淘宝网 Taobao.com 淘我喜欢！

Java 常见疑惑和陷阱

逾轮

2010/11/3

Topic

- Java基础的常见陷阱
- 集合框架的系统梳理
- 揭开神秘的锁机制
- 窥视Java并发包（JUC）
- 一些学习体会

Java基础的常见陷阱

- 不一样的数字的宽类型和窄类型
- 令人崩溃的字符串常量池和subString()
- 不正常的finally和null
- equals()也不容易
- …

Java基础的常见陷阱

- 发生在我们身边的事

```
StringBuffer clienetCookieList = new StringBuffer("<h1>Client</h1> :  </br>");
Set<Entry<String, String>> cookieSet = TaobaoSession.getCookiesPool().entrySet();
while (cookieSet.iterator().hasNext()) {
        Entry entry = cookieSet.iterator().next();
        clienetCookieList.append(entry.getKey() + " " + entry.getValue() + "</br>");
}
```

```
for(Entry<String,String> e:TaobaoSession.getCookiesPool().entrySet())
        clienetCookieList.append(e.getKey() + " " + e.getValue() + "</br>");
```

●常规问题采用常规的方式处理
●不确定问题可以增加一些特殊/特定的条件(比如while循环中增加一些强制退出机制)

Java基础的常见陷阱

- 诡异的数字

```
System.out.println(12345+5432l);
```

66666?

```
List l = new ArrayList<String>();
l.add("Foo");
System.out.println(1);

0x100000000L + 0xcafebabe = ?
```

```
Long num = 0x111111111L;
```

●变量名称永远不要用l
●数字结尾永远不要用l，表示long使用L

Java基础的常见陷阱

- 诡异的数字

```
for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++) {
        if (b == 0x99)
                System.out.print("Found it!");
}
```

- byte与int进行比较时，会进行有符号扩展
- 窄类型与宽类型比较时特别需要注意符号扩展

```
b == (byte)0x99
(b&0xFF)==0x99
```

Java基础的常见陷阱

• 不经意的规则

X = x +1;  x+=1;

```
byte x = 10;
x += 1234;      // x?
x = x + 1234;  // x?

Object taobao= “taobao”;
String  ali = “ali”;
taobao = taobao+ali; // taobao?
taobao += ali;            // taobao?
```

- *复合赋值表达式自动地将它们所执行的计算的结果转型为其左侧变量的类型*
- 赋值操作数据越界导致编译错误。
- 复合赋值只能是基本类型，基本类型的包装类型以及String，String的复合操作左侧必须是String类型。
- **使用原则是：宽类型向低类型赋值一定不要用复合操作。**

Java基础的常见陷阱

• 重新认识字符串

“A”+”B”  ‘A’+’B’

```
new StringBuffer().append(‘A’).append(‘B’)
new StringBuffer(‘A’).append(‘B’)
new StringBuffer(“A”).append(“B”)
String s = "who";
System.out.println("who" == s);
System.out.println("who" == "who");
System.out.println("who" == new String("who"));
System.out.println("who" == new String("who").intern());
```

- 不要指望==
- 不要指望常量池，尽量不用intern()
- 不要往常量池扔过多东西，会导致持久代OOM

Java基础的常见陷阱

• 重新认识字符串

subString()陷阱

```
public String substring(int beginIndex, int endIndex) {
        return ((beginIndex == 0) && (endIndex == count)) ? this :
            new String(offset + beginIndex, endIndex - beginIndex, value);
}
String(int offset, int count, char value[]) {
        this.value = value;
        this.offset = offset;
        this.count = count;
}
```

- **如果只是普通的比较，匹配，传参，临时变量等，可以直接使用subString()**
- **如果需要成为常驻内存对象，需要包装下new String(s.subString(from,to))**

Java基础的常见陷阱

• 关于类的潜规则

```
public class StrungOut {
    public static void main(String[] args) {
        String s = new String("Hello world");
        System.out.println(s);
    }
}
class String {
    private final java.lang.String s;
    public String(java.lang.String s) {
        this.s = s;
    }
    public java.lang.String toString() {
        return s;
    }
}
```

- 规则1：永远不要命名为java.lang下的类名
- 规则2：包名不能命名为java/javax/java.lang/java.util/javax.sql等开头
- 规则3：永远不能类命名完全一致而实现不一致
- 规则4：尽可能的避免相同名称，尽量不使用默认包

Java基础的常见陷阱

- 关于类的潜规则

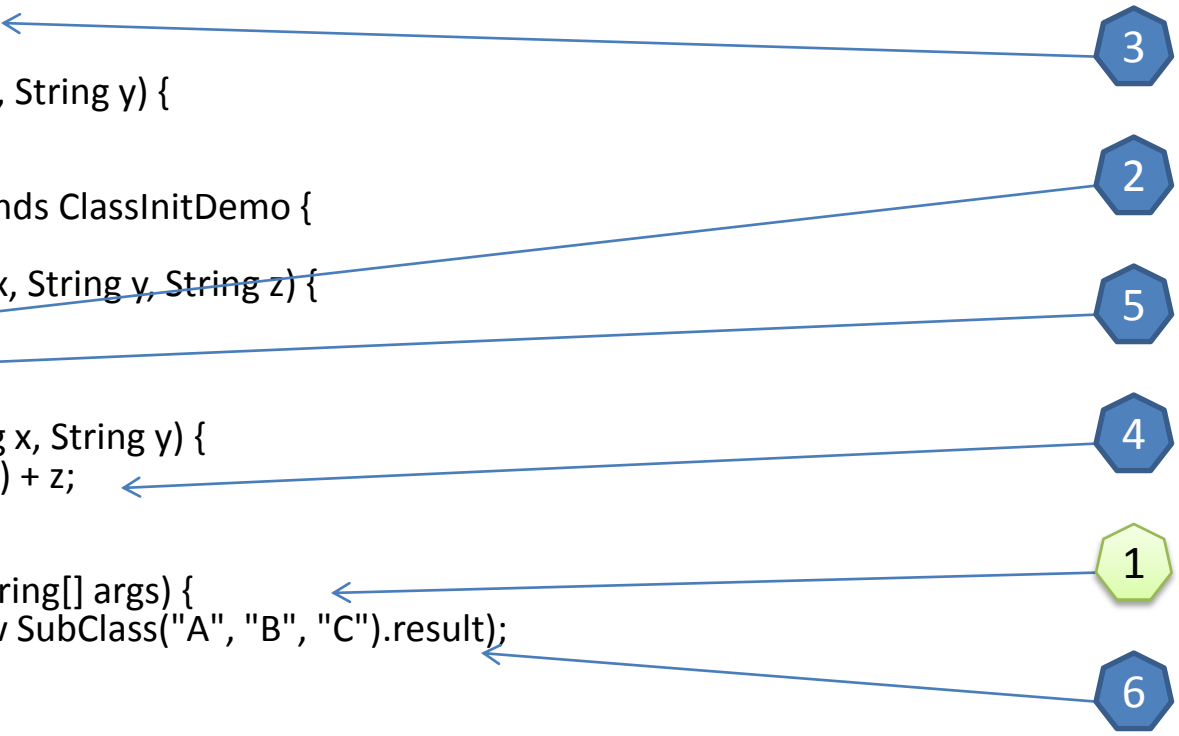
```
public class ILoveTaobao {
    public static void main(String[] args) {
        System.out.print("I love ");
        http://www.taobao.com
        System.out.println("taobao!");
    }
}
```

- 非常正常的输出了：I love taobao!
- 永远不要用Label特性（多亏没有goto）

Java基础的常见陷阱

- 类的初始化

```
public class ClassInitDemo {
    final String result;
    public ClassInitDemo(String x, String y) {
        this.result = add(x, y);
    }
    public String add(String x, String y) {
        return x + y;
    }
    static class SubClass extends ClassInitDemo {
        String z;
        public SubClass(String x, String y, String z) {
            super(x, y);
            this.z = z;
        }
        public String add(String x, String y) {
            return super.add(x, y) + z;
        }
    }
    public static void main(String[] args) {
        System.out.println(new SubClass("A", "B", "C").result);
    }
}
```



Java基础的常见陷阱

- 不正常的Finally

```
public class FinallyDemo{
    static int value = 0;
    static int inc() {return value++;}
    static int dec() {return value--;}
    static int getResult() {
        try {
            return inc();
        } finally {
            return dec();
        }
    }
    public static void main(String[] args) {
        System.out.println(getResult());
        System.out.println(value);
    }
}
```

●原则：finally里面不允许有return/break/continue/throw等改变正常退出的逻辑。

Java基础的常见陷阱

- 能调用null的方法么？

```
public class Null {
    public static void greet() {
        System.out.println("Hello world!");
    }
    public static void main(String[] args) {
        Null x = null;
        x.greet();              //(1)
        ((Null)x).greet();      //(2)
        ((Null) null).greet(); //(3)
    }
}
```

●能够输出“Hello world!”么？？？

Java基础的常见陷阱

- equals到底是什么东东

```
public class Name {
   private String first, last;
   public Name(String first, String last) {
      this.first = first;
      this.last = last;
   }
   public boolean equals(Object o) {
      Name n = (Name)o;
      return n.first.equals(first) && n.last.equals(last);
   }
   public static void main(String[] args) {
      Set s = new HashSet();
      s.add(new Name("Mickey", "Mouse"));
      System.out.println(s.contains(new Name("Mickey", "Mouse")));
   }
}
```

- True or False?
- Set/HashSet换成List/ArrayList或者Map/HashMap又会怎样呢？

Java基础的常见陷阱

- equals到底是什么东东

➢Equals特性

- ●自反性：对于任意引用x，x.equals(x) ==true。
- ●对称性：对于任意引用x,y，x.equals(y) == y.equals(x)
- ●传递性：对于任意引用x,y,z，如果x.equals(y)==y.equals(z)==true那么，x.equals(z)==true
- ●一致性：对于任意引用x,y，多次调用x.equals(y)应该返回相同的值
- ●非空性：对于任意非空引用x，x.equals(null)==false

最佳实践：equals()和hashCode()总是成对出现。

Java基础的常见陷阱

- override & hidden

```
class Parent {
    public String name = "Parent";
}
class Child extends Parent {
    private String name = "Child";
}
public class PrivateMatter {
    public static void main(String[ ] args) {
        System.out.println(new Child().name);
    }
}
```

- override针对实例方法
- hidden针对静态方法、属性、内部类
- override父类方法不能被调用，除非在子类内部使用super
- hidden父类属性、静态方法等可以通过强制类型转换被调用

Java基础的常见陷阱

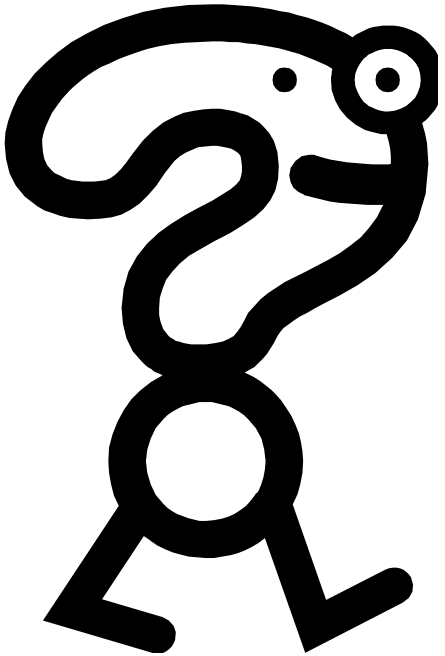
- override & hidden

overload	overwrite	override	hidden
●同一个类 ●针对方法 ●相同名称 ●不同签名	●继承关系 ●针对方法 ●相同名称 ●不同签名	●继承关系 ●针对方法 ●相同名称 ●相同签名	●继承关系 ●针对属性、内部类、静态方法 ●属性和类名称相同 ●静态方法签名相同

不过瘾？去研究下遮蔽（shadow）和遮掩（obscure）吧！

我的疑惑

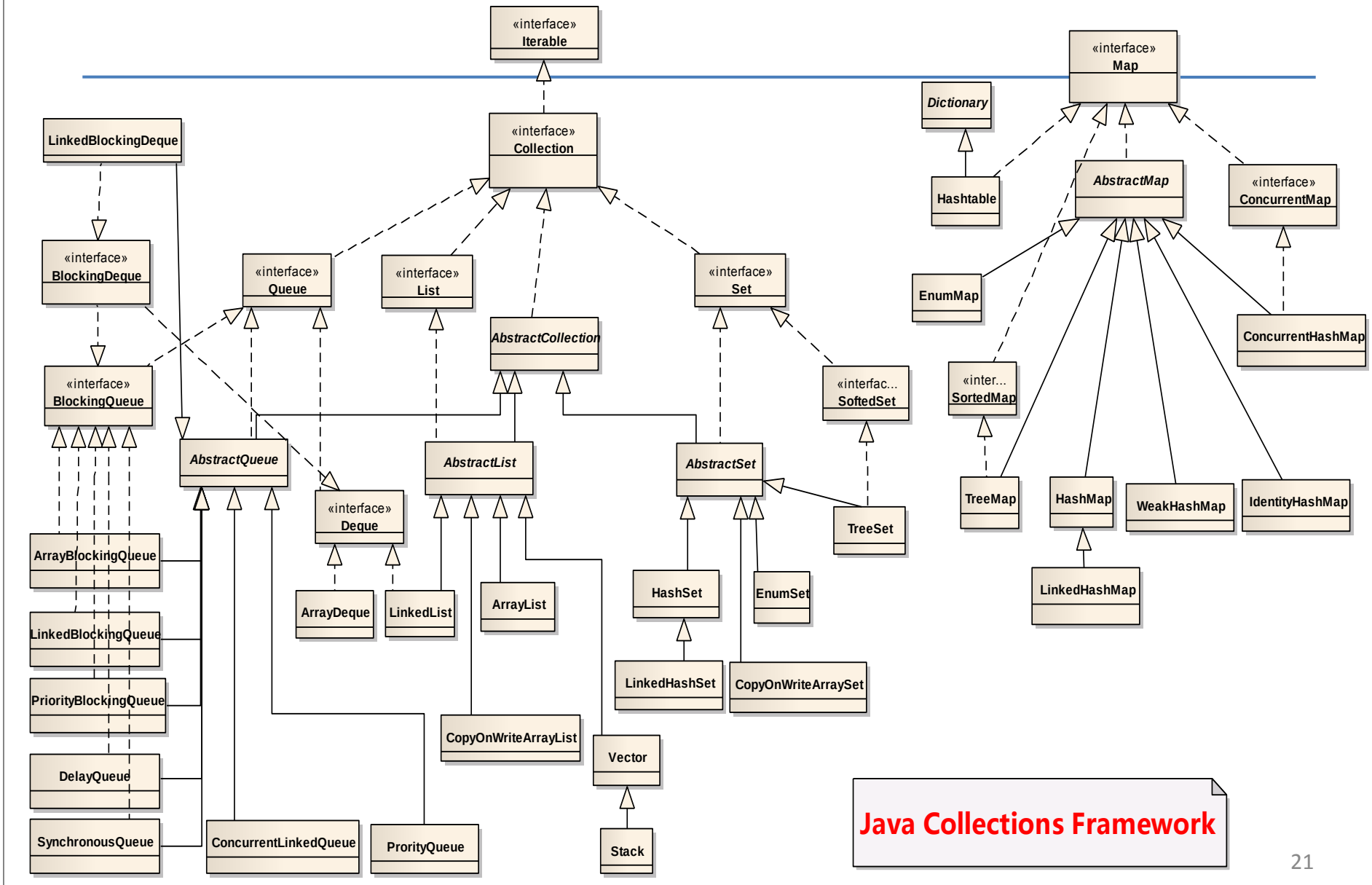
- 集合
  - List/Set/Map/ConcurrentMap
- 锁
  - synchronized/volatile/lock
- 线程池
  - ThreadPool/Timer/Future



集合框架的系统梳理

- JCF概览
- ArrayList/HashMap原理
- JCF 常见疑惑和陷阱

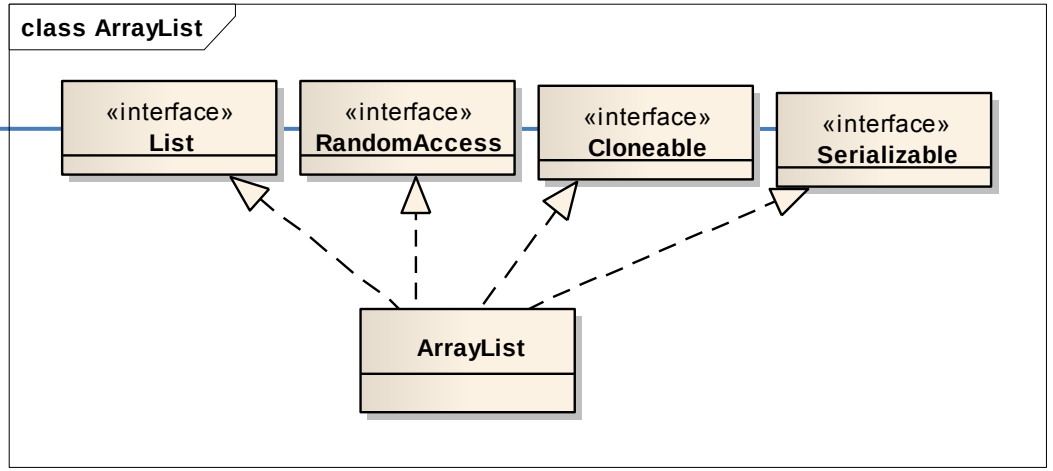
class JCF



Java Collections Framework

ArrayList原理

- 特性
  - 容量可变集合
  - 随机访问
  - 可克隆
  - 可序列化
- 原理
  - 数组
  - 1.5倍扩容
  - modCount特性



ArrayList注意点

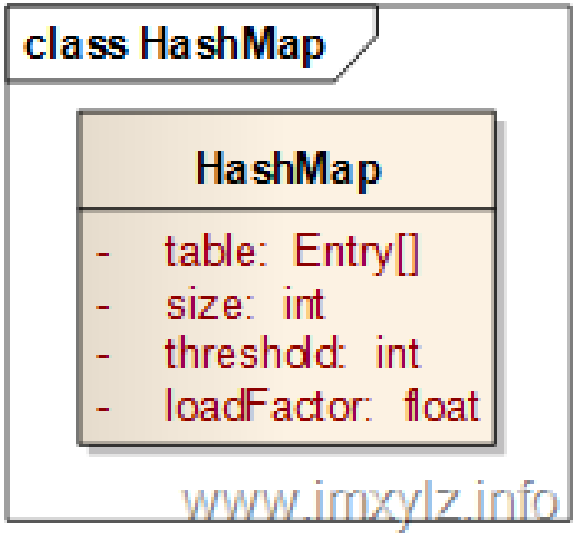
- 静态引用防止内存泄露：clear()/trimToSize()
- 循环注意并发修改
- 带索引迭代：list.listIterator()
- 循环删除方式

```
for(Iterator<String> it =
list.iterator();it.hasNext();) {
    String item = it.next();
    if("2".equals(item))
        it.remove();
}
```

```
for(int i=list.size()-1;i>0;i--) {
    if("3".equals(list.get(i)))
        list.remove(i);
}
```

HashMap原理

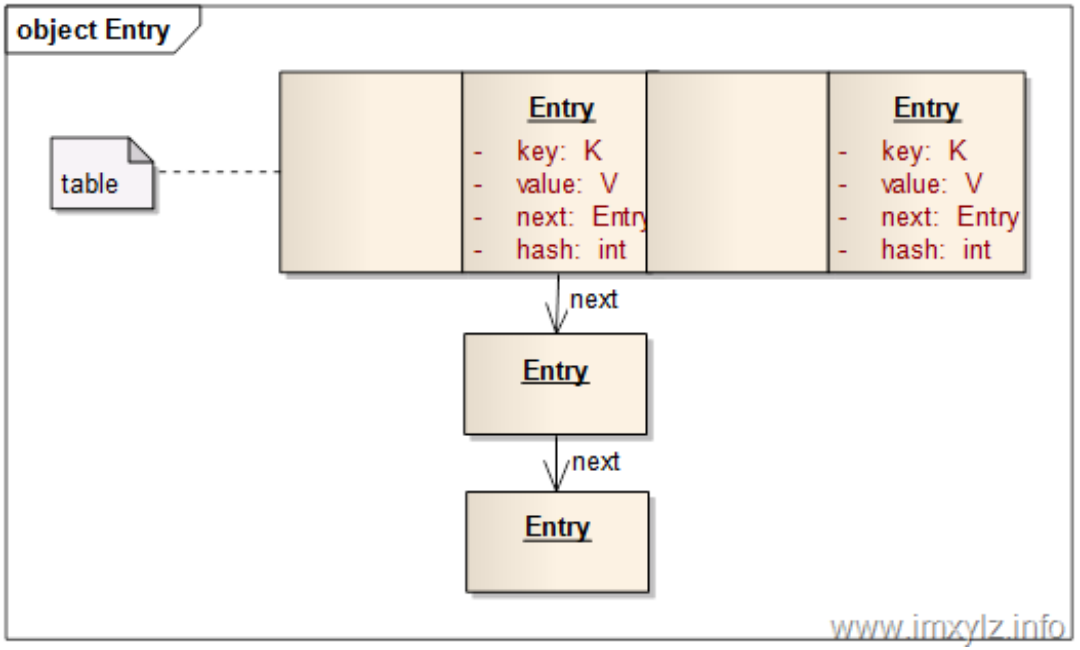
- Map的特性
  - 任意Key编码
  - 快速定位元素
  - 自动扩充容量
- HashMap实现
  - hashCode()
  - indexFor()
  - resize()/rehash()



http://www.blogjava.net/xylz/archive/2010/07/20/326584.html

HashMap原理

- “碰撞”问题
  - hashCode()
  - equals()




```
static int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

HashMap注意点

- 静态引用防止内存泄露：clear()/capacity
- 循环并发修改
- 循环使用entrySet()

JCF 常见疑惑和陷阱

- 框架过于庞大、复杂怎么办？
- “内存泄露”到底该怎么办？
- 如果进行扩展、性能优化？

揭开神秘的锁机制

- 指令重排序与volatile
- 原子操作与CAS
- 可重入锁与读写锁
- 条件变量与线程挂起、唤醒

指令重排序与Volatile

- 线程安全
  - 当多个线程访问一个类时，如果不用考虑这些线程在运行时环境下的调度和交替运行，并且不需要额外的同步及在调用方代码不必做其他的协调，这个类的行为仍然是正确的，那么这个类就是线程安全的。
- 指令重排序
  - JVM能够根据处理器的特性（CPU的多级缓存系统、多核处理器等）适当的重新排序机器指令，使机器指令更符合CPU的执行特点，最大限度的发挥机器的性能。

指令重排序

```
public class ReorderingDemo {
   static int x = 0, y = 0, a = 0, b = 0;
   public static void main(String[] args) throws Exception {
      Thread one = new Thread() {
         public void run() {
            a = 1;
            x = b;
         }
      };
      Thread two = new Thread() {
            public void run() {
               b = 1;
               y = a;
            }
      };
      one.start();
      two.start();
      one.join();
      two.join();
      System.out.println(x + " " + y);
   }
}
```

Happens-before法则

1. 同一个线程中的每个Action都happens-before于出现在其后的任何一个Action。
2. 对一个监视器的解锁happens-before于每一个后续对同一个监视器的加锁。
3. 对volatile字段的写入操作happens-before于每一个后续的同一个字段的读操作。
4. Thread.start()的调用会happens-before于启动线程里面的动作。
5. Thread中的所有动作都happens-before于其他线程检查到此线程结束或者Thread.join（ ）中返回或者Thread.isAlive()==false。
6. 一个线程A调用另一个另一个线程B的interrupt（ ）都happens-before于线程A发现B被A中断（B抛出异常或者A检测到B的isInterrupted（ ）或者interrupted()）。
7. 一个对象构造函数的结束happens-before与该对象的finalizer的开始
8. 如果A动作happens-before于B动作，而B动作happens-before与C动作，那么A动作happens-before于C动作。

http://www.blogjava.net/xylz/archive/2010/07/03/325168.html

volatile语义

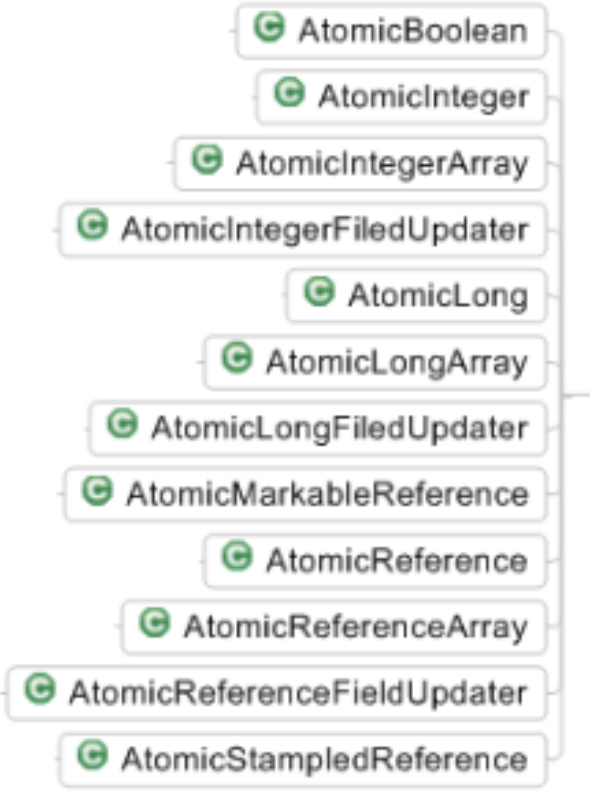
- synchronized的弱实现
- Java 存储模型不会对valatile指令的操作进行重排序：这个保证对volatile变量的操作时按照指令的出现顺序执行的。
- volatile变量的修改，其它线程总是可见的，并且不是使用自己线程栈内部的变量。
- 非线程安全

指令重排序与volatile案例

```
1  public class DoubleLockSingleton {
2
3      private static volatile DoubleLockSingleton instance = null;
4
5      private DoubleLockSingleton() {}
6
7      public static DoubleLockSingleton getInstance() {
8          if (instance == null) {
9              synchronized (DoubleLockSingleton.class) {
10                 if (instance == null) {
11                     instance = new DoubleLockSingleton();
12                 }
13             }
14         }
15         return instance;
16     }
17 }
```

原子操作与CAS

- 原子操作
  - 多个线程执行一个操作时，其中任何一个线程要么完全执行完此操作，要么没有执行此操作的任何步骤，那么这个操作就是原子的。



CAS操作

- 非阻塞算法(**nonblocking algorithms**)
  - 一个线程的失败或者挂起不应该影响其他线程的失败或挂起。
- CAS(Compare and Swap)
  - CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

CAS原理

- boolean compareAndSet(int expect, int update)

```
while(false == compareAndSet(expect,update)){
    expect = getCurrent();
}
```

```
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```

可重入锁与读写锁

- ReentrantLock
  - lock/tryLock
  - synchronized
- ReadWriteLock
  - readLock()
  - writeLock()

可重入锁与读写锁

- lock()

```
while(synchronization state does not allow acquire){
    enqueue current thread if not already queued;
    possibly block current thread;
}
dequeue current thread if it was queued;
```

- unlock()

```
update synchronization state;
if(state may permit a blocked thread to acquire)
    unlock one or more queued threads;
```

条件变量与线程挂起、唤醒

- **Condition**
  - **await() → wait()**
  - **signal() → notify()**
  - **signalAll() → notifyAll()**
- **LockSupport**
  - park()
  - unpark()

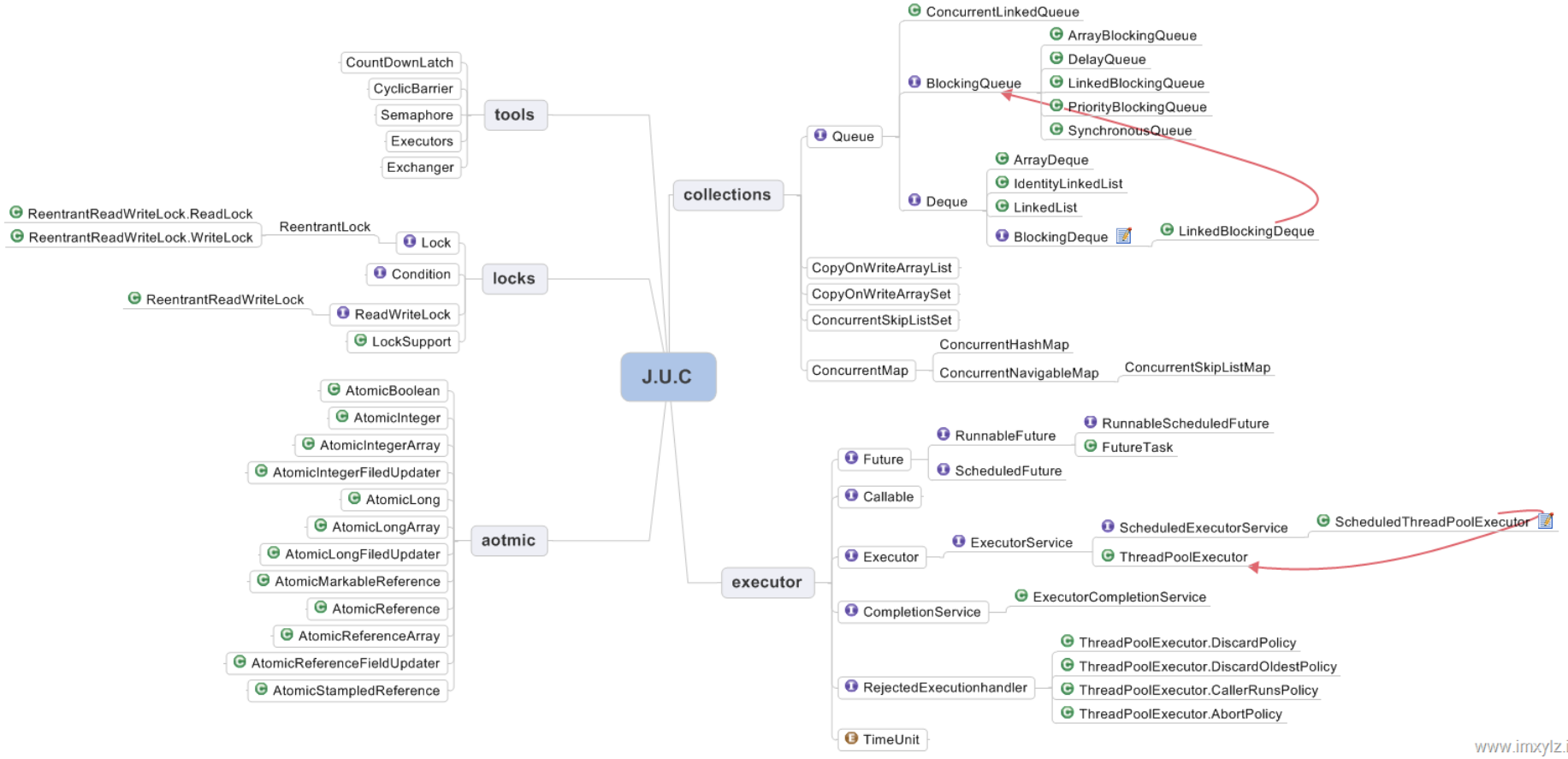
锁机制其它问题

- 内部锁
  - synchronized
  - volatile
- 性能
- 线程阻塞
  - 自旋等待/挂起、唤醒
- 锁竞争
  - 持有时间/请求频率/共享锁、独占锁
- 死锁

窥视Java并发包

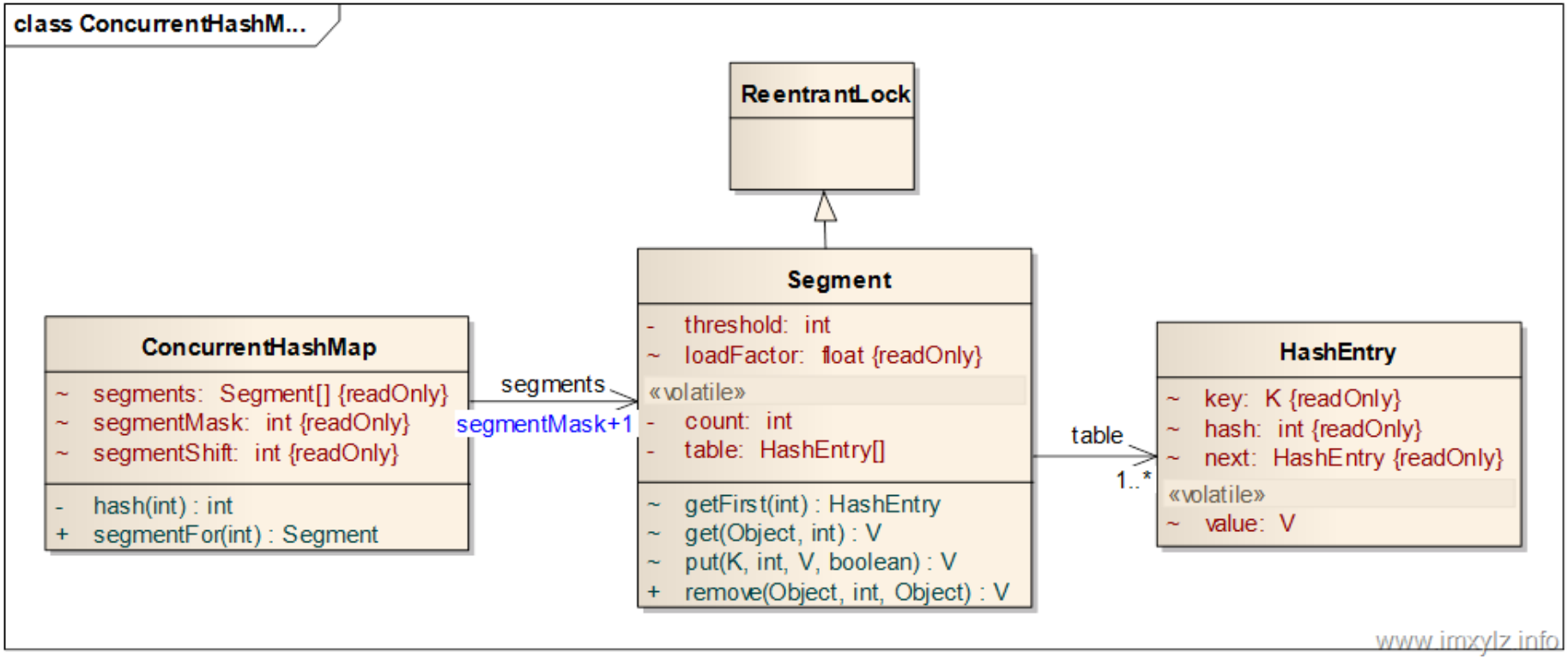
- JUC体系结构
- ConcurrentHashMap的原理
- 徘徊BlockingQueue
- 线程池的最佳实践

JUC体系结构



CocurrentHashMap原理

- 分片段独占锁



ConcurrentHashMap注意点

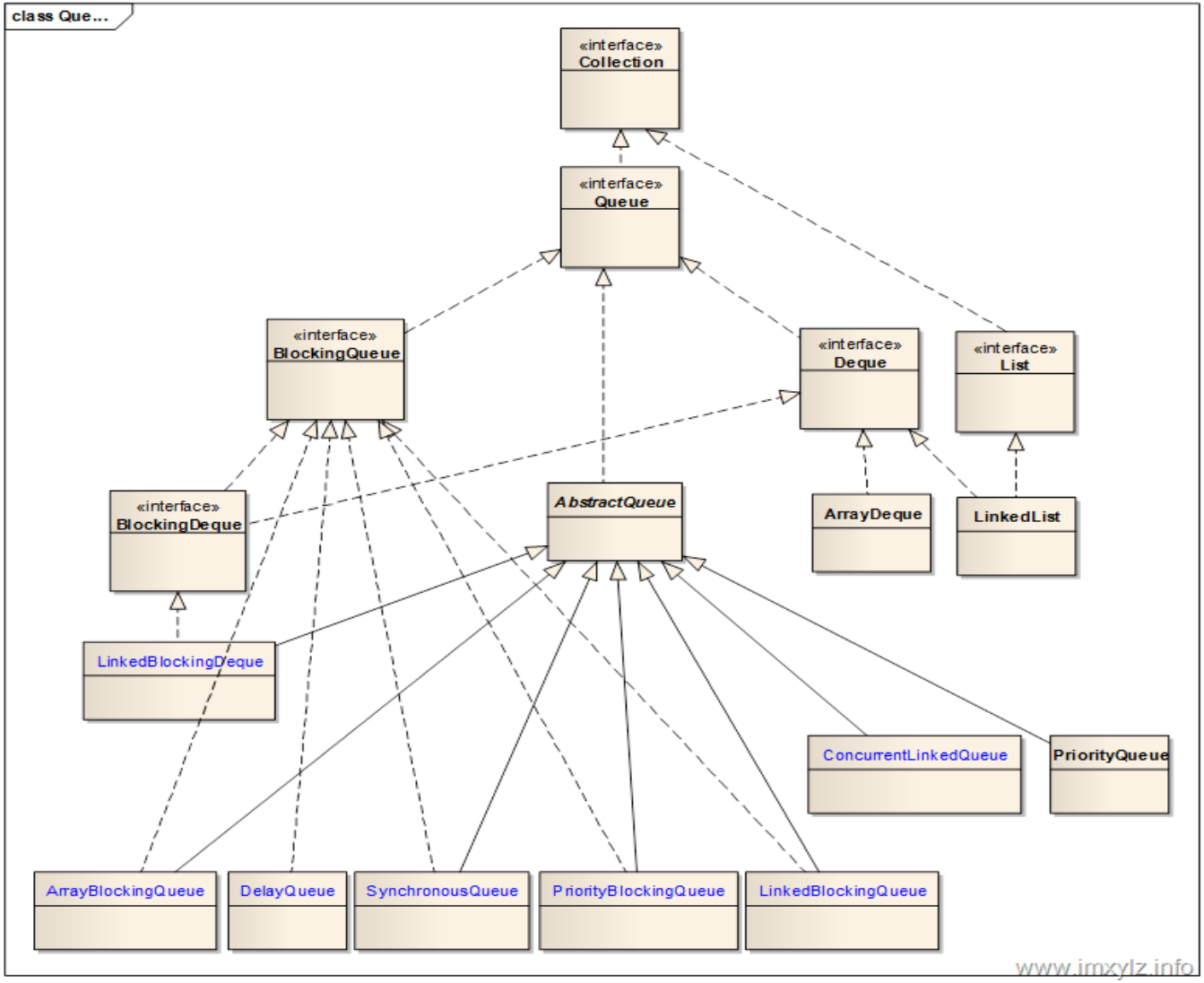
- 注意点
  - 遍历操作开销大
    - size()
    - containsValue(Object)
    - keys()/values()/entrySet()
  - putIfAbsent/replace
  - get()操作可能无锁

ConcurrentHashMap注意点

- get()无锁？有锁？

```
334         V get(Object key, int hash) {
335             if (count != 0) { // read-volatile
336                 HashEntry<K,V> e = getFirst(hash);
337                 while (e != null) {
338                     if (e.hash == hash && key.equals(e.key)) {
339                         V v = e.value;
340                         if (v != null)
341                             return v;
342                         return readValueUnderLock(e); // recheck
343                     }
344                     e = e.next;
345                 }
346             }
347             return null;
348         }
349
```

徘徊BlockingQueue



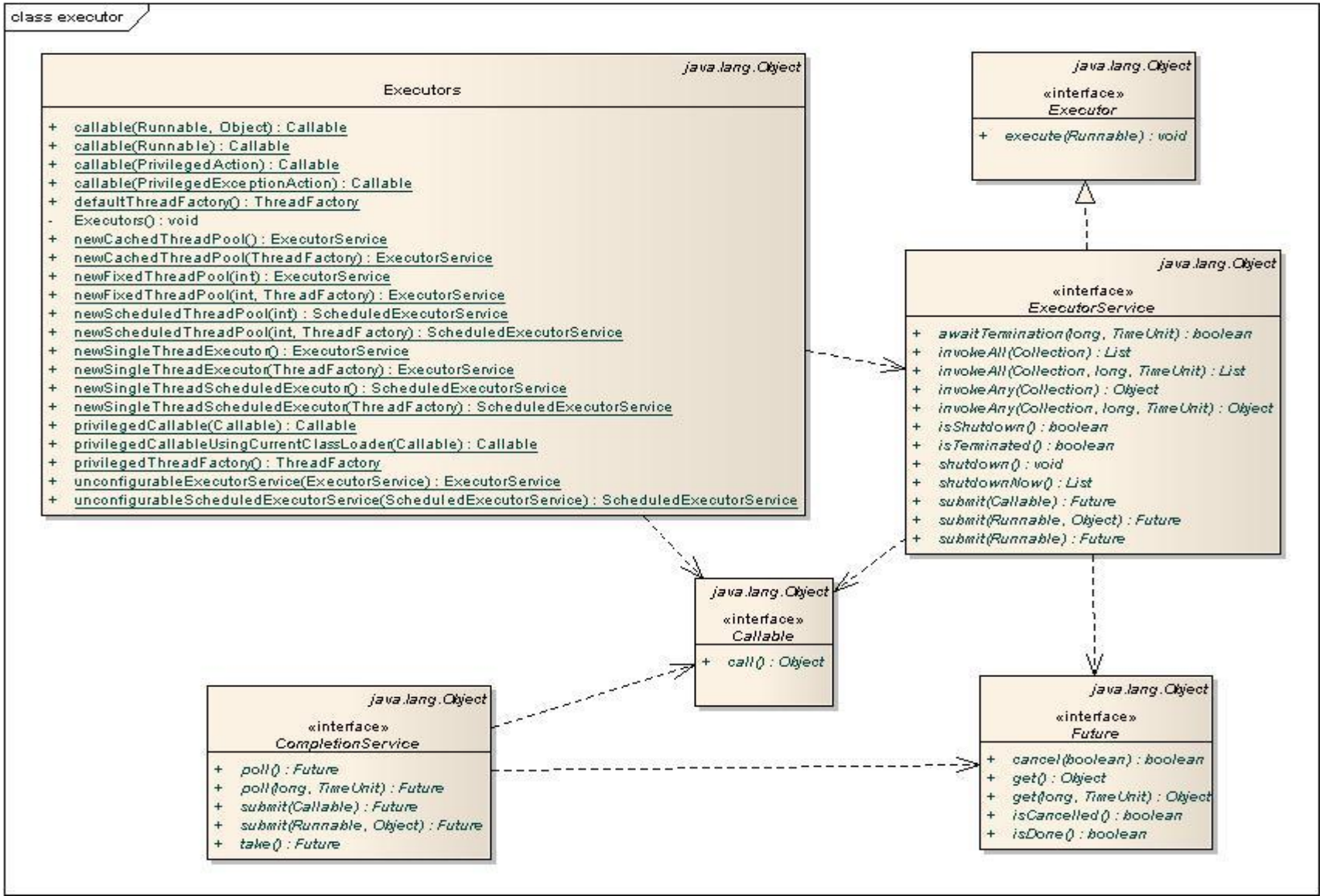
徘徊BlockingQueue

	抛出异常	特殊值	阻塞	超时	描述
插入	add(e)	offer(e)	put(e)	offer(e,time,unit)	将元素加入到队列尾部
移除	remove()	poll()	take()	poll(time,unit)	移除队列头部的元素
检查	element()	peek()			读取而不删除队列头部的元素

徘徊BlockingQue

队列	场景	优点	缺点
ConcurrentLinkedQueue	Queue 的最佳线程安全版本，在不适用阻塞功能下几乎是最有效的。	使用原子操作，效率最高，同时是一种无界的队列	不能阻塞线程，同时无法直接获取队列大小，也不能控制队列的容量
LinkedBlockingQueue	基于链表的 Queue 的可阻塞线程安全版本，在需要阻塞且无界的环境下，这是一种不错的选择。	可阻塞，入出队列锁分离，效率高，支持容量限制，可以作为无界的队列实现	由于存在锁机制，同时链表需要遍历遍历才能准确定位元素，因此效率有一点的影响
ArrayBlockingQueue	基于数组的 Queue 的可阻塞线程版本，在容量固定的环境下是一种不错的阻塞实现。	可阻塞，入出队列效率更高，同时能省内存	容量固定，不能扩容，入出队列不能同时进行，遍历元素更快
PriorityBlockingQueue	按照自然排序实现的阻塞队列，在元素需要排序的情况下是唯一的选择。	可阻塞，元素有序，能够自动扩容	出入队列比较慢，效率比较低，基于数组实现，每次扩容需要数组复制，同时容量不能减小，入队列不能够被阻塞
SynchronousQueue	一种直接交换元素的实现，这在快速处理任务队列是最有效的方式。	可阻塞，快速交换队列	内部没有容量
DelayQueue	延时处理队列的是吸纳，队列中每个元素都有一个延时时间，当且仅当延时时间过期后才能出队列。	可阻塞，可延时	基于排序的 Queue 实现，效率很低，入队列不能够被阻塞

线程池的最佳实践



线程池的最佳实践

- **Executors**
  - newFixedThreadPool(int nThreads)
  - newCachedThreadPool()
  - newSingleThreadExecutor()
  - newScheduledThreadPool(int corePoolSize)
- **Callable/Future获取任务结果**
- **CompletionService顺序获取完成结果**

线程池的最佳实践

```
public ThreadPoolExecutor(
    int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory,
    RejectedExecutionHandler handler)
```

真的需要这么复杂的结构么？

并发的小陷阱

```
public class Runner{
   int x,y;
   Thread thread;
   public Runner(){
      this.x=1;
      this.y=2;
      this.thread=new MyThread();
      this.thread.start();
   }
}
```

并发的小陷阱

```
Map map=Collections.synchronizedMap(new HashMap());
        if(!map.containsKey("a")){
                map.put("a", value);
        }
```

并发的小陷阱

```
public class ThreadSafeCache{
  int result;
  public int getResult(){
     return result;
  }
  public synchronized void setResult(int result)
  {
     this.result=result;
  }
}
```

学习体会

- 编码规范
- 算法、原理
- 设计模式
- 性能、安全、扩展

面向对象的五个原则

- SRP(Single Responsibility Principle) 单一职责原则
  - 事情不要太复杂
- OCP(Open-Closed Principle) 开闭原则
  - 对扩展开发，对修改关闭
- DIP(Dependency Inversion Principle) 依赖倒置
  - 实现依赖抽象，而不是抽象依赖实现
- ISP(Interface Segregation Principle) 接口隔离原则
  - 依赖的接口粒度要小，不强迫依赖于无关的接口
- LSP(Liskov Substitution Principle) 里氏替换原则
  - 子类拥有父类全部的特征

明星软件工程师的十种特质

- 热爱编程
- 完成事情
- 持续重构代码
- 使用设计模式
- 编写测试
- 善用现有代码
- 专注可用性
- 编写可维护代码
- 能用任何语言编程
- 知晓基本的计算机科学知识

http://www.williamlong.info/archives/2424.html

淘宝网 Taobao.com 淘我喜欢！

Q&A

THANX