。在理想状态下，ConcurrentHashMap 可以支持 16 个线程执行并发写操作（如果并发级别设置为 16），及任意数量线程的读操作。

用 HashEntry 对象的不变性来降低读操作对加锁的需求

在代码清单“HashEntry 类的定义”中我们可以看到，HashEntry 中的 key，hash，next 都声明为 final 型。这意味着，不能把节点添加到链接的中间和尾部，也不能在链接的中间和尾部删除节点。这个特性可以保证：在访问某个节点时，这个节点之后的链接不会被改变。这个特性可以大大降低处理链表时的复杂性。

同时，HashEntry 类的 value 域被声明为 Volatile 型，Java 的内存模型可以保证：某个写线程对 value 域的写入马上可以被后续的某个读线程“看”到。在 ConcurrentHashMap 中，不允许用 unll 作为键和值，当读线程读到某个 HashEntry 的 value 域的值为 null 时，便知道产生了冲突——发生了重排序现象，需要加锁后重新读入这个 value 值。这些特性互相配合，使得读线程即使在不加锁状态下，也能正确访问 ConcurrentHashMap。

下面我们分别来分析线程写入的两种情形：对散列表做非结构性修改的操作和对散列表做结构性修改的操作。

1. 非结构性修改操作只是更改某个 HashEntry 的 value 域的值。由于对 Volatile 变量的写入操作将与随后对这个变量的读操作进行同步。当一个写线程修改了某个 HashEntry 的 value 域后，另一个读线程读这个值域，Java 内存模型能够保证读线程读取的一定是更新后的值。所以，写线程对链表的非结构性修改能够被后续不加锁的读线程“看到”。
2. 对 ConcurrentHashMap 做结构性修改，实质上是对某个桶指向的链表做结构性修改。如果能够确保：在读线程遍历一个链表期间，写线程对这个链表所做的结构性修改不影响读线程继续正常遍历这个链表。那么读 / 写线程之间就可以安全并发访问这个 ConcurrentHashMap。

结构性修改操作包括 put，remove，clear。下面我们分别分析这三个操作。

clear 操作只是把 ConcurrentHashMap 中所有的桶“置空”，每个桶之前引用的链表依然存在，只是桶不再引用到这些链表（所有链表的结构并没有被修改）。正在遍历某个链表的读线程依然可以正常执行对该链表的遍历。

从上面的代码清单“在 Segment 中执行具体的 put 操作”中，我们可以看出：put 操作如果需要插入一个新节点到链表中时 , 会在链表头部插入这个新节点。此时，链表中的原有节点的链接并没有被修改。也就是说：插入新健 / 值对到链表中的操作不会影响读线程正常遍历这个链表。

下面来分析 remove 操作，先让我们来看看 remove 操作的源代码实现:

清单 7.remove 操作

V remove(Object key, int hash, Object value) {

lock(); // 加锁

try{

int c = count - 1;

HashEntry<K,V>[] tab = table;

// 根据散列码找到 table 的下标值

int index = hash & (tab.length - 1);

// 找到散列码对应的那个桶

HashEntry<K,V> first = tab[index];

HashEntry<K,V> e = first;

while(e != null&& (e.hash != hash || !key.equals(e.key)))

e = e.next;

V oldValue = null;

if(e != null) {

V v = e.value;

if(value == null|| value.equals(v)) { // 找到要删除的节点

oldValue = v;

++modCount;

// 所有处于待删除节点之后的节点原样保留在链表中

// 所有处于待删除节点之前的节点被克隆到新链表中

HashEntry<K,V> newFirst = e.next;// 待删节点的后继结点

for(HashEntry<K,V> p = first; p != e; p = p.next)

newFirst = new HashEntry<K,V>(p.key, p.hash,

newFirst, p.value);

// 把桶链接到新的头结点

// 新的头结点是原链表中，删除节点之前的那个节点

tab[index] = newFirst;

count = c; // 写 count 变量

}

}

return oldValue;

} finally{

unlock(); // 解锁

}

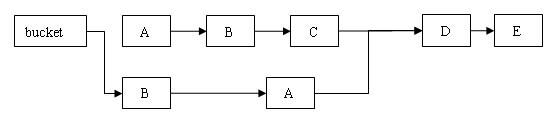
}

和get操作一样，首先根据散列码找到具体的链表；然后遍历这个链表找到要删除的节点；最后把待删除节点之后的所有节点原样保留在新链表中，把待删除节点之前的每个节点克隆到新链表中。下面通过图例来说明remove操作。假设写线程执行 remove 操作，要删除链表的 C 节点，另一个读线程同时正在遍历这个链表。

图 4. 执行删除之前的原链表：

图 4. 执行删除之前的原链表：

图 5. 执行删除之后的新链表



从上图可以看出，删除节点 C 之后的所有节点原样保留到新链表中；删除节点 C 之前的每个节点被克隆到新链表中，注意：它们在新链表中的链接顺序被反转了。

在执行 remove 操作时，原始链表并没有被修改，也就是说：读线程不会受同时执行 remove 操作的并发写线程的干扰。

综合上面的分析我们可以看出，写线程对某个链表的结构性修改不会影响其他的并发读线程对这个链表的遍历访问。

用 Volatile 变量协调读写线程间的内存可见性(get方法)

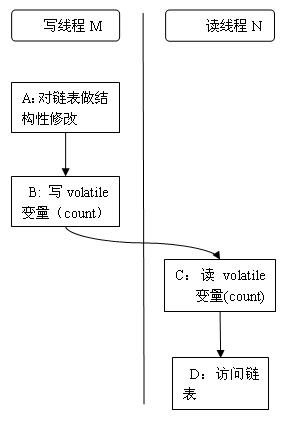
由于内存可见性问题，未正确同步的情况下，写线程写入的值可能并不为后续的读线程可见。以ConcurrentHashMap中的get方法为例，下面以写线程 M 和读线程 N 来说明 ConcurrentHashMap 如何协调读 / 写线程间的内存可见性问题。

假设线程 M 在写入了 volatile 型变量 count 后，线程 N 读取了这个 volatile 型变量 count。

1. 根据 happens-before 关系法则中的程序次序法则，A appens-before 于 B，C happens-before D。
2. 根据 Volatile 变量法则，B happens-before C。
3. 根据传递性，连接上面三个 happens-before 关系得到：A appens-before 于 B； B appens-before C；C happens-before D。也就是说：写线程 M 对链表做的结构性修改，在读线程 N 读取了同一个 volatile 变量后，对线程 N 也是可见的了。

虽然线程 N 是在未加锁的情况下访问链表。Java 的内存模型可以保证：只要之前对链表做结构性修改操作的写线程 M 在退出写方法前写 volatile 型变量 count，读线程 N 在读取这个 volatile 型变量 count 后，就一定能“看到”这些修改。

图 6. 协调读 - 写线程间的内存可见性的示意图：



ConcurrentHashMap 中，每个 Segment 都有一个变量 count。它用来统计 Segment 中的 HashEntry 的个数。这个变量被声明为 volatile。Segment的get操作实现非常简单和高效。先经过一次再哈希，然后使用这个哈希值通过哈希运算定位到segment，再通过哈希算法定位到元素。

清单 8.Count 变量的声明

transient volatile int count;

所有不加锁读方法，在进入读方法时，首先都会去读这个 count 变量。比如下面的 get 方法：

清单 9.get 操作

V get(Object key, int hash) {

if(count != 0) { // 首先读 count 变量

HashEntry<K,V> e = getFirst(hash);

while(e != null) {

if(e.hash == hash && key.equals(e.key)) {

V v = e.value;

if(v != null)

return v;

// 如果读到 value 域为 null，说明发生了重排序，加锁后重新读取

return readValueUnderLock(e);

}

e = e.next;

}

}

return null;

}

在 ConcurrentHashMap 中，所有执行写操作的方法（put, remove, clear），在对链表做结构性修改之后，在退出写方法前都会去写这个 count 变量。所有未加锁的读操作（get, contains, containsKey）在读方法中，都会首先去读取这个 count 变量。

根据 Java 内存模型，对 同一个 volatile 变量的写 / 读操作可以确保：写线程写入的值，能够被之后未加锁的读线程“看到”。

这个特性和前面介绍的 HashEntry 对象的不变性相结合，使得在 ConcurrentHashMap 中，读线程在读取散列表时，基本不需要加锁就能成功获得需要的值。这两个特性相配合，不仅减少了请求同一个锁的频率（读操作一般不需要加锁就能够成功获得值），也减少了持有同一个锁的时间（只有读到 value 域的值为 null 时 , 读线程才需要加锁后重读）。

ConcurrentHashMap的size操作

如果我们要统计整个ConcurrentHashMap里元素的大小，就必须统计所有Segment里元素的大小后求和。Segment里的全局变量count是一个volatile变量，那么在多线程场景下，我们是不是直接把所有Segment的count相加就可以得到整个ConcurrentHashMap大小了呢？不是的，虽然相加时可以获取每个Segment的count的最新值，但是拿到之后可能累加前使用的count发生了变化，那么统计结果就不准了。所以最安全的做法，是在统计size的时候把所有Segment的put，remove和clean方法全部锁住，但是这种做法显然非常低效。 因为在累加count操作过程中，之前累加过的count发生变化的几率非常小，所以ConcurrentHashMap的做法是先尝试2次通过不锁住Segment的方式来统计各个Segment大小，如果统计的过程中，容器的count发生了变化，则再采用加锁的方式来统计所有Segment的大小。

那么ConcurrentHashMap是如何判断在统计的时候容器是否发生了变化呢？使用modCount变量，在put , remove和clean方法里操作元素前都会将变量modCount进行加1，那么在统计size前后比较modCount是否发生变化，从而得知容器的大小是否发生变化。

###### ConcurrentHashMap 实现高并发的总结

基于通常情形而优化

在实际的应用中，散列表一般的应用场景是：除了少数插入操作和删除操作外，绝大多数都是读取操作，而且读操作在大多数时候都是成功的。正是基于这个前提，ConcurrentHashMap 针对读操作做了大量的优化。通过 HashEntry 对象的不变性和用 volatile 型变量协调线程间的内存可见性，使得 大多数时候，读操作不需要加锁就可以正确获得值。这个特性使得 ConcurrentHashMap 的并发性能在分离锁的基础上又有了近一步的提高。

总结

ConcurrentHashMap 是一个并发散列映射表的实现，它允许完全并发的读取，并且支持给定数量的并发更新。相比于 HashTable 和用同步包装器包装的 HashMap(Collections.synchronizedMap(new HashMap()))，ConcurrentHashMap 拥有更高的并发性。在 HashTable 和由同步包装器包装的 HashMap 中，使用一个全局的锁来同步不同线程间的并发访问。同一时间点，只能有一个线程持有锁，也就是说在同一时间点，只能有一个线程能访问容器。这虽然保证多线程间的安全并发访问，但同时也导致对容器的访问变成串行化的了。

**在使用锁来协调多线程间并发访问的模式下，减小对锁的竞争可以有效提高并发性。有两种方式可以减小对锁的竞争：**

* 减小请求 同一个锁的 频率。
* 减少持有锁的 时间。

ConcurrentHashMap **的高并发性主要来自于三个方面：**

1. 用分离锁实现多个线程间的更深层次的共享访问。

使用分离锁，减小了请求 同一个锁的频率。

1. 用 HashEntery 对象的不变性来降低执行读操作的线程在遍历链表期间对加锁的需求。

通过 HashEntery 对象的不变性及对同一个 Volatile 变量的读 / 写来协调内存可见性，使得 读操作大多数时候不需要加锁就能成功获取到需要的值。由于散列映射表在实际应用中大多数操作都是成功的 读操作，所以 2 和 3 既可以减少请求同一个锁的频率，也可以有效减少持有锁的时间。

1. 通过对同一个 Volatile 变量的写 / 读访问，协调不同线程间 读/写操作的内存可见性。

通过减小请求同一个锁的频率和尽量减少持有锁的时间 ，使得ConcurrentHashMap的并发性相对于HashTable和用同步包装器包装的 HashMap有了质的提高。

**[参考]**

[探索 ConcurrentHashMap 高并发性的实现机制](http://www.ibm.com/developerworks/cn/java/java-lo-concurrenthashmap/#ibm-pcon)

[聊聊并发（四）——深入分析](http://www.infoq.com/cn/articles/ConcurrentHashMap)

##### CopyOnWriteArrayList

CopyOnWriteArrayList是一个线程安全，并且在读操作时无锁的ArrayList。一个有效的不变对象正确的发布之后，之后读取该对象不需要额外的同步操作。CopyOnWriteArrayList通过不变对象的这一性质获得线程安全，其内部创建一个数组，并且每次修改都会重新创建一个数组。在写操作之间通过ReentrantLock实现互斥访问。

add(E)

通过ReentrantLock实现写互斥访问，同时创建一个大小为size+1的数组，将之前的内容复制到新数组中，并将新添加元素存入末尾，最后做引用切换，指向新创建的对象数组；

remove(E)

同样通过ReentrantLock实现互斥修改，创建一个大小为size-1的数组，对就数组每一个元素，如果equals，则将之后的元素复制到新数组中最后做引用切换， 如果不equals则复制到新数组中，对最后一个元素稍有不同，如果不equals，直接返回false，否则直接引用切换；

get(int)

直接返回对应的index的元素，但有可能读到脏数据，但对于读多写少的情况可以容忍；

iterator()

只对当前的数组的快照遍历，不会抛出ConcurrentModificationException异常；

##### CopyOnWriteArraySet

CopyOnWriteArraySet基于CopyOnWriteArrayList的实现，只是在add时候需要调用addIfAbsent()检查元素时候已经存在， 每次add都需要遍历数组，如果不存在需要创建一个大小为size+1的新数组然后copy原有数组到新数组中，性能比CopyOnWriteArrayList慢；

##### BlockingQueue

Blocking queues provide blocking **put and take** methods as well as the timed equivalents **offer and poll**. If the queue is full, put blocks until space becomes available; if the queue is empty, take blocks until an element is available. Queues can be bounded or unbounded; unbounded queues are never full, so a put on an unbounded queue never blocks. [【5.3. Blocking Queues and the Producerconsumer Pattern】](../EBooks_IT_pdf/java/Java%20Concurrency%20in%20Practice.pdf)

BlockingQueue很好的诠释了Producer-Consumer模式，Java的Executor Task Execution Framework便基于ThreadPool和BlockingQueue。

BlockingQueue的实现：

a) LinkedBlockingQueue and ArrayBlockingQueue

are FIFO queues, analogous to LinkedList and ArrayList but with better concurrent performance than a synchronized List.

b) PriorityBlockingQueue

PriorityBlockingQueue is a priority‐ordered queue, which is useful when you want to process elements in an order other than FIFO. Just like other sorted collections, PriorityBlockingQueue can compare elements according to their natural order (if they implement Comparable) or using a Comparator.

c) SynchronousQueue

A SynchronousQueue has no storage capacity, put and take will block unless another thread is already waiting to participate in the handoff. Synchronous queues are generally suitable only when there are enough consumers that there nearly always will be one ready to take the handoff.

##### Deque

Java 6 also adds another two collection types, Deque (pronounced "deck") and BlockingDeque, that extend Queue and BlockingQueue. A Deque is a double‐ended queue that allows efficient insertion and removal from both the head and the tail. Implementations include ArrayDeque , LinkedList for Deque and LinkedBlockingDeque for BlockingQueue.

Just as blocking queues lend themselves to the producer‐consumer pattern, deques lend themselves to a related pattern called **work stealing**.

A producer‐consumer design has one shared work queue for all consumers; in a **work**

**stealing design**, every consumer has its own deque. If a consumer exhausts the work in its own deque, it can steal work from the tail of someone else's deque. Work stealing can be more scalable than a traditional producer‐consumer design because workers don't contend for a shared work queue; most of the time they access only their own deque, reducing contention.

**When a worker has to access another's queue, it does so from the tail rather than the head, further reducing contention.**

**Work stealing is well suited to problems in which consumers are also producers.** When a worker identifies a new unit of work, it places it at the end of its own deque (or alternatively, in a work sharing design, on that of another worker); when its deque is empty, it looks for work at the end of someone else's deque, ensuring that each worker stays busy.

#### 同步器(Synchronizers)

A synchronizer is any object that coordinates the control flow of threads based on its state. Blocking queues can act as synchronizers. **Blocking queues are unique among the collections classes:** not only do they act as containers for objects, but they can also coordinate the control flow of producer and consumer threads because take and put block until the queue enters the desired state (not empty or not full).

Other types of synchronizers include **semaphores**, **barriers**, and **latches**.

所有的synchronizers都具有相似的性质：内部封装的状态值来决定到达的线程应该等待还是继续，有一组方法改变这些状态，并且有一组方法可以让线程选择等待synchronizer进入某个状态。

###### Latches

###### Barriers

###### Semaphores

###### FutureTask

#### AQS (AbstractQueuedSynchronizer)

参照参考Doug Lea大师的原著

[The java.util.concurrent Synchronizer Framework](http://gee.cs.oswego.edu/dl/papers/aqs.pdf)

源码剖析AQS在几个同步工具类中的使用

<http://www.idouba.net/sync-implementation-by-aqs/>

深度解析Java 8：JDK1.8 AbstractQueuedSynchronizer的实现分析

<http://www.infoq.com/cn/articles/jdk1.8-abstractqueuedsynchronizer> （上）

<http://www.infoq.com/cn/articles/java8-abstractqueuedsynchronizer> （下）