Java

1. 一个".java"源文件中是否可以包括多个类?

可以，但只能有一个public类，而且如果有public类的话，这个文件的名字要和这个类的名字一样。

2. 源文件javac出多个class文件出来是怎么回事?

```
public class A {
}

class B {
}

class C {
}
```

这样每个class会是一个.class文件。

```
public class A {
    class B {
    }
}
```

这会产生两个.class文件，一个A.class，一个A$B.class。

```
public class A {
    void xxx() {
        button.addActionLisener(new ActionListener() {
            ...
        });
    }
}
```

使用匿名类，这也会产生多个.class，一个A.class，一个A$1.class。

3. 什么是匿名类?

```
new 父类构造器(参数列表)|实现接口() {
        //匿名内部类的类体部分
    }
```

- 使用匿名内部类时，我们必须是继承一个类或者实现一个接口，但是两者不可兼得，同时也只能继承一个类或者实现一个接口。
- 匿名内部类中是不能定义构造函数的。
- 匿名内部类中不能存在任何的静态成员变量和静态方法。
- 匿名内部类为局部内部类，所以局部内部类的所有限制同样对匿名内部类生效。
- 匿名内部类不能是抽象的，它必须要实现继承的类或者实现的接口的所有抽象方法。

4. `switch case` 中 `switch` 后的变量类型可以是什么?

- 可以转换为int的类型。
- String类型。
- 枚举类型。

5. `char` 型变量与汉字。

Java中的一个 `char` 占2个字节。Java采用unicode，2个字节来表示一个字符，如 `char x = '编'` 。

`String.getBytes(encoding)` 方法是获取指定编码的 `byte` 数组表示，通常gbk/gb2312是2个字节，utf-8是3个字节。如果不指定 `encoding` 则取系统默认的 `encoding` 。

6. 使用final关键字修饰一个变量时，是引用不能变，还是引用的对象不能变?

使用final关键字修饰一个变量时，是指引用变量不能变，引用变量所指向的对象中的内容 还是可以改变的。

7. Overload和Override的区别。

Overload是重载的意思，Override是覆盖的意思，也就是重写。

8. 构造器Constructor是否可被override?

构造器Constructor不能被继承，因此不能重写Override，但可以被重载Overload。

9. Java抽象类（abstract class）和类（class）的区别?

- 抽象类不能实例化；
- 抽象类允许有abstract方法；
- 抽象类的非抽象子类必须实现abstract方法。

10. Java接口与抽象类如何合作?

- 接口可以继承接口。抽象类可以实现（implements）接口，抽象类是可继承实体类，但前提是实体类必须有明确的构造函数。
- 一个Java抽象类实现一个接口时，可以不实现接口中所有的方法，但抽象类的子类必须实现。

11. Java中实现多态的机制是什么?

多态由重载和重写体现。

重载通过静态分配实现，即依赖静态类型来定位方法执行的版本。静态分配发生在编译阶段，确定静态分配的动作实际上不是由虚拟机来执行的。

重写通过动态分配实现，即在运行期根据实际类型确定方法执行版本。Java虚拟机为类载方法区中建立一个虚方法表，虚方法表中存放着各个方法的实际入口地址。如果某个方法在子类中没有被重写，那么子类的虚方法表里面的地址入口和父类相同方法的入口地址是一致的，都指向父类的实现入口。如果子类中重写了这个方法，子类方法表中的地址将会替换为指向子类实现版本的入口地址。

12. Java实现了闭包吗?

Java实现了闭包，但仅实现了值捕获，没有实现引用捕获。

13. `String s = new String("xyz")` 创建了几个String Object?

两个或一个，`"xyz"` 对应一个对象，这个对象放在字符串常量缓冲区，常量 `"xyz"` 不管出现多少遍，都是

缓冲区中的那一个。每次 `new` 都会创建一个新的对象，但 `"xyz"` 仍从缓冲区获取。

```
public class Main {
    public static void main(String[] args) {
        String a = new String("abc");
        String b = "abc";
        System.out.println("abc" == a);//false
        System.out.println("abc" == b);//true
        System.out.println(a == b);//false
    }
}
```

```
public String(String original) {
    this.value = original.value;
    this.hash = original.hash;
}
```

14. try-catch-finally-return的执行顺序

1. 不管是否出现异常，finally块都会执行；
2. 当try或catch中有return时，finally块仍然会执行；
3. 若try或catch中执行到return，则finally是在return后的表达式执行完成后才执行的（此时return语句并未返回，而是将要返回的值保存起来，待finally执行完成后返回；如果finally中有return，则返回值以finally中为准。

参考：Java异常捕获之try-catch-finally-return的执行顺序

15. 当一个线程进入一个对象的一个synchronized方法后，其它线程是否可进入此对象的其它方法?

若synchronized修饰的是static方法，则获取到的是类锁，否则是对象锁。若其他线程进入的是非synchronized修饰的方法，则可进入；若修饰的方法需要的锁与当前线程相同，则不可进入。

16. ArrayList和Vector的区别

- ArrayList在容量不够时默认是扩展50% + 1个，Vector是默认扩展1倍。
- Vector提供 `indexOf(obj, start)` 方法，ArrayList没有。
- Vector是线程安全的，而ArrayList不是。

17. HashMap和Hashtable的区别

1. HashTable是线程安全的，而HashMap不是。
2. HashMap和HashTable都实现了Map接口。
3. HashMap继承自AbstractMap，HashTable继承自Dictionary。
4. HashMap允许key和value为null，而HashTable不允许。

18. List，Set，Map是否继承自Collection接口?

List，Set是，Map不是

19. Collection和 Collections的区别。

Collection是集合类的上级接口，继承于它的接口主要有Set和List。

Collections是针对集合类的一个帮助类，他提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

20. Java中有几种类型的流？JDK为每种类型的流提供了一些抽象类以供继承，请说出他们分别是哪些类?

字节流，字符流。字节流继承于 `InputStream` / `OutputStream` ，字符流继承于 `InputStreamReader` / `OutputStreamWriter` 。在java.io包中还有许多其他的流，主要是为了提高性能和使用方便。

21. 描述一下JVM加载class文件的原理机制?

JVM中类的装载是由ClassLoader和它的子类来实现的，ClassLoader是一个重要的Java运行时系统组件。它负责在运行时查找和装入类文件的类。

22. 能不能自己写个类，也叫java.lang.String?

可以，但在应用的时候，需要用自己的类加载器去加载，否则，系统的类加载器永远只是去加载re.jar包中的那个 `java.lang.String` 。

23. Java中反射的作用是什么?

Java反射机制是在运行时，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法；这种动态获取的信息以及动态调用对象的方法的功能称为Java语言的反射机制。Java反射机制主要提供了以下功能：

1. 在运行时判断任意一个对象所属的类；
2. 在运行时构造任意一个类的对象；
3. 在运行时判断任意一个类所具有的成员变量和方法；
4. 在运行时调用任意一个对象的方法；生成动态代理。

24. 成员变量、局部变量、静态变量的区别

属性	成员变量	局部变量	静态变量
定义位置	在类中，方法外	方法中，或者方法的形式参数	在类中，方法外
初始化值	有默认初始化值	无，先定义，赋值后才能使用	有默认初始化值
调用方式	对象调用	---	对象调用，类名调用
存储位置	堆中	栈中	方法区
生命周期	与对象共存亡	与方法共存亡	与类共存亡
别名	实例变量	---	类变量

25. 谈谈你对StrongReference、WeakReference和SoftReference的认识

- 强引用（StrongReference）：就是在代码中普遍存在的，类似 `Object obj = new Object()` 这类的引用，只要强引用还存在，GC永远不会回收掉被引用的对象。
- 软引用（SoftReference）：用来描述一些还有用但非必须的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常时，将会把这些对象列入回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。
- 弱引用（WeakReference）：也是用来描述非必须对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到了下一次GC发生之前。当GC工作时，无论当时内存是否足够，都会回收只被弱引用关联的对象。
- 虚引用（PhantomReference）：虚引用也称幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用的唯一目的就是在这个对象被GC回收是收到一个系统通知。

26. `==` 与 `equals()` 的区别?

- `==` 判断两个对象的地址是否相等（对于基本类型判断值是否相等）。
- `equals()` 会寻找自身或最近的父类实现，调用其 `equals()` 。
- 若无任何父类实现 `equals()` ，则会调用Object的 `equals()` ，其与==相同。

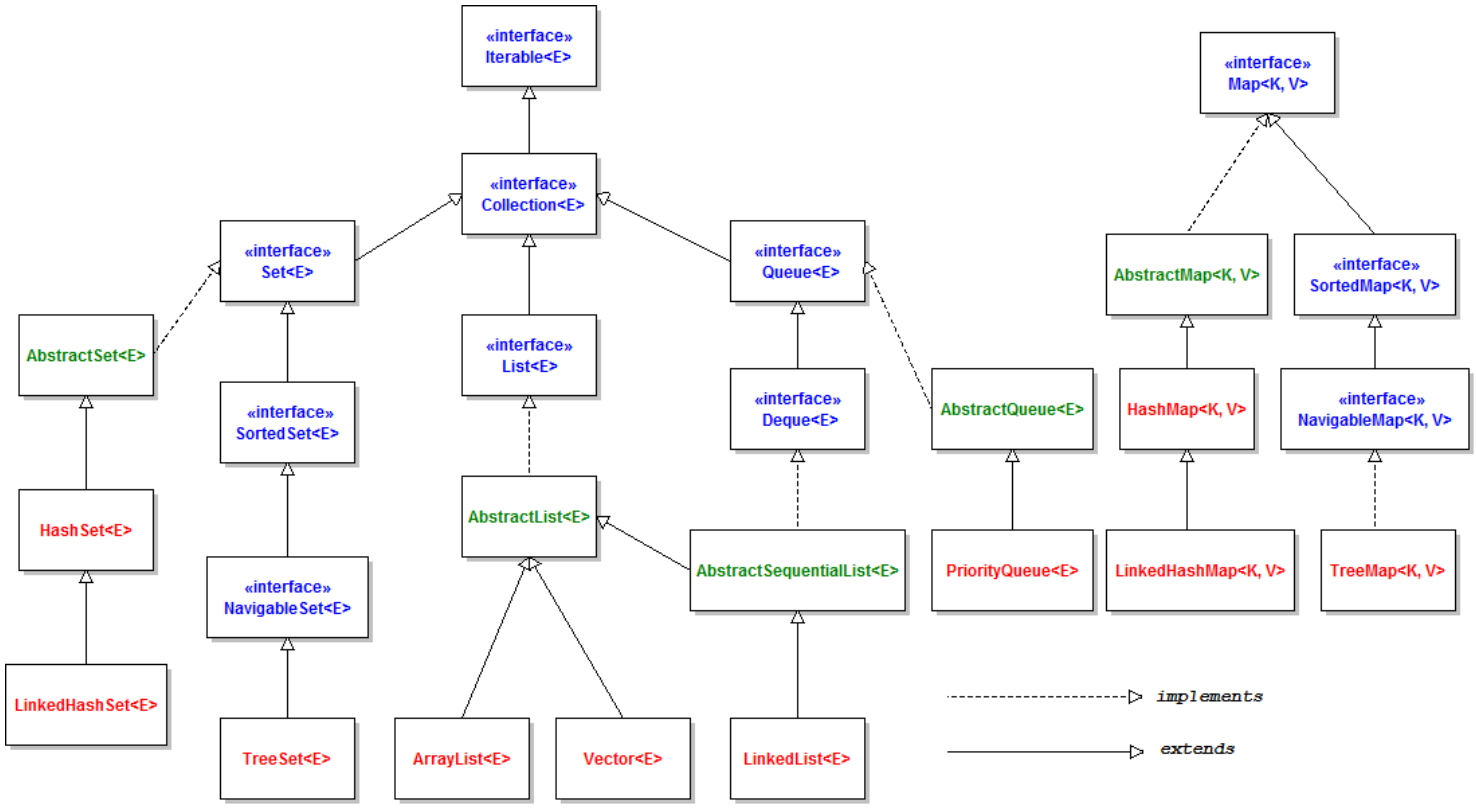
27. `equals()` 与 `hashCode()` 的区别?

- `equals()` 仅在显示调用 `equals()` 时使用，而 `hashCode()` 在如散列表中会自动调用，以判断是否为同一对象。
- 重写 `equals()` 时必须重写 `hashCode()` ，否则会造成不可预料的后果。

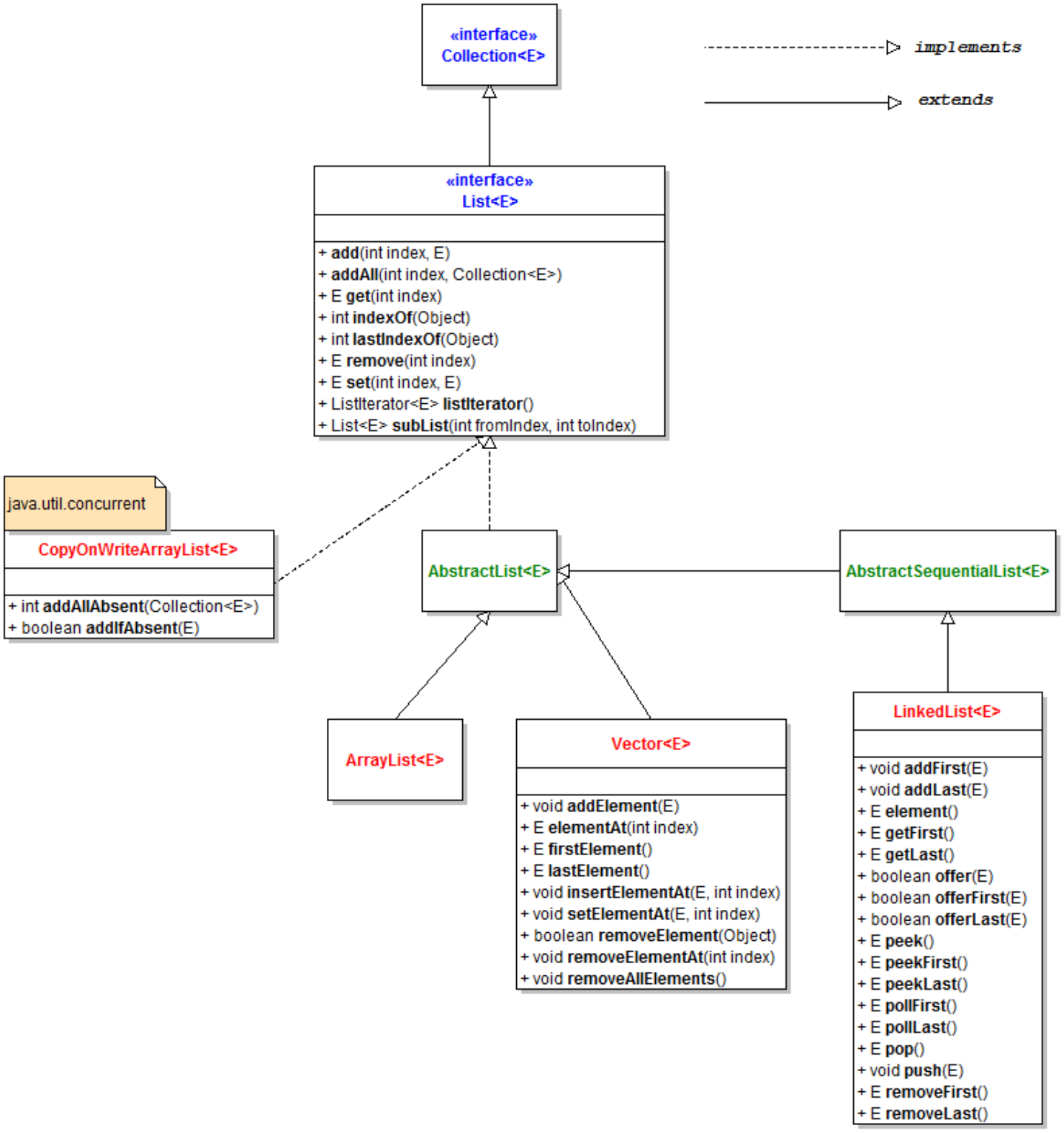
28. Java集合框架示意图

28.1. 集合框架概览

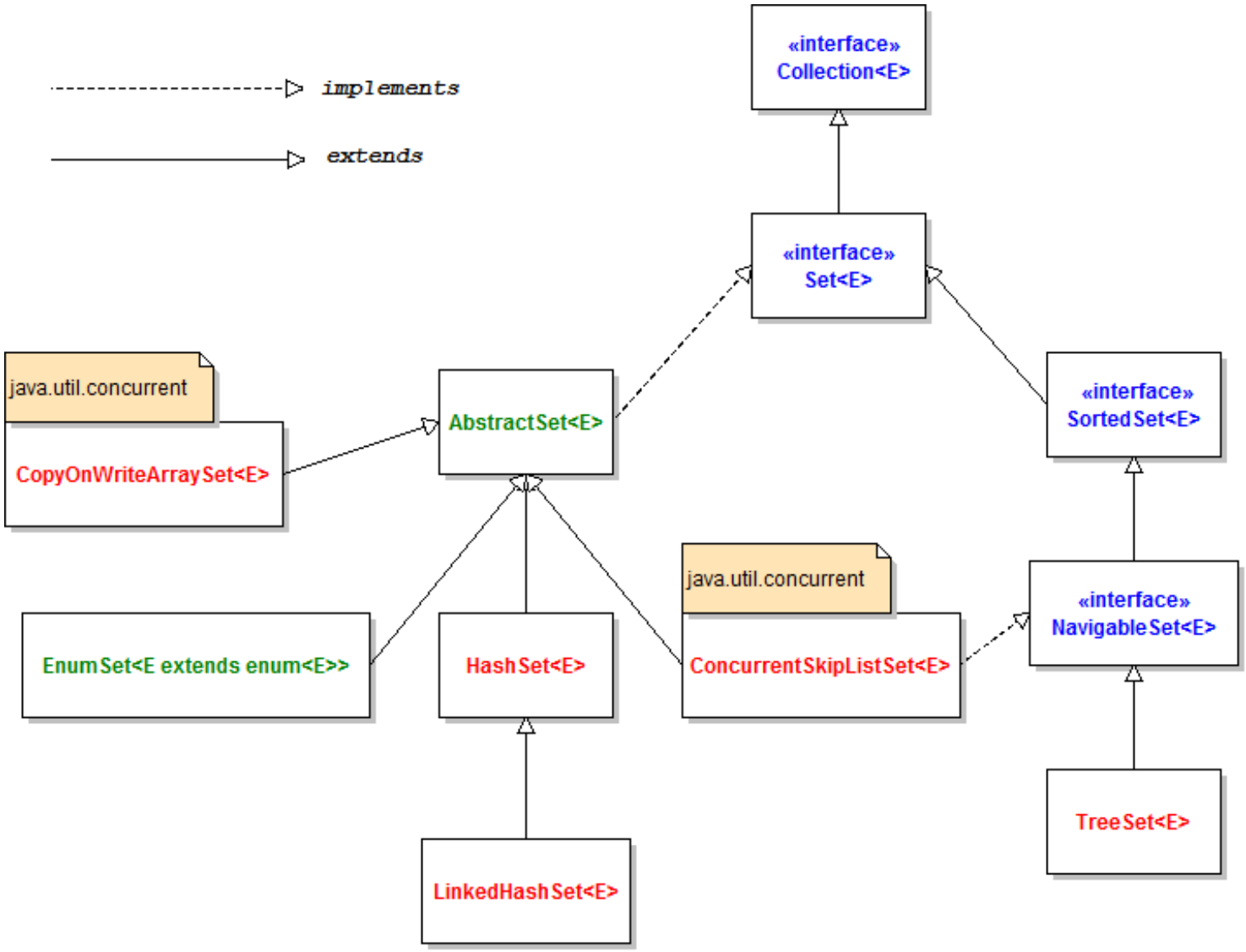
Collection class	Allow duplicate	Ordered	Sorted	Thread-safe
ArrayList	Yes	Yes	No	No
LinkedList	Yes	Yes	No	No
Vector	Yes	Yes	No	Yes
HashSet	No	No	No	No
LinkedHashSet	No	Yes	No	No
TreeSet	No	Yes	Yes	No
HashMap	No	No	No	No
LinkedHashMap	No	Yes	No	No
Hashtable	No	No	No	Yes
TreeMap	No	Yes	Yes	No


«interface»
Iterable<E>
«interface»
Collection<E>
«interface»
Set<E>
«interface»
List<E>
«interface»
Queue<E>
«interface»
Map<K, V>
AbstractMap<K, V>
«interface»
SortedMap<K, V>
AbstractSet<E>
«interface»
SortedSet<E>
«interface»
Deque<E>
AbstractQueue<E>
HashMap<K, V>
«interface»
NavigableMap<K, V>
HashSet<E>
AbstractList<E>
«interface»
NavigableSet<E>
AbstractSequentialList<E>
PriorityQueue<E>
LinkedHashMap<K, V>
TreeMap<K, V>
LinkedHashSet<E>
TreeSet<E>
ArrayList<E>
Vector<E>
LinkedList<E>
implements
extends

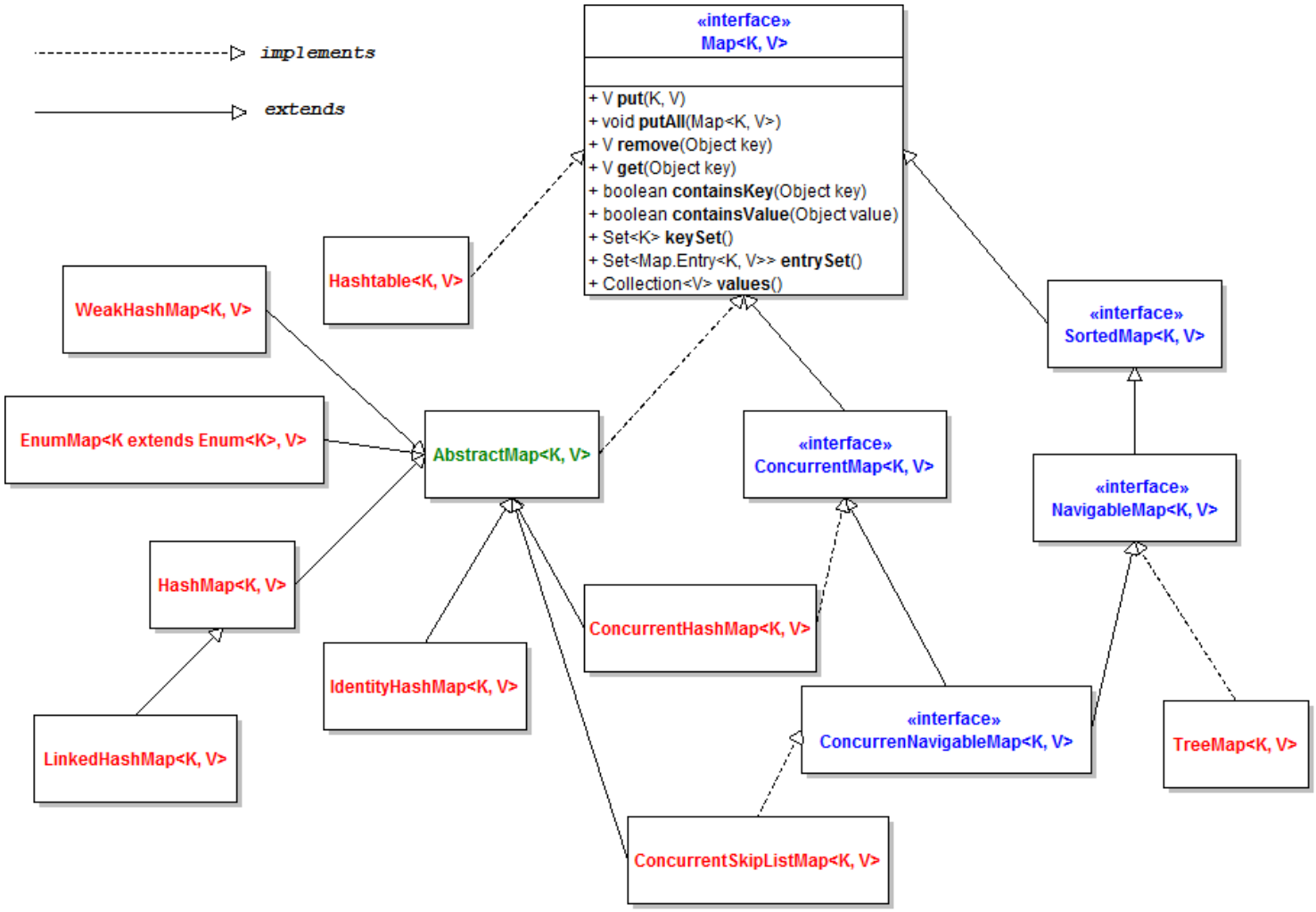
28.2. List


«interface»
Collection<E>
implements
extends
«interface»
List<E>
+ add(int index, E)
+ addAll(int index, Collection<E>)
+ E get(int index)
+ int indexOf(Object)
+ int lastIndexOf(Object)
+ E remove(int index)
+ E set(int index, E)
+ ListIterator<E> listIterator()
+ List<E> subList(int fromIndex, int toIndex)
java.util.concurrent
CopyOnWriteArrayList<E>
+ int addAllAbsent(Collection<E>)
+ boolean addIfAbsent(E)
AbstractList<E>
AbstractSequentialList<E>
ArrayList<E>
Vector<E>
+ void addElement(E)
+ E elementAt(int index)
+ E firstElement()
+ E lastElement()
+ void insertElementAt(E, int index)
+ void setElementAt(E, int index)
+ boolean removeElement(Object)
+ void removeElementAt(int index)
+ void removeAllElements()
LinkedList<E>
+ void addFirst(E)
+ void addLast(E)
+ E element()
+ E getFirst()
+ E getLast()
+ boolean offer(E)
+ boolean offerFirst(E)
+ boolean offerLast(E)
+ E peek()
+ E peekFirst()
+ E peekLast()
+ E pollFirst()
+ E pollLast()
+ E pop()
+ void push(E)
+ E removeFirst()
+ E removeLast()

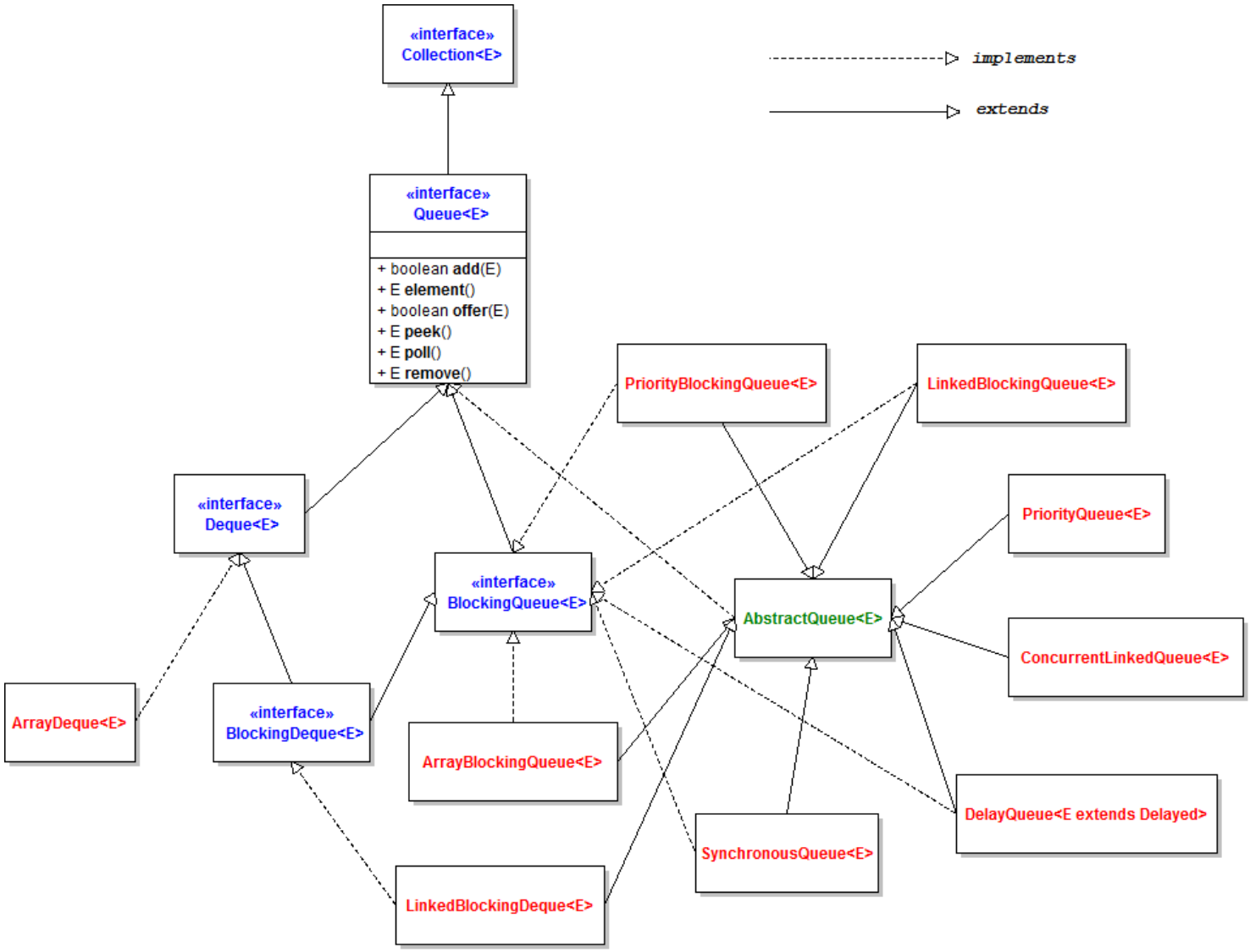
28.3. Set


implements
extends
«interface»
Collection<E>
«interface»
Set<E>
java.util.concurrent
CopyOnWriteArraySet<E>
AbstractSet<E>
«interface»
SortedSet<E>
java.util.concurrent
EnumSet<E extends enum<E>>
HashSet<E>
ConcurrentSkipListSet<E>
«interface»
NavigableSet<E>
TreeSet<E>
LinkedHashSet<E>

28.4. Map


implements
extends
«interface»
Map<K, V>
+ V put(K, V)
+ void putAll(Map<K, V>)
+ V remove(Object key)
+ V get(Object key)
+ boolean containsKey(Object key)
+ boolean containsValue(Object value)
+ Set<K> keySet()
+ Set<Map.Entry<K, V>> entrySet()
+ Collection<V> values()
Hashtable<K, V>
WeakHashMap<K, V>
EnumMap<K extends Enum<K>, V>
AbstractMap<K, V>
«interface»
ConcurrentMap<K, V>
«interface»
SortedMap<K, V>
«interface»
NavigableMap<K, V>
HashMap<K, V>
ConcurrentHashMap<K, V>
IdentityHashMap<K, V>
LinkedHashMap<K, V>
«interface»
ConcurrenNavigableMap<K, V>
TreeMap<K, V>
ConcurrentSkipListMap<K, V>

28.5. Queue



29. Error和Exception的区别

Error类和Exception类的父类都是throwable类，他们的区别是：

- Error类一般是指与虚拟机相关的问题，如系统崩溃，虚拟机错误，内存空间不足，方法调用栈溢等。对于这类错误的导致的应用程序中断，仅靠程序本身无法恢复和和预防，遇到这样的错误，建议让程序终止。
- Exception类表示程序可以处理的异常，可以捕获且可能恢复。遇到这类异常，应该尽可能处理异常，使程序恢复运行，而不应该随意终止异常。

30. 用户线程（User Thread）与守护线程（Daemon Thread）

- JVM中存在两种线程：用户线程和守护线程
- 当线程中只剩下守护线程时JVM就会退出，反之还有任意一个用户线程在，JVM都不会退出。

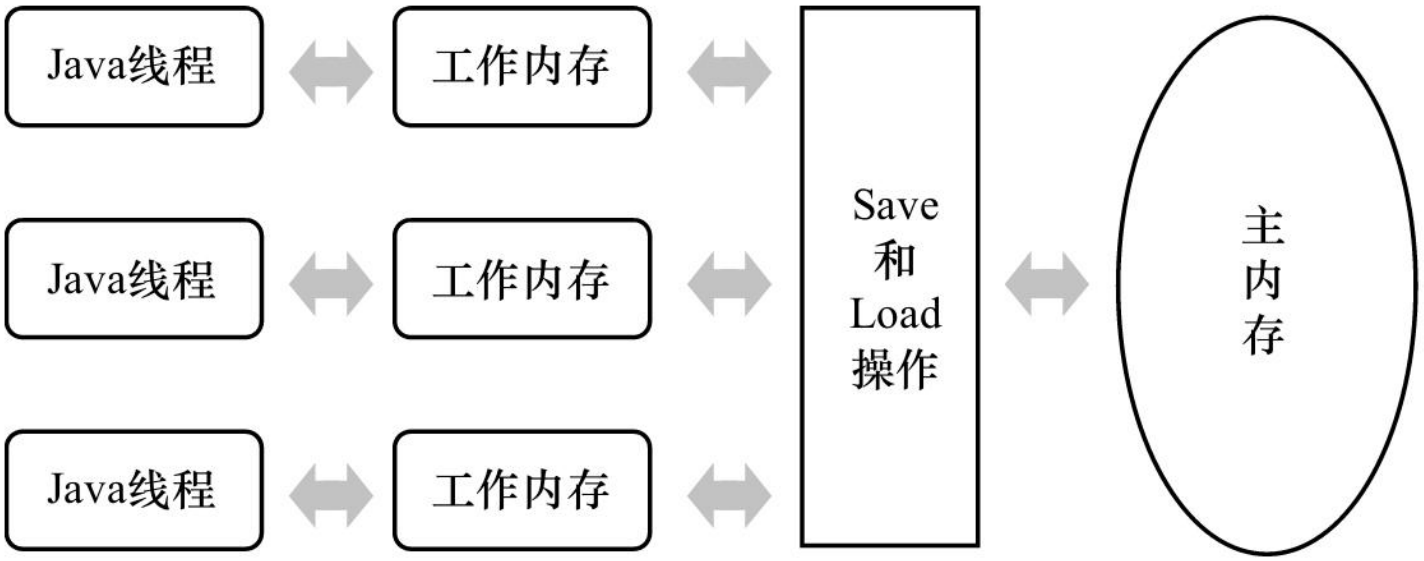
- `thread.setDaemon(true)` 必须在 `thread.start()` 之前设置，否则会抛出IllegalThreadStateException异常。
- 在守护线程中产生的线程也是守护线程。

31. Java内存模型

Java内存模型（Java Memory Model，JMM）用来屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。Java内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存中和从内存中取出变量这样的底层细节（此处的变量与Java编程中所说的变量有所区别，它包括了实例字段、静态字段和构成数组对象的元素，但不包括局部变量与方法参数）。

31.1. 主内存与工作内存

Java内存模型规定了所有的变量都存储在主内存（Main Memory）中。每条线程还有自己的工作内存（Working Memory），线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。



31.2. 内存间交互操作

关于主内存与工作内存之间具体的交互协议，Java内存模型定义了8种操作来完成，虚拟机保证每种操作都是原子的、不可再分的：

- lock（锁定）：作用于主内存的变量，它把一个变量标识为一条线程独占的状态。
- unlock（解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其

他线程锁定。

- read（读取）：作用于主内存的变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用。
- load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。
- use（使用）：作用于工作内存的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作。
- assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
- store（存储）：作用于工作内存的变量，它把工作内存中一个变量的值传送到主内存中，以便随后的write操作使用。
- write（写入）：作用于主内存的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中。

如果要把一个变量从主内存复制到工作内存，就要顺序执行read和load操作；如果要把变量从工作内存同步到主内存，就要顺序执行store和write操作。Java内存模型只要求上述两个操作必须按顺序执行，而没有保证是连续执行。

Java内存模型还规定了在执行时满足的规则：

- 不允许read和load，store和write操作之一单独出现，即不允许一个变量从主内存读取了但工作内存不接受，或者从工作内存发起会写了但主内存不接受的情况出现。
- 不允许一个线程丢弃它最近的assign操作，即变量在工作内存中改变了之后必须把该变化同步回主内存。
- 不允许一个线程无原因地(没有发生任何assign操作)把数据从线程的工作内存同步回主内存中。
- 一个新的变量只能在主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化(load或assign)的变量，换句话说就是对一个变量实施use和store操作之前，必须先执行了load和assign操作。
- 一个变量在同一个时刻只允许一条线程对其进行lock操作，但lock操作可以被同一条线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁。
- 如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load操作初始化变量的值。
- 如果一个变量事先没有被lock操作锁定，则不允许对它执行unlock操作；也不允许unlock一个被其它线程锁定住的变量。
- 对一个变量执行unlock之前，必须先把此变量同步回主内存中。

这8种内存访问操作以及上述规则限制，再加上volatile的一些特殊规定，就完全确定了Java程序中哪些内存访问操作在并发下是安全的。

31.3. 对volatile型变量的特殊规则

volatile是Java虚拟机提供的最轻量级的同步机制。当一个变量定义为volatile之后，它将具备两种特性：

- 保证此变量对所有线程的可见性。这里“可见性”是指当一条线程修改了这个变量的值，新值对于其他线程来

说是可以立即得知的。
- 禁止指令重排优化。

假定T表示一个线程，V和W分别表示两个volatile变量，那么在进行read、load、use、assign、store和write时需要满足以下三条规则：

- 只有当线程T对变量V执行的前一个动作是load时，T才能对V执行use；并且，只有当T对V执行的后一个动作是use时，T才能对V执行load。T对V的use动作可以认为是和线程T对V的load，read动作相关联，必须连续一起出现（这条规则要求在工作内存中，每次使用V前都必须先从主内存刷新最新的值，用于保证能看见其他线程对变量V所做的修改后的值）。
- 只有当线程T对变量V执行的前一个动作是assign时，T才能对V执行store动作；并且，只有当T对变量V执行的后一个动作是store时，线程T才能对变量V执行assign动作。线程T对变量V的assign动作可认为是和线程T对变量V的store，write动作相关联，必须连续一起出现（这条规则要求在工作内存中，每次修改V后都必须立刻同步回主内存中，用于保证其他线程可以看到自己对变量V所做的修改）。
- 假定动作A是线程T对变量V实施的use或assign操作，假定动作F是和动作A相关联的load或store动作，假定动作P是和动作F相应的变量V的read或write动作；类似的，假定动作B是线程T对变量W实施的use或assign动作，假定动作G是和动作B相关联的load或store动作，假定动作Q是和动作G相应的变量W的read或write动作。如果A先于B，那P先于Q（这条规则要求volatile修饰的变量不会被指令重排序优化，保证代码的执行顺序与程序的顺序相同）。

为了实现volatile的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。对于编译器来说，发现一个最优布置来最小化插入屏障的总数几乎不可能，为此，JMM采取保守策略。下面是基于保守策略的JMM内存屏障插入策略：

- 在每个volatile写操作的前面插入一个StoreStore屏障。
- 在每个volatile写操作的后面插入一个StoreLoad屏障。
- 在每个volatile读操作的后面插入一个LoadLoad屏障。
- 在每个volatile读操作的后面插入一个LoadStore屏障。

LoadLoad屏障

序列：Load1, LoadLoad, Load2

确保Load1所要读入的数据能够在被Load2和后续的load指令访问前读入。通常能执行预加载指令或/和支持乱序处理的处理器中需要显式声明Loadload屏障，因为在这些处理器中正在等待的加载指令能够绕过正在等待存储的指令。而对于总是能保证处理顺序的处理器上，设置该屏障相当于无操作。

StoreStore屏障

序列：Store1, StoreStore, Store2

确保Store1的数据在Store2以及后续Store指令操作相关数据之前对其它处理器可见（例如向主存刷新数据）。

通常情况下，如果处理器不能保证从写缓冲或/和缓存向其它处理器和主存中按顺序刷新数据，那么它需要使用StoreStore屏障。

LoadStore屏障

序列：Load1, LoadStore, Store2

确保Load1的数据在Store2和后续Store指令被刷新之前读取。在等待Store指令可以越过loads指令的乱序处理器上需要使用LoadStore屏障。

StoreLoad屏障

序列: Store1, StoreLoad, Load2

确保Store1的数据在被Load2和后续的Load指令读取之前对其他处理器可见。StoreLoad屏障可以防止一个后续的load指令 不正确的使用了Store1的数据，而不是另一个处理器在相同内存位置写入一个新数据。

31.4. 原子性、可见性与有序性

Java内存模型是围绕着在并发过程中如何处理原子性、可见性和有序性这3个特征来建立的。

31.4.1. 原子性（Atomicity）

由Java内存模型来直接保证的原子性变量操作包括read、load、assign、use、store和write，我们大致可以认为基本数据类型的访问读写是具备原子性的。

如果还需要更大范围的原子性保证，Java内存模型还提供了lock和unlock操作，反映到字节码指令就是monitorenter和monitorexit，反映到Java代码中就是synchronized关键字。

31.4.2. 可见性（Visibility）

可见性是指当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。Java内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介来实现可见性的，无论是普通变量还是volatile变量都是如此，普通变量与volatile变量的区别是，volatile的特殊规则保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。因此，可以说volatile保证了多线程时操作变量的可见性，而普通变量则不能保证这一点。

synchronized和final关键字也能实现可见性，synchronized的可见性是由“对一个变量执行unlock操作之前，必须先把此变量同步回主内存中”这条规则获得的。另外，final关键字也可以实现可见性，因为被final修饰的字段在构造器中一旦初始化完成，并且构造器没有把this传递出去，那在其他线程中就能看见final字段的值。

31.4.3. 有序性（Ordering）

Java内存模型的有序性可以总结为一句话，如果在本线程内观察，所有的操作都是有序的；如果在一个线程中观察另一个线程，所有的操作都是无序的。前半句是指“线程内表现为串行的语义”，后半句是指“指令重排序”现象和“工作内存与主内存同步延迟”现象。 Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性，volatile关键字本身就包含了禁止指令重排序的语义，而synchronized则是由“一个变量在同一时刻只允许一条线程对其进行lock操作”这条规则获得的，这条规则规定了持有同一个锁的两个同步块只能串行地进入。

31.5. 先行发生原则

对于有序性，Java语言有“先行发生”（happends-before）的原则，它是判断数据是否存在竞争、线程是否安全的主要依据。依靠这个原则，我们可以通过几条规则一揽子地解决并发环境下两个操作之间是否可能存在冲突的所有问题。

先行发生是Java内存模型中定义的两项操作之间的偏序关系，如果说操作A先行发生于操作B，其实就是说在发生操作B之前，操作A产生的影响能被操作B观察到，“影响”包括修改了内存中共享变量的值、发送了消息、调用了方法等。

下面是Java内存模型下一些“天然的”先行发生关系，这些先行发生关系无须任何同步器协助就已经存在，可以在编码中直接使用。如果两个操作之间的关系不在此列，并且无法从下列规则推导出来的话，它们就没有顺序性保障，虚拟机可以随意地进行重排：

- 程序次序规则（Program Order Rule）：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。
- 管程锁定规则（Monitor Lock Rule）：一个unlock操作先行发生于后面对同一个锁的lock操作。
- volatile变量规则（Volatile Variable Rule）：对一个volatile变量的写操作先行发生于后面对这个变量的读取操作。
- 线程启动规则（Thread Start Rule）：Thread对象的 `start()` 方法先行发生于此线程的每一个动作。
- 线程终止规则（Thread Termination Rule）：线程中的所有操作都先行发生于对此线程的终止检测。
- 线程中断规则（Thread Interruption Rule）：对线程 `interrupt()` 方法的调用先行发生于被中断线程的代码检测到中断事件的发生。
- 对象终结规则（Finalizer Rule）：一个对象初始化完成(构造方法执行完成)先行发生于它的 `finalize()` 方法的开始。
- 传递性（Transitivity)：如果操作A先行发生于操作B，操作B先行发生于操作C，那就可以得出操作A先行发生于操作C的结论。

一个操作“时间上的先发生”不代表这个操作会是“先行发生”，一个操作“先行发生”不代表这个操作会是“时间上的先发生”。

32. Java中的BIO，NIO，AIO分别是什么？

32.1. BIO（synchronous Blocking IO，同步阻塞IO）

如Apache，Tomcat。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。

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。

32.2. NIO（synchronous Non blocking IO，同步非阻塞IO）

如Nginx，Netty。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。

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。

32.3. AIO（Asynchronous non blocking IO，异步非阻塞IO）

还不是特别成熟。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，

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。

33. Serializable接口和序列化与反序列化

- Serializable接口没有任何方法
- 一个类只要实现了Serializable接口，即可被序列化
- 实现Serializable接口的类，在序列化时不能有不可被序列化的成员变量
- 通过ObjectOutputStream和ObjectInputStream对对象进行序列化及反序列化
- 虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个类的序列化ID是否一致（即 `private static final long serialVersionUID` ）
- transient关键字的作用是控制变量的序列化，在变量声明前加上该关键字，可以阻止该变量被序列化到文件中；在被反序列化后，transient变量的值被设为初始值。
- 在序列化过程中，如果被序列化的类中定义了 `writeObject()` 和 `readObject()` 方法，虚拟机就会试图调用对象类里的 `writeObject()` 和 `readObject()` 方法，进行用户自定义的序列化和反序列化。如果没有这样的方法，则默认调用是ObjectOutputStream的 `defaultWriteObject()` 方法以及ObjectInputStream的 `defaultReadObject()` 方法。

序列化算法一般会按步骤做如下事情：

- 将对象实例相关的类元数据输出。
- 递归地输出类的超类描述直到不再有超类。
- 类元数据完了以后，开始从最顶层的超类开始输出对象实例的实际数据值。
- 从上至下递归输出实例的数据。

34. ArrayList的 `subList()` 方法注意事项

`subList()` 方法接口为

```
List<E> subList(int fromIndex, int toIndex);
```

其返回的是原List从[fromIndex,toIndex)之间的一部分的视图（如ArrayList的内部类SubList），实际依赖于原List，且对subList的修改也会作用到原List中。

ArrayList的 `subList()` 结果不可以强制转换为ArrayList（否则会抛出ClassCastException异常）。

```
SubList(AbstractList<E> parent, int offset, int fromIndex, int toIndex) {
    this.parent = parent;
    this.parentOffset = fromIndex;
    this.offset = offset + fromIndex;
    this.size = toIndex - fromIndex;
    this.modCount = ArrayList.this.modCount;
}
```

35. Arrays的 `asList()` 方法注意事项

`Arrays.asList()` 可将数组转换为集合，但转换得到的集合不能使用如 `add()` 、 `remove()` 和 `clear()` 等方法（否则会抛出UnsupportedOperationException异常）。

原因是 `asList()` 返回的是Arrays的内部类ArrayList，使用到的是适配器模式，并未实现集合的某些修改方法。

```
public static <T> List<T> asList(T... a) {
    return new ArrayList<>(a);
}
```

```
ArrayList(E[] array) {
    if (array==null)
        throw new NullPointerException();
    a = array;
}
```

36. Comparator注意事项

Comparator要满足自反性、传递性和对称性，否则会抛出IllegalArgumentException。

- 自反性：x，y的比较结果和y，x的比较结果相反。
- 传递性：x > y，y > z，则x > z。
- 对称性：x = y，则x，z的比较结果和y，z比较结果相同。

37. HashMap多线程下死循环问题

多线程 `put()` 时触发 `resize()` ，进而导致新建Entry数组，并将之前数组中每个链表都重新hash到新的数组中；由于多线程下Entry数组私有，但Entry链表中的元素共享，且由于采用头插法hash到新的链表数组中，导致链表出现环。

而当 `get()` 到此环，而 `get()` 的hash值又与此环的任何元素都不相等时，则出现死循环。

疫苗：Java HashMap的死循环 | 酷 壳 - CoolShell

38. 什么是ConcurrentHashMap

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。

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。

39. Map类集合k / V能否存储null值的情况

集合类	Key允许为null	Value允许为null	Super	说明
Hashtable	No	No	Dictionary	线程安全
ConcurrentHashMap	No	No	AbstractMap	分段锁技术
TreeMap	No	Yes	AbstractMap	线程不安全
HashMap	Yes	Yes	AbstractMap	线程不安全

40. SimpleDataFormat线程安全吗?

SimpleDataFormat线程不安全，一般不要定义为static变量，如果定义为static则必须加锁，或者使用DataUtils工具类。

41. Timer可以用来并行处理定时任务吗?

一个Timer对象仅有一个线程，如果向Timer提交多个TimerTask，且某个TimerTask很耗时，则其他TimerTask即使到了执行时间，也仍会等待之前的task执行完毕；甚至，如果某个TimerTask抛出异常导致线程终止，则其后的TimerTask将不会执行。

42. 可以在多线程下使用Random吗?

Random是线程安全的（由AtomicLong实现），但在多线程时可能遇到效率问题。Random的seed是AtomicLong类型，其使用CAS（compare-and-set）操作来更新；CAS在资源高度竞争时表现会变得很糟糕。

ThreadLocalRandom克服了如上Random的缺陷。

43. `Thread.join()`是如何实现的?

`join()`方法是通过`wait()`实现的。当当前线程调用`otherThread.join()`时，当前线程会获得对象otherThread的锁，调用该对象的`wait()`方法开始等待；直到otherThread唤醒当前线程。而当otherThread退出时，会在native方法中调`notifyAll()`从而唤醒当前线程，当前线程继续运行。

44. GC中可回收对象的判定方法

44.1. 引用计数法

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。

其优点是简单，高效；缺点是很难处理循环引用。

44.2. 可达性分析算法

通过一系列称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GC roots没有任何引用链相连（用图论的话来说，就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。

可作为GC Roots的对象包括：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象。
- 方法区中类静态属性引用的对象。
- 方法区中常量引用的对象。
- 本地方法栈中JNI（即一般说的Native方法）引用的对象。

45. 垃圾收集算法

45.1. 标记——清除算法

标记——清除算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象（标记过程即使用可回收对象的判定方法）。

主要有两方面的不足：

- 效率问题，标记和清除两个过程的效率都不高。
- 空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

45.2. 复制算法

复制算法将可用内存容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另一块上，然后把已使用过的内存空间一次性清理掉。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按序分配内存即可，实现简单，运行高效。

主要的不足是将内存缩小为了原来的一半，代价较高。

45.3. 标记－－整理算法

标记－－整理算法的标记过程仍然与标记－－清除算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

45.4. 分代收集算法

分代收集算法，即：分代分配，分代回收。对象将根据存活的时间被分为：年轻代（Young Generation）、年老代（Old Generation）、永久代（Permanent Generation，也就是方法区）。

45.4.1. 年轻代

年轻代可以分为3个区域：Eden区和两个存活区（Survivor 0、Survivor 1）。

1. 绝大多数刚创建的对象会被分配在Eden区，其中的大多数对象很快就会消亡。Eden区是连续的内存空间，因此在其上分配内存极快；
2. 最初一次，当Eden区满的时候，执行Minor GC，将消亡的对象清理掉，并将剩余的对象复制到一个存活区Survivor0（此时，Survivor1是空白的，两个Survivor总有一个是空白的）；
3. 下次Eden区满了，再执行一次Minor GC，将消亡的对象清理掉，将存活的对象复制到Survivor1中，然后清空Eden区；
4. 将Survivor0中消亡的对象清理掉，将其中可以晋级的对象晋级到Old区，将存活的对象也复制到Survivor1区，然后清空Survivor0区；
5. 当两个存活区切换了几次（HotSpot虚拟机默认15次，用-XX:MaxTenuringThreshold控制，大于该值进入老年代，但这只是个最大值，并不代表一定是这个值）之后，仍然存活的对象（其实只有一小部分，比如，我们自己定义的对象），将被复制到老年代。

这种垃圾回收的方式就是复制算法。由于绝大部分的对象都是短命的，甚至存活不到Survivor中，所以，Eden区与Survivor的比例较大，HotSpot默认是8:1，即分别占新生代的80%，10%，10%。如果一次回收中，Survivor + Eden中存活下来的内存超过了10%，则需要将一部分对象分配到老年代。

45.4.2. 年老代

对象如果在年轻代存活了足够长的时间而没有被清理掉（即在几次Young GC后存活了下来），则会被复制到年老代，年老代的空间一般比年轻代大，能存放更多的对象，在年老代上发生的GC次数也比年轻代少。当年老代内存不足时，将执行Major GC，也叫Full GC。

年老代的垃圾回收方式是标记－－整理算法。

45.4.3. 永久代

永久代的回收并不是必须的。

其回收有两种：常量池中的常量和无用的类信息。常量的回收很简单，没有引用了就可以被回收。对于无用的类进行回收，必须保证3点：

- 类的所有实例都已经被回收；
- 加载类的ClassLoader已经被回收；
- 类对象的Class对象没有被引用（即没有通过反射引用该类的地方）。

46. Java是值传递还是引用传递?

Java中方法参数传递方式是按值传递。

- 如果参数是基本类型，传递的是基本类型的字面量值的拷贝。
- 如果参数是引用类型，传递的是该参量所引用的对象在堆中地址值的拷贝。

47. 线程同步的方法

- 同步方法：synchronized关键字修饰的方法。
- 同步代码块：synchronized关键字修饰的语句块。
- volatile关键字。
- 可重入锁：ReentrantLock类是可重入、互斥、实现了Lock接口的锁。
- ThreadLocal。

48. Java创建线程的方式

48.1. 继承Thread类创建线程类

1. 定义Thread类的子类，并重写该类的 `run()` 方法，该 `run()` 方法的方法体就代表了线程要完成的任务。因此把 `run()` 方法称为执行体。
2. 创建Thread子类的实例，即创建了线程对象。
3. 调用线程对象的 `start()` 方法来启动该线程。

48.2. 通过Runnable接口创建线程类

1. 定义runnable接口的实现类，并重写该接口的 `run()` 方法，该 `run()` 方法的方法体同样是该线程的线程执行体。
2. 创建Runnable实现类的实例，并依此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。
3. 调用线程对象的 `start()` 方法来启动该线程。

48.3. 通过Callable和Future创建线程

1. 创建Callable接口的实现类，并实现 `call()` 方法，该 `call()` 方法将作为线程执行体，并且有返回值。
2. 创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的 `call()` 方法的返回值。
3. 使用FutureTask对象作为Thread对象的target创建并启动新线程。
4. 调用FutureTask对象的 `get()` 方法来获得子线程执行结束后的返回值。

49. 线程池原理

java.uitl.concurrent.ThreadPoolExecutor类是线程池中最核心的一个类，其构造函数为

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

- `corePoolSize`：核心池大小，默认情况下线程不会超过核心大小。
- `maximumPoolSize`：最大线程数，当达到一定负载时，线程数会超过核心数，但始终小于最大线程数。当负载较轻会回收线程至核心池数量。
- `keepAliveTime`：表示线程没有任务执行时，的存活时间。默认情况，当线程数大于核心小于最大数量时才会启用；如果调用allowCoreThreadTimeOut(boolean)方法，线程数下界为0。
- `unit`：`keepAliveTime` 的时间单位。
- `workQueue`：阻塞队列，用来存储等待执行的任务。
    - ArrayBlockingQueue：基于数组的先进先出，创建时必须指定大小。
    - PriorityBlockingQueue：优先级队列。
    - LinkedBlockingQueue：基于链表的队列，默认长度为Integer.MAX_VALUE。
    - SynchronousQueue：不保存任务，直接创建新线程。
- `threadFactory`：线程工厂。
- `handler`：对拒绝任务的处理策略，四种参数。
    - AbortPolicy：丢弃任务并抛异常。

- DiscardPolicy：丢弃任务不抛异常。
- DiscardOldestPolicy：丢弃最前面的任务。
- CallerRunsPolicy：交由调用线程处理。

运行流程：

1. 接收到任务。
2. 判断已存在线程数是否大于等于核心线程数，如果不是，则创建新线程执行任务；否则转3。
3. 判断任务队列是否有界，如果不是，将任务加入队列中；否则转4。
4. 判断任务队列是否已满，如果不是，将任务加入队列中；否则转5。
5. 判断已存在线程数是否等于最大线程数，如果不是，则创建新线程执行任务；否则转6。
6. 拒绝该任务。

参考：Java并发编程：线程池的使用 - 海 子 - 博客园

50. 类与类加载器的关系

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在Java虚拟机中的唯一性；每个类加载器，都拥有一个独立的类名称空间。

比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义。否则，即使这两个类来源于同一个class文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。

这里所指的“相等”，包括代表类的class对象的 `equals()` 方法、 `isAssignableFrom()` 方法、 `isInstance()` 方法的返回结果，也包括使用 `instanceof` 关键字做对象所属关系判定等情况。

51. 双亲委派模型

从Java虚拟机角度来看，只存在两种类加载器：

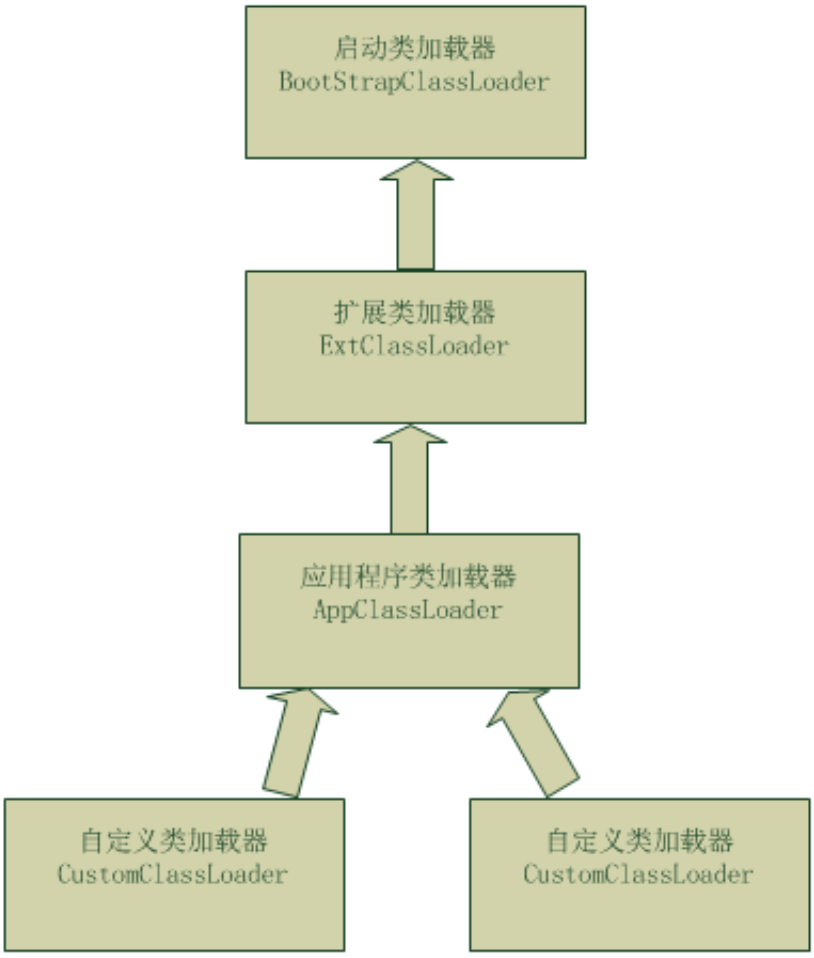
- 启动类加载器（Bootstrap ClassLoader）：这个类加载器使用C++语言实现，是虚拟机自身的一部分。
- 所有其他类加载器：这些类加载器都由Java语言实现，独立于虚拟机外部，并且全都继承自抽象类java.lang.ClassLoader。

从Java开发人员角度来看，绝大多数Java程序都会用到以下3种系统提供的类加载器：

- 启动类加载器（Bootstrap ClassLoader）：负责将存放在<JAVA_HOME>/lib目录中的，或者被-Xbootclasspath参数所指定的路径中的，并且是虚拟机识别的（仅按文件名识别，如rt.jar）类库加载到虚拟机内存中。启动类加载器无法被Java程序直接引用。
- 扩展类加载器（Extension ClassLoader）：由sun.misc.Launcher$ExtClassLoader实现，负责加载JAVA_HOME>/lib/ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库。开发者可以直接

使用扩展类加载器。

- 应用程序类加载器（Application ClassLoader）：由sun.misc.Launcher$AppClassLoader实现，由于这个类加载器是ClassLoader中的 `getSystemClassLoader()` 方法的返回值，所以一般也称它为系统类加载器。负责加载用户类路径（ClassPath)上所指定的类库。开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。



```
protected Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundExcept
ion {
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }

            if (c == null) {
                // If still not found, then invoke findClass in order
                // to find the class.
                c = findClass(name);
            }
        }
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }
}
```

双亲委派模型的工作过程是：如果一个类加载器收到了类加载请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此。因此，所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求时，子类加载器才会尝试自己去加载。

52. 如何自定义类加载器

继承ClassLoader类，实现 `findClass()` 方法，调用父类的 `defineClass()` 方法，返回加载后的类。

需要被加载的类：

```
package com.example.tomcat;

public class Foo {

    public void sayHi() {
        System.out.println("hello world");
    }
}
```

自定义类加载器：

```
package com.example.tomcat;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;

public class MyClassLoader extends ClassLoader {
    //类加载器的名称
    private String name;
    //类存放的路径
    private String classpath = "/demo/target/classes/com/example/tomcat/";

    MyClassLoader(String name) {
        this.name = name;
    }

    MyClassLoader(ClassLoader parent, String name) {
        super(parent);
        this.name = name;
    }

    /**
     * 重写findClass方法
     */
    @Override
    public Class<?> findClass(String name) {
        byte[] data = loadClassData(name);
        return this.defineClass(name, data, 0, data.length);
    }

    public byte[] loadClassData(String name) {
        try {
            name = name.replace(".", "//");
```

```
            FileInputStream is = new FileInputStream(new File(classpath + name + ".cl
ass"));
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            int b = 0;
            while ((b = is.read()) != -1) {
                baos.write(b);
            }
            return baos.toByteArray();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }

    public static void main(String args[]) throws Exception {
        MyClassLoader my = new MyClassLoader("myLoader");
        Class<?> loadClass = my.loadClass("com.example.tomcat.Foo");
        Foo cast = (Foo) loadClass.newInstance();
        cast.sayHi();
    }
}
```

53. 类的生命周期

- 加载（Loading）
- 连接（Linking）
  - 验证（Verification）
  - 准备（Preparation）
  - 解析（Resolution）
- 初始化（Initialization）
- 使用（Using）
- 卸载（Unloading）

54. 类初始化的时机

- 遇到new、getstatic、putstatic或invokestatic这4条字节码指令时，如果类没有进行过初始化，则需先触发其初始化。生成这4条指令的最常见的Java代码场景是：
  - 使用new关键字实例化对象的时候
  - 读取或设置一个类的静态字段的时候
  - 调用一个类的静态方法的时候

- 使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需先触发其初始化。
- 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需先触发其父类的初始化。
- 当虚拟机启动时，用户需要指定一个要执行的主类（包含 `main()` 方法的那个类），虚拟机会先初始化这个主类。

55. 类的加载过程

55.1. 加载

1. 通过一个类的全限定名来获取定义此类的二进制字节流。
2. 将这个字节流所代表的静态存储结构转换为方法区的运行时数据结构。
3. 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

55.2. 连接

55.2.1. 验证

验证的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。验证阶段大致完成4个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

文件格式验证

即验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理。主要包括以下验证点：

- 是否以魔术0xCAFEBABE开头。
- 主、次版本号是否在当前虚拟机处理范围之内。
- 常量池的常量中是否有不被支持的常量类型（检查常亮tag标识）。
- 指向常亮的各种索引值中是否有指向不存在的常量或不符合类型的常量。
- CONSTANT*Uft8*info型的常量中是否有不符合UTF8编码的数据。
- Class文件中各个部分及文件本身是否有被删除或附加的其他信息。
- ……

元数据验证

即对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言规范的要求。这个阶段可能包括以下验证点：

- 这个类是否有父类（除了java.lang.Object之外，所有的类都应该有父类）。
- 这个类的父类是否继承了不允许被继承的类（被final修饰的类）。
- 如果这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有方法。

- 类中的字段、方法是否与父类产生矛盾（如覆盖了父类的final字段，或出现不符合规则的方法重载，如方法参数一致，但返回值类型不同）。
- ……

字节码验证

即通过数据流河控制流分析，确定程序语义是合法的、符合逻辑的。这个阶段将对类的方法体进行校验分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的事件，如：

- 保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作，例如不会出现类似在操作栈放置了一个int类型的数据，使用时却按long类型来加载入本地变量表中。
- 保证跳转指令不会跳转到方法体以外的字节码指令上。
- 保证方法体中的类型转换是有效的，例如不能把父类对象赋值给子类数据类型。

符号引用验证

即对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验。发生在虚拟机将符号引用转化为直接引用的时候，通常包括以下几个验证点：

- 符号引用中通过字符串描述的全限定名是否能找到对应的类。
- 在指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段。
- 符号引用中的类、字段、方法的访问性（private、protected、public、default）是否可以被当前类访问。
- ……

55.2.2. 准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量使用的内存都将在方法区中进行分配。这时候进行内存分配的仅包括类变量（被static修饰的变量），而不包括示例变量。

55.2.3. 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

55.3. 初始化

初始化阶段，根据程序员通过程序制定的主观计划去初始化类变量和其他资源。即初始化阶段是执行类构造器 `<clinit>()` 方法的过程。

- `<clinit>()` 方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static {}块）中语句合并产生。编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静

态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但不能访问。

- `<clinit>()` 方法与类的构造函数（或者说实例构造器 `<init>()` 方法）不同，它不需要显式地调用父类构造器，虚拟机会保证在子类的 `<clinit>()` 方法执行之前，父类 `<clinit>()` 方法已经执行完毕。
- 由于父类的 `<clinit>()` 方法先执行，也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作。
- `<clinit>()` 方法对于类或接口来说并不是必须的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生成 `<clinit>()` 方法。
- 接口中不能使用静态语句块，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成 `<clinit>()` 方法。但接口与类不同的是，执行接口的 `<clinit>()` 方法不需要先执行父接口的 `<clinit>()` 方法。只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口的实现类在初始化时也一样不会执行接口的 `<clinit>()` 方法。
- 虚拟机会保证一个类的 `<clinit>()` 方法在多线程环境中被正确的加锁、同步，如果多个线程同时去初始化一个类，那么只有一个线程去执行这个类的 `<clinit>()` 方法，其他线程都需要阻塞等待，直到活动线程执行 `<clinit>()` 方法完毕。

56. 符号引用和直接引用的区别

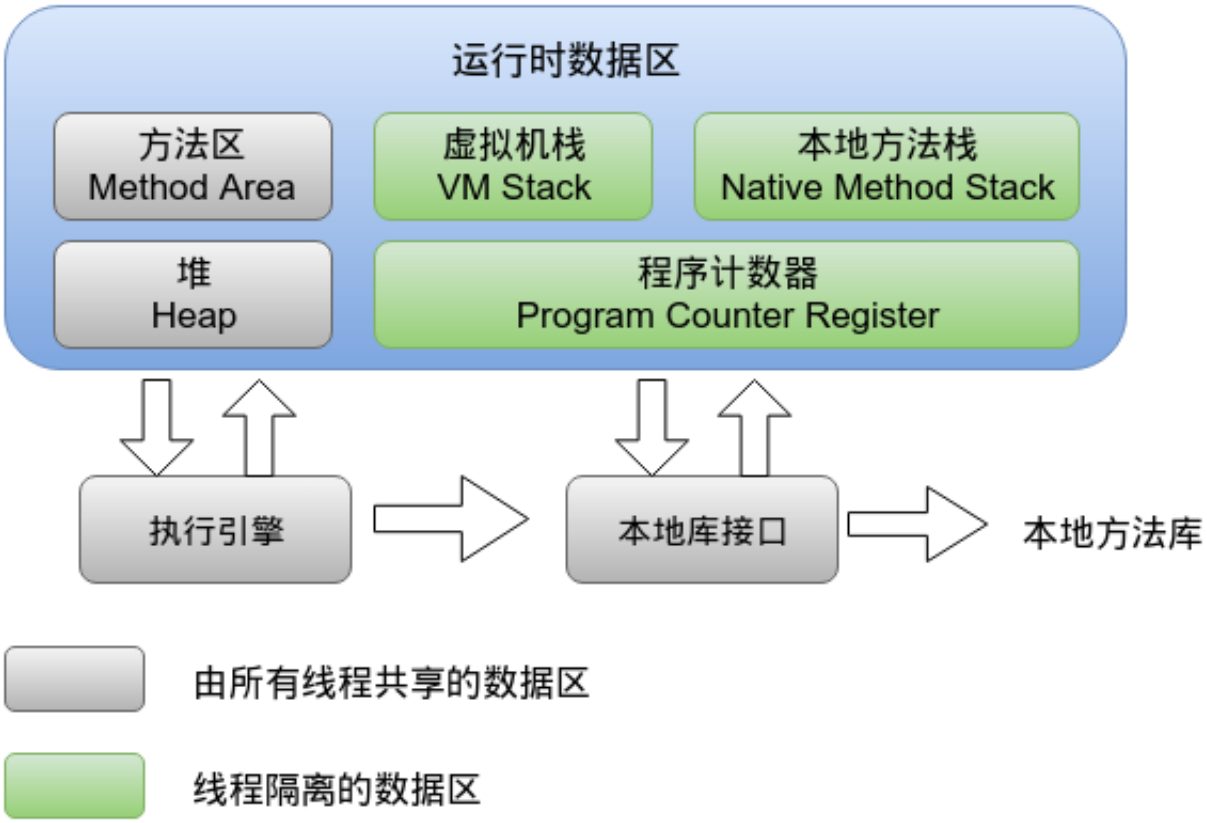
- 符号引用（Symbolic References）： 符号引用以一组符号来描述所引用的目标，符号可以是任意形式的字面量，只要使用时能无歧义地定位到目标即可。符号饮用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。
- 直接引用（Direct References）： 直接引用可以时直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是和虚拟机实现内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。

57. 如何理解平台无关性

虚拟机和字节码存储格式是实现语言无关性的基础。Java虚拟机不和包括Java在内的任何语言绑定，它只与“Class文件”这种特定的二进制文件格式所关联，Class文件中包含了Java虚拟机指令集和符号表以及若干其它辅助信息。任何一门功能性语言都可以表示为一个能被Java虚拟机所接受的有效的Class文件。

58. 运行时数据区域

Java虚拟机在执行Java程序过程中会把内存区域划分为若干个不同的数据区域，这些区域各有各自的用途、创建和销毁时间。有的区域随着虚拟机进程的启动而存在，有些区域则依赖用户线程的启动和结束而建立和销毁。



58.1. 程序计数器（Program Counter Register）

程序计数器占用较小的内存空间，可以看做是当前线程所执行的字节码的行号指示器，由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说就是一个内核）都只会执行一条线程中的指令。因此，为了线程切换后能够恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器。

如果线程正在执行Java方法，则计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Native方法，则这个计数器则为空。

58.2. Java虚拟机栈（Java Virtual Machine Stacks）

虚拟机栈也是线程私有，而且生命周期与线程相同，每个Java方法在执行的时候都会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。

Java虚拟机规范中，对该区域规定了两种异常情况：

- 如果线程请求的栈深度大于虚拟机所允许的深度，讲抛出StackOverflowError异常；
- 虚拟机栈可以动态拓展，当扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

58.3. 本地方法栈（Native Method Stack）

本地方法栈的作用与虚拟机栈作用是非常类似的。虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的Native方法服务。

58.4. Java堆（Java Heap）

对大多数应用来说，Java堆（Heap）是Java虚拟机所管理的内存中最大的一块，Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。该内存区域唯一的目的就是存放对象实例，Java对象实例以及数组都在堆上分配（随着JIT编译器发展等技术成熟，所有对象分配在堆上也渐渐不是那么“绝对”了）。

根据Java虚拟机规范的规定，Java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样（或者说，像链表一样虽然内存上不一定连续，但逻辑上是连续）。

58.5. 方法区（Method Area）

方法区与Java堆一样，是各个线程共享的内存区域，用于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。

58.6. 运行时常量池（Runtime Constant Pool）

运行时常量池是方法区的一部分。Class文件中除了有关类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

运行时常量池相对于Class文件常量池的另一个重要特征是具备动态性，Java语言并非不要求常量一定只有编译期才能产生，也就是并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可以将新的常量池放入池中。

59. Class文件结构

Class文件由顺序的8位字节为基础单位构成的二进制流。各个项目严格按照顺序紧凑排列，无分隔符。Class文件只有两种数据结构：无符号数和表。

无符号数属于基本的数据类型，以u1、u2、u4、u8分别代表1个、2个、4个、8个字节的无符号数。可以用来描述数字、索引引用、数量值或者按照UTF-8编码构成的字符串。

表由多个无符号数或其他表作为数据项构成的复合数据类型，所有表以“_info”结尾。表用来描述具有层次关系的复合结构数据。整个Class文件本质上就是一张表。

Class文件由魔数与Class文件的版本、常量池、访问标识、类索引、父类索引与接口索引集合、字段表集合、方法表集合、属性表集合等构成。

59.1. 魔数与Class文件的版本

每个Class文件的头4个字节称为魔数（Magic Number），它唯一作用就是用来确定文件是否能被虚拟机接受。

接下来的4个字节存储着Class文件的版本号，第五第六个字节为次版本号（Minor Version），第七第八为主版本号（Major Version）。版本号主要用于版本控制，高版本的JDK能向下兼容以前版本的Class文件，但不能运行以后版本的Class文件。

59.2. 常量池

紧接着版本号之后的就是常量池入口，常量池入口后面还必须有一个u2数据项作为常量池容量计数器（因为常量池数量不固定）。常量池是一个表类型的数据项，相当于Class文件的资源仓库，与Class文件其他项目关联最多，占用Class空间最大的数据项之一，且是第一个出现的表类型数据项目。

常量池主要存储两大类常量：

- 字面量（Literal）：相当于Java语言中的常量概念，比如字符串，声明为final的常量值。
- 符号引用（Symbolic References）：属于编译原理方面的概念包括三类常量：
  - 类和接口的全限定名（Fully Qualified Name）
  - 字段的名称和描述符（Descriptor）
  - 方法的名称和描述符

59.3. 访问标识

常量池之后就是由两个字节代表的访问标识（access flags），这些标识用于识别一些类或者接口层次的访问信息，包括：

- 这个Class是类还是接口；
- 是否定义为public；
- 是否定义为abstract类型；
- 如果是类的话，是否被final修饰。

59.4. 类索引、父类索引与接口索引集合

访问标志位之后就是u2类型的类索引，父类索引和接口索引集合。Class文件由这三项数据确定这个类的继承关系。这三项数据（u2类型的索引值）各指向类型为CONSTANT*Class*info的类描述符常量。

59.5. 字段表集合

字段表用于描述接口或者类中声明的变量。字段（field）包括类级变量以及实例级变量，但不包括在方法内部声明的局部变量。字段表中字段的各种描述信息（作用域比如public，private，是否被final，static修饰，是否可序

列化等）均使用标志位表示，名称则引用常量池中的常量来描述。

59.6. 方法表集合

在方法表中，方法的描述和字段的描述基本一致，依次包括访问标志（access*flags*）、*名称索引*（*name*index）、描述符索引（descriptor_index）、属性表集合（attributes）几项。

方法中的代码经过编译器编译成字节码指令后存放在方法属性表集合中一个名为“Code”的属性里面。

如果父类方法在子类中没有被重写，方法表集合中就不会出现来自父类的方法信息。

59.7. 属性表集合

Class文件、字段表、方法表都可以携带自己的属性表集合，以用于描述某些场景专有的信息。

为了能正确解析Class文件，在Java SE 7中预定义了21项属性，虚拟机在运行时会忽略他不认识的属性。

60. 运行时栈帧结构

栈帧（Stack Frame）是用于虚拟机执行时方法调用和方法执行时的数据结构，它是虚拟栈数据区的组成元素。每一个方法从调用到方法返回都对应着一个栈帧入栈出栈的过程。

每一个栈帧在编译程序代码的时候所需要多大的局部变量表，多深的操作数栈都已经决定了，并且写入到方发表的Code属性之中，一次一个栈帧需要多少内存，不会受到程序运行期变量数据的影响，仅仅取决于具体的虚拟机实现。

一个线程中方法调用可能很长，很多方法都处于执行状态。对于执行引擎来说，只有处于栈顶的栈帧才是有效的，称为当前栈帧（Current Stack Frame），与之相关联的方法称为当前方法（Current Method）。

在概念模型上，典型的栈帧主要由局部变量表（Local Stack Frame）、操作数栈（Operand Stack）、动态链接（Dynamic Linking）、返回地址（Return Address）组成，如下图所示：

图 8-1　栈帧的概念结构

60.1. 局部变量表（Local Variable Table）

局部变量表是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。在Java程序编译为Class文件时，就在方法的Code属性的max_locals数据项中确定了该方法所需要分配的局部变量表的最大容量。

在方法执行时，虚拟机是使用局部变量表完成参数值到参数变量列表的传递过程的，如果执行的是实例方法（非static的方法），那局部变量表中第0位索引的Slot默认是用于传递方法所属对象实例的引用，在方法中可以通过关键字“this”来访问到这个隐含的参数。其余参数则按照参数表顺序排列，占用从1开始的局部变量Slot，参数表分配完毕后，再根据方法体内部定义的变量顺序和作用域分配其余的Slot。

60.2. 操作数栈（Operand Stack）

操作数栈（Operand Stack）也常称为操作栈，是一个后入先出栈。在Class文件的Code属性的max_stacks指定

了执行过程中最大的栈深度。Java 虚拟机的解释执行引擎称为”基于栈的执行引擎“，这里的栈就是指操作数栈。

方法执行中进行算术运算或者是调用其他的方法进行参数传递的时候是通过操作数栈进行的。

在概念模型中，两个栈帧是相互独立的。但是大多数虚拟机的实现都会进行优化，令两个栈帧出现一部分重叠。令下面的部分操作数栈与上面的局部变量表重叠在一块，这样在方法调用的时候可以共用一部分数据，无需进行额外的参数复制传递。

60.3. 动态连接（Dynamic Linking）

每个栈帧都包含一个执行运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接（Dynamic Linking）。

Class文件中存放了大量的符号引用，字节码中的方法调用指令就是以常量池中指向方法的符号引用作为参数。这些符号引用一部分会在类加载阶段或第一次使用时转化为直接引用，这种转化称为静态解析。另一部分将在每一次运行期间转化为直接引用，这部分称为动态连接。

60.4. 方法返回地址

当一个方法开始执行以后，只有两种方法可以退出当前方法：

- 当执行遇到返回指令，会将返回值传递给上层的方法调用者，这种退出的方式称为正常完成出口（Normal Method Invocation Completion），一般来说，调用者的PC计数器可以作为返回地址。
- 当执行遇到异常，并且当前方法体内没有得到处理，就会导致方法退出，此时是没有返回值的，称为异常完成出口（Abrupt Method Invocation Completion），返回地址要通过异常处理器表来确定。

当方法返回时，可能进行3个操作：

- 恢复上层方法的局部变量表和操作数栈
- 把返回值压入调用者调用者栈帧的操作数栈
- 调整PC计数器的值以指向方法调用指令后面的一条指令

60.5. 附加信息

虚拟机规范并没有规定具体虚拟机实现包含什么附加信息，这部分的内容完全取决于具体实现。在实际开发中，一般会把动态连接，方法返回地址和附加信息全部归为一类，称为栈帧信息。

61. 基于栈的指令集和基于寄存器的指令集

61.1. 例子

分别使用基于栈的指令集和基于寄存器的指令集计算“1+1”的结果，基于栈的指令集会是：

```
iconst_1
iconst_1
iadd
istore_0
```

两条iconst*1指令连续把两个常量1压入栈后，iadd指令把栈顶的两个值出栈、相加，然后把结果放回栈顶，最后istore*0把栈顶的值放到局部变量表的第0个Slot中。

如果基于寄存器，那就会是：

```
mov eax, 1
add eax, 1
```

mov指令把EAX寄存器的值设为1，然后add指令再把这个值加1，结果就保存在EAX寄存器里面。

61.2. 区别

基于栈的指令集主要的优点就是可移植，寄存器由硬件直接提供，程序直接依赖这些硬件寄存器则不可避免地要受到硬件的约束。例如，现在32位80x86体系的处理器中提供了8个32位的寄存器，而ARM体系的CPU（在当前的手机、PDA中相当流行的一种处理器）则提供了16个32位的通用寄存器。如果使用栈架构的指令集，用户程序不会直接使用这些寄存器，就可以由虚拟机实现来自行决定把一些访问最频繁的数据（程序计数器、栈顶缓存等）放到寄存器中以获取尽量好的性能，这样实现起来也更加简单一些。栈架构的指令集还有一些其他的优点，如代码相对更加紧凑（字节码中每个字节就对应一条指令，而多地址指令集中还需要存放参数）、编译器实现更加简单（不需要考虑空间分配的问题，所需空间都在栈上操作）等。

栈架构指令集的主要缺点是执行速度相对来说会稍慢一些。所有主流物理机的指令集都是寄存器架构也从侧面印证了这一点。

虽然栈架构指令集的代码非常紧凑，但是完成相同功能所需的指令数量一般会比寄存器架构多，因为出栈、入栈操作本身就产生了相当多的指令数量。更重要的是，栈实现在内存之中，频繁的栈访问也就意味着频繁的内存访问，相对于处理器来说，内存始终是执行速度的瓶颈。尽管虚拟机可以采取栈顶缓存的手段，把最常用的操作映射到寄存器中避免直接内存访问，但这也只能是优化措施而不是解决本质问题的方法。由于指令数量和内存访问的原因，所以导致了栈架构指令集的执行速度会相对较慢。

62. Javac编译过程

Javac编译过程大致可以分为3个过程：

- 解析与填充符号表过程

- 插入式注解处理器的注解处理过程
- 分析与字节码生成过程

62.1. 解析与填充符号表过程

62.1.1. 解析（词法、语法分析）

词法分析是将源代码的字符流转变为标记（Token）集合，单个字符是程序编写过程的最小元素，而标记则是编译过程的最小元素，关键字、变量名、字面量、运算符都可以成为编辑，如“int a+b=2”这句代码中包含了6个标记，分别是int、a、=、b、+、2，虽然关键字int由三个字符构成，但是它只是一个Token,不可再拆分。

语法分析是根据Token序列构造抽象语法树的过程，抽象语法树是一种用来描述程序代码语法结构的树形表示方式，语法树的每一个节点都代表着程序代码中的一个语法结构，例如包、类型、修饰符、运算符、接口、返回值甚至代码注释等都可以是一个语法结构。

62.1.2. 填充符号表

符号表是由一组符号地址和符号信息构成的表格，可以把它想象成哈希表K-V值对的形式。符号表中所登记的信息在编译的不同阶段都要用到。在语义分析中，符号表所登记的内容将用于语义检测和产生中间代码。在目标代码生成阶段，当对符号名进行地址分配时，符号表是地址分配的依据。

62.2. 注解处理器

在JDK1.6中实现了JSR-269规范，提供了一组插入式注解处理器的标准API在编译期间对注解进行处理，我们可以把它看做是一组编译器的插件，在这些插件里面，可以读取、修改、添加抽象语法树中的任意元素。如果这些插件在处理注解期间对语法树进行了修改，那么编译器将回到解析及填充符号表的过程重新处理，直到所有的插入式注解处理器都没有再对语法树进行修改位置。

62.3. 语义分析与字节码生成

- 标注检查：标注检查步骤检查的内容包括诸如变量使用前是否已被声明、变量与赋值之间的数据类型是否能够匹配等。
- 数据及控制流分析：数据及控制流分析是对程序上下文逻辑更进一步的验证，它可以检查出诸如程序局部变量在使用前是否有赋值、方法的每条路径是否都有返回值、是否所有的受查异常都被正确处理了等问题。
- 解语法糖：Java中最常用的语法糖主要是泛型、变长参数、自动装箱／拆箱等，虚拟机运行时不支持这些

语法，它们在编译阶段被还原回简单的基础语法结构，这个过程称为解语法糖。
- 字节码生成：字节码生成是Javac编译过程的最后一个阶段。字节码生成阶段不仅仅是把前面各个步骤所生成的信息（语法树、符号表）转化成字节码写到磁盘中，编译器还进行了少量的代码添加和转换工作。

63. 什么是JIT

Java程序最初是通过解释器（Interpreter）进行解释执行的，当虚拟机发现某个方法或代码块的运行特别频繁时，就会把这些代码认定为是“热点代码”（Hot Spot Code）。为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，完成这个任务的编译器称为即时编译器（Just In Time Complier, JIT编译器）。

64. 什么是热点代码

在运行过程中会被即时编译器编译的“热点代码”有两类：

- 被多次调用的方法
- 被多次执行的循环体

65. 如何判断热点代码

判断一段代码是不是热点代码，是不是需要触发即时编译，这样的行为称为热点探测（Hot Spot Detection），目前主要的热点探测判定方式有两种：

- 基于采样的热点探测：采用这种方法的虚拟机会周期性地检查各个线程的栈顶，如果发现某些方法经常出现在栈顶，那这段方法代码就是“热点代码”。这种探测方法的好处是实现简单高效，还可以很容易地获取方法调用关系，缺点是很难精确地确认一个方法的热度，容易因为受到线程阻塞或别的外界因素的影响而扰乱热点探测。
- 基于计数器的热点探测：采用这种方法的虚拟机会为每个方法，甚至是代码块建立计数器，统计方法的执行次数，如果执行次数超过一定的阈值，就认为它是“热点方法”。这种统计方法实现复杂一些，需要为每个方法建立并维护计数器，而且不能直接获取到方法的调用关系，但是它的统计结果相对更加精确严谨。

66. Java与C／C++的编译器对比

Java虚拟机的即时编译器与C/C++的静态优化编译器相比，可能会由于下列原因，而导致输出的本地代码有一些劣势（下面列举的也包括一些虚拟机执行子系统的性能劣势）：

- 因为即时编译器运行占用的是用户程序的运行时间，具有很大的时间压力，它能提供的优化手段也严重受制于编译成本。如果编译速度达不到要求，那用户将在启动程序或程序的某部分察觉到重大延迟，这点使得即

时编译器不敢随便引入大规模的优化技术，而编译的时间成本在静态优化编译器中并不是主要的关注点。

- Java语言是动态的类型安全语言，这就意味着需要由虚拟机来确保程序不会违反语言语义或访问非结构化内存。从实现层面上看，这就意味着虚拟机必须频繁地进行动态检查，如实例方法访问时检查空指针、数组元素访问时检查上下界范围、类型转换时检查继承关系等。对于这类程序代码没有明确写出的检查行为，尽管编译器会努力进行优化，但是总体上仍然要消耗不少的运行时间。
- Java语言中虽然没有virtual关键字，但是使用虚方法的频率却远远大于C/C++语言，这意味着运行时对方法接收者进行多态选择的频率要远远大于C/C++语言，也意味着即时编译器在进行一些优化（如方法内联）时的难度要远远大于C/C++的静态优化编译器。
- Java语言是可以动态扩展的语言，运行时加载新的类可能改变程序类型的继承关系，这使得很多全局的优化难以进行，因为编译器无法看清程序的全貌，许多全局的优化都只能以激进优化的方式来完成，编译器不得不时刻注意并随着类型的变化而在运行时撤销或重新进行一些优化。
- Java语言的对象内存是在堆上，只有方法的局部变量才能在栈上分配，而C/C++的对象则有多重内存分配方式，既可能在堆上分配，又可能在栈上分配，如果可以在栈上分配线程私有的对象，将减轻内存回收的压力。另外，C/C++中主要由用户用程序代码来回收分配的内存，这就不存在无用对象筛选的过程，因此效率上（仅是运行效率，排除开发效率）也比Java的垃圾收集机制要高。

Java语言的这些性能上的劣势都是为了换取开发效率上的优势而付出的代价，动态安全、动态扩展、垃圾回收这些“拖后腿”的特性都为Java语言的开发效率作出了很大的贡献。Java编译器的另外一个红利时由它的动态性所带来的，由于C/C++编译器的所有优化都在编译期完成，以运行期性能监控为基础的优化措施它都无法进行，如调用频率预测（Call Frequency Prediction）、分支频率预测（Branch Frequency Prediction）、裁剪未被选择的分支（Untaken Branch Pruning）等，这些都会成为Java语言独有的性能优势。

67. Java线程的实现

操作系统实现线程主要有3种方式：

- 使用内核线程实现（一对一线程模型）
- 使用用户线程实现（一对多线程模型）
- 使用用户线程加轻量级进程混合实现（多对多线程模型）

Java线程在JDK 1.2之前，是基于称为“绿色线程”（Green Threads）的用户线程实现的；而在JDK 1.2种，线程模型替换为基于操作系统原生线程模型来实现。对于Sun JDK来说，它的Windows版与Linux版都是使用一对一的线程模型实现的，一条Java线程就映射到一条轻量级进程之中，因为Windwos和Linux系统提供的线程模型就是一对一的。

线程调度主要有两种方式：

- 协同式线程调度（Cooperative Threads-Scheduling）
- 抢占式线程调度（Preemptive Threads-Scheduling）

Java使用的线程调度方式是抢占式调度，由操作系统自动完成。

68. 线程的状态和转换关系

Java定义了5种线程状态，在任意一个时间点，一个线程只能有且只有其中一种状态：

- 新建（New）：创建了但未启动
- 运行（Runable）：包括了操作系统线程状态中的Running和Ready。处于此状态的线程有可能正在执行，也有可能正在等待着CPU为它分配执行时间。
- 无限期等待（Waiting）：处于这种状态的线程不会被分配CPU执行时间，它们要等待被其他线程显式地唤醒。以下方法会触发该状态：
  - 没有设置Timeout参数的 `Object.wait()` 方法。
  - 没有设置Timeout参数的 `Thread.join()` 方法。
  - `LockSupport.park()` 方法。

- 限期等待（Timed Waiting）：处于这种状态的线程也不会被分配CPU执行时间，但系统在一定时间后会自动唤醒它。以下方法会触发该状态：
  - `Thread.sleep()` 方法。
  - 设置Timeout参数的 `Object.wait()` 方法。
  - 设置Timeout参数的 `Thread.join()` 方法。
  - `LockSupport.parkNanos()` 方法。
  - `LockSupport.parkUntil()` 方法。

- 阻塞（Blocked）：阻塞状态下是在等待着获取一个排他锁，这个事件将在另外一个线程放弃这个锁的时候发生；在程序等待进入同步区域的时候，线程将进入这个状态。
- 结束（Terminated）：线程已经结束执行。

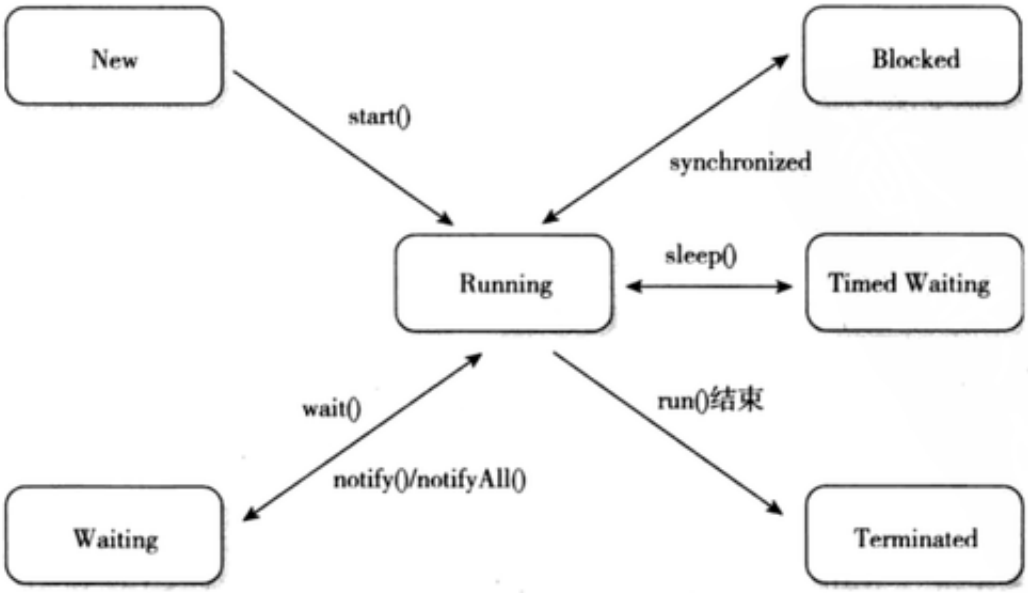
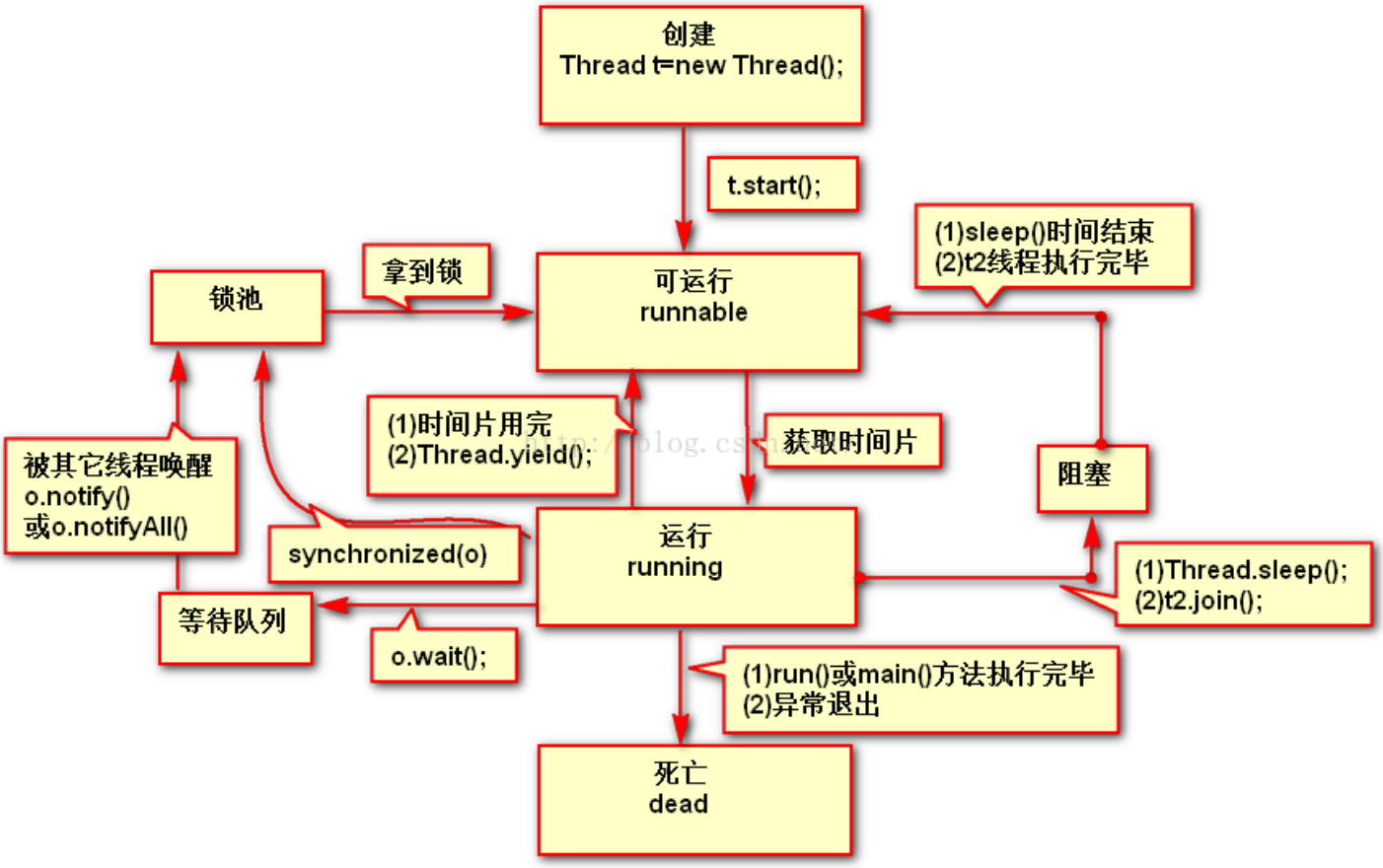


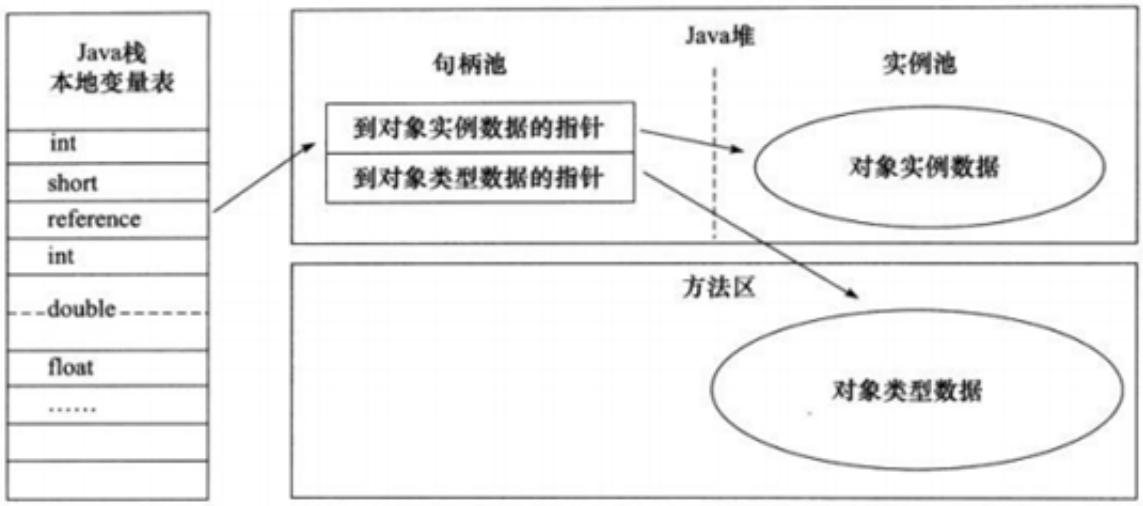
图 12-6　线程状态转换关系

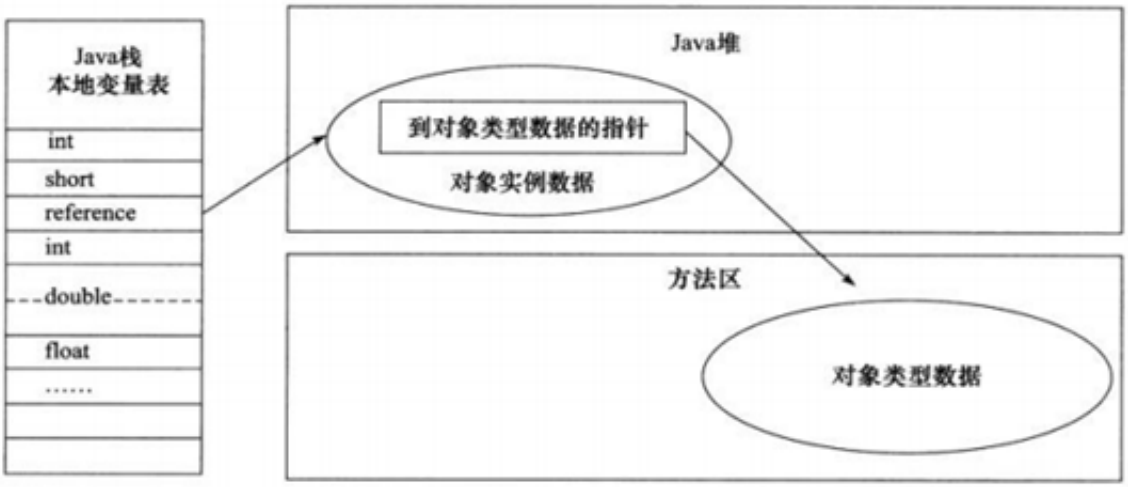


69. 对象的访问定位

目前主流的访问对象方式有两种：

- 使用句柄：Java堆划分出一块内存作为句柄池，reference种存储的是对象的句柄地址；而句柄中包含了对象实例数据和类型数据各自的地址。其优点是对象被移动（垃圾手机时移动对象是非常普遍的行为）时只会改变句柄中实例数据的地址，而reference本身不需要修改。
- 直接指针：reference中存储的直接就是对象实例数据的地址，而对象实例数据中需要有这个对象类型数据的地址。其优点是节省了一次指针定位的时间开销，速度更快。





70. Java有哪些语法糖

- 泛型与类型擦除
- 自动装箱、拆箱与遍历循环
- 条件编译

71. Java线程安全的实现

71.1. 互斥同步（Mutual Exclusion & Synchronization）

同步是指在多个线程并发访问共享数据时，保证共享数据在同一个时刻只被一个（或一些）线程使用。而互斥是实现同步的一种手段，临界区（Critical Section）、互斥量（Mutex）和信号量（Semaphore）都是主要的互斥实现方式。互斥是因，同步是果；互斥是方法，同步是目的。

最基本的互斥同步手段时synchronized关键字，synchronized关键字在编译后，会在同步块的前后分别形成monitorenter和monitorexit这两个字节码指令，这两个字节码都需要一个reference类型参数来指明要锁定和解锁的对象；如果没有指明，那就根据synchronized修饰的是实例方法还是类方法，去取对象的实例或Class对象来作为锁对象。

在执行monitorenter指令时，首先要尝试获取对象的锁。如果这个对象没被锁定，或者当前线程已经拥有了那个对象锁，则把锁的计数器加1；相应地，在monitorexit时，锁的计数器减1，当计数器减到0时，锁就被释放了。如果获取对象锁失败，则当前线程就要阻塞等待，直到对象锁被另一线程释放。

synchronized同步块对于同一条线程来说是可重入的，不会出现自己把自己锁死的情况；同步块在已进入的线程执行完前，会阻塞后面其他线程的进入。如果要阻塞或唤醒一个线程，都需要操作系统来帮忙完成，这就需要从用户态转换到和心态，因此状态转换需要耗费很多时间。

除synchronized外，还可以使用java.util.concurrent包中的重入锁（ReentrantLock）来实现同步。ReentrantLock表现为API层面的互斥锁（look()和unlock()方法），synchronized表现为原生语法层面的互斥锁。ReentrantLock还增加了以下高级功能：

- 等待可中断。当持有锁的线程长期不释放时，正在等待的线程可以选择放弃等待。
- 公平锁。多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁；而非公平锁则在锁被释放时，任何一个等待锁的线程都有机会获得锁。synchronized中的锁是非公平的，ReentrantLock默认是非公平的，但可通过设置使用公平锁。
- 锁可绑定多个条件。ReentrantLock对象可以同时绑定多个Condition对象。在synchronized中，锁对象的wait()和notify()或notifyAll()方法可以实现一个隐含的条件，如果要和多个条件关联，就需要添加额外的锁；而ReentrantLock则只需要多次调用newCondition()方法即可。

71.2. 非阻塞同步（Non-Blocking Synchronization）

通俗地说，就是先进行操作，如果没有其他线程争用共享数据，那操作就成功了；如果共享数据有争用，产生了冲突，那就再采取其他的补偿措施（最常见的补偿措施就是不断重试，直到成功为止）。

非阻塞同步需要硬件指令的支持，常用的有：

- 测试并设置（Test-and-Set）
- 获取并增加（Fetch-and-Increment）
- 交换（Swap）

- 比较和交换（Compare-and-Swap, CAS）
- 加载链接 / 条件存储（Load-Linked/Store-Conditional, LL/SC）

CAS指令需要3个操作数，分别是内存位置（V）、旧的预期值（A）和新值（B）。CAS指令执行时，当且仅当V符合旧预期值A时，处理器用新值B更新V的值，否则它就不更新。

CAS操作由sun.misc.Unsafe类中的compareAndSwapInt()和compareAndSwapLong()等几个方法包装提供，虚拟机即时编译出来的结果就是一条平台相关的处理器CAS指令。由于Unsafe类不是提供给用户程序调用的类，我们只能通过其他的API来间接使用；如java.util.concurrent中的AtomicInteger类，其中的compareAndSet()和getAndIncrement()等方法都使用了Unsafe类的CAS操作。

71.3. 无同步方案

如果一个方法本来就不涉及共享数据，那自然就无须任何同步措施去保证正确性。

- 可重入代码（Reentrant Code）：可以在代码执行的任何时刻中断它，转而去执行另外一段代码，而在控制权返回后，原来的程序不会出现任何错误。
- 线程本地存储（Thread Local Storage）：如果共享数据的可见范围限制在同一个线程之内，这样就无须同步也能保证线程之间不会出现数据争用的问题。

72. 锁优化

适应性自旋（Adaptive Spinning）、锁消除（Lock Elimination）、锁粗化（Lock Coarsening）、轻量级锁（Lightweight Locking）和偏向锁（Biased Locking）等，都是为了在线程之间更高效地共享数据，以解决竞争问题，提高程序执行效率。

72.1. 自旋锁与自适应自旋

在许多应用上，共享数据的锁定状态只会持续很短的一段时间，为了这段时间去挂起和恢复线程并不值得。如果物理机器有一个以上的处理器，能让两个或两个以上的线程同时并行执行，我们就可以让后面请求锁的那个线程“稍等一下”，但不放弃处理器的执行时间，看看持有锁的线程是否很快就会释放锁。为了让线程等待，我们只需让县城执行一个忙循环（自旋），这项技术就是所谓的自旋锁。

自旋等待虽然避免了线程切换的开销，但它要占用处理器时间。因此，如果锁被占用的时间很短，自旋等待效果就会非常好；反之，如果锁被占用时间很长，那么自旋的线程只会白白消耗处理器资源。因此，自旋等待时间是有限度的，如果自旋超过限定次数（默认10次）仍然没有成功获得锁，则会使用传统方式去挂起线程。

JDK 1.6中默认开启了自旋锁。

在JDK 1.6中引入了自适应自旋锁。自适应意味着自旋的时间不再固定了，而是由前一次在同一锁上的自旋时间及锁的拥有者的状态来决定。

72.2. 锁消除

锁消除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但被检测到不可能存在共享数据竞争的锁进行消除。锁消除的主要判定依据来源于逃逸分析的数据支持。

72.3. 锁粗化

如果虚拟机探测到有一串连续的操作都对同一个对象加锁，将会把加锁同步的范围扩大（粗化）到整个操作序列的外部，这样只需要加一次锁就可以了。

72.4. 轻量级锁

轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥产生的性能消耗。

HotSpot虚拟机的对象头（Object Header）分为两部分信息：第一部分用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄（Generational GC Age）等，这部分数据长度为32位或64位，称为“Mark Word”。其内容可以是：

| 存储内容 | 标志位 | 状态 | | 对象哈希码、对象分代年龄 | 01 | 未锁定 | 指向锁记录的指针 | 00 | 轻量级锁定 | | 指向重量级锁的指针 | 10 | 膨胀（重量级锁定） | | 空，不需要记录信息 | 11 | GC标记 | | 偏向线程ID、偏向时间戳、对象分代年龄 | 01 | 可偏向 |

在代码进入同步块的时候，如果此同步对象没有被锁定（锁标志位为“01”状态），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝（Displaced Mark Word）。

然后虚拟机使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针。如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位（Mark Word的最后2 bit）将转变为“00”，即表示此对象处于轻量级锁定状态。

如果这个更新操作失败了，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈，如果只说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行；否则说明这个锁对象已经被其他线程抢占了，这时轻量级锁膨胀为重量级锁，锁标志的状态值变为“10”，Mark Word中存储的就是指向重量级锁（互斥锁）得指针。

轻量级锁得解锁过程也是通过CAS操作来进行。如果对象的Mark Word仍然指向线程的锁记录，那就用CAS操作把对象当前的Mark Word和线程中复制的Displaced Mark Word替换回来。如果替换成功，则整个同步过程完成；否则说明有其他线程尝试获取该锁，则需要在释放锁的同时，唤醒被挂起的线程。

轻量级锁能提升程序同步性能的依据是：对于绝大部分的锁，在整个同步周期内度不是存在竞争的。

72.5. 偏向锁

偏向锁的目的是消除数据在无竞争情况下的同步原语，进一步提高程序的运行性能。如果说轻量级锁是在无竞争的情况下使用CAS操作去消除同步使用的互斥量，那偏向锁就是在无竞争的情况下把整个同步都消除掉，连CAS操作都不需要。

偏向锁的意思是这个锁会偏向于第一个获得它的线程，如果在接下来的执行过程中，该锁没有被其他的线程获取，则持有偏向锁的线程将永远不需要再进行同步。

当锁对象第一次被线程获取的时候，虚拟机会将会把对象头中的标志位设置为“01”，即偏向模式。同时使用CAS操作把获取到这个锁的线程ID记录在对象的Mark Word中，如果CAS操作成功，持有偏向锁的线程以后每次进入这个锁相关的同步块时，虚拟机都可以不再进行任何同步操作。

当有另外一个线程去尝试获取这个锁时，偏向模式就宣告结束。根据锁对象目前是否处于被锁定的状态，撤销偏向（Revoke Bias）后恢复到未锁定（标志位为“01”）或轻量级锁定（标志位为“00”）的状态，后续的同步操作如轻量级锁那样执行。

73. 如何理解Java是一门静态多分派且动态单分派的语言?

代码：

```
class Fruit {
}

class Apple extends Fruit {
}

class People {
    void eat(Fruit fruit) {
        System.out.println("People eat Fruit");
    }

    void eat(Apple apple) {
        System.out.println("People eat Apple");
    }
}

class Boy extends People {
    @Override
    void eat(Fruit fruit) {
        System.out.println("Boy eats Fruit");
    }

    @Override
    void eat(Apple apple) {
        System.out.println("Boy eats Apple");
    }
}
```

运行：

```
People boy = new Boy();
Fruit apple = new Apple();
boy.eat(apple);
```

结果：

```
Boy eats Fruit
```

74. 为什么synchronized修饰的变量推荐定义为final?

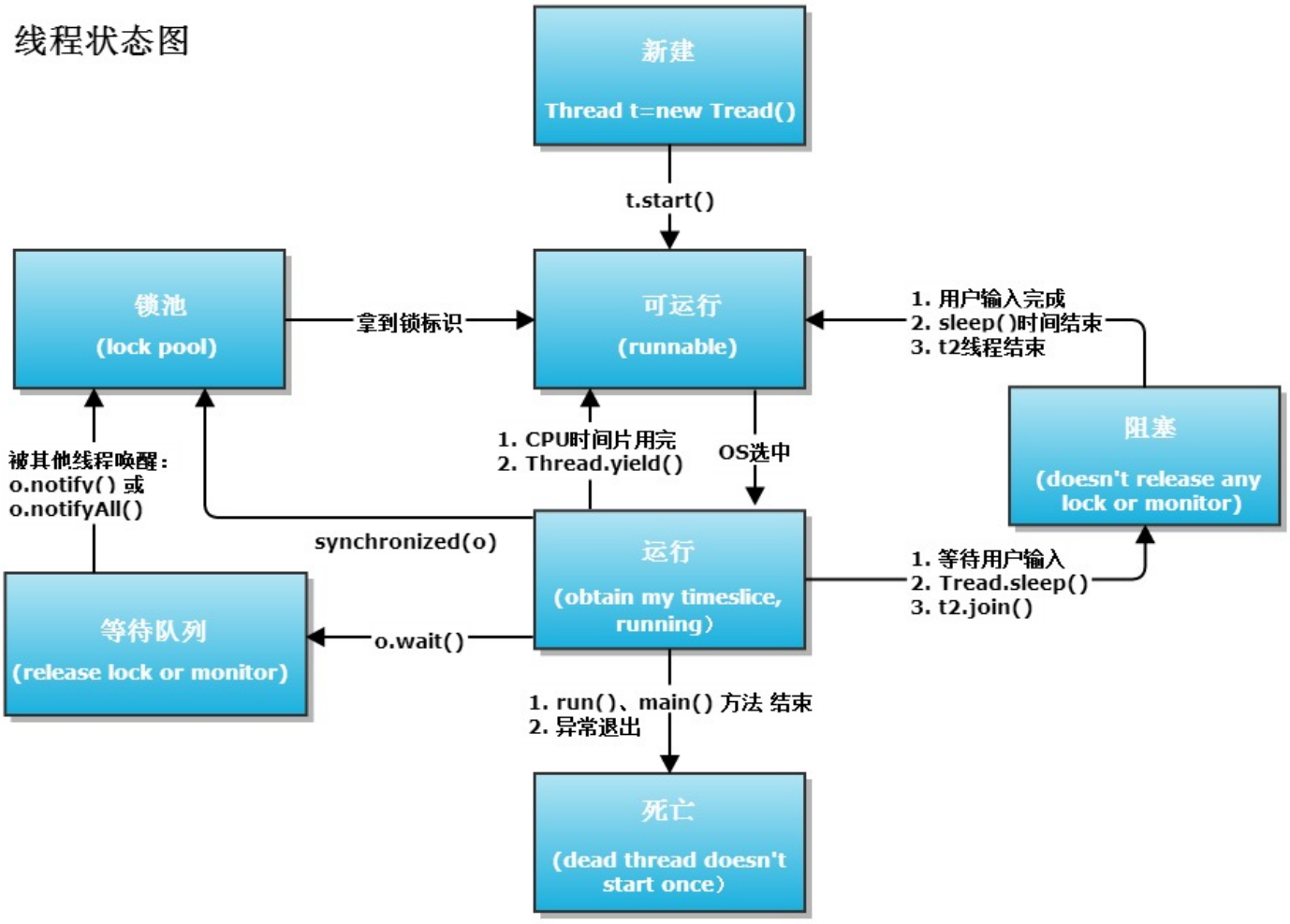
因为非final变量的引用常常会改变，一旦锁改变了，那synchronization就失去了意义。同时，也不推荐使用String对象作为synchronized代码块的锁，即使是 `final String` 。因为String存放在内存的String变量池中，

可能会有其他代码或者第三方的代码使用了同一个String对象为锁，这样容易导致一些无法预测的问题。

75. Object类有哪些方法

- `protected Object clone()`：创建并返回此对象的一个副本。
- `boolean equals(Object obj)`：指示某个其他对象是否与此对象“相等”。
- `protected void finalize()`：当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。
- `Class<? extendsObject> getClass()`：返回一个对象的运行时类。
- `int hashCode()`：返回该对象的哈希码值。
- `void notify()`：唤醒在此对象监视器上等待的单个线程。
- `void notifyAll()`：唤醒在此对象监视器上等待的所有线程。
- `String toString()`：返回该对象的字符串表示。
- `void wait()`：导致当前的线程等待，直到其他线程调用此对象的 `notify()` 方法或 `notifyAll()` 方法。
- `void wait(long timeout)`：导致当前的线程等待，直到其他线程调用此对象的 `notify()` 方法或 `notifyAll()` 方法，或者超过指定的时间量。
- `void wait(long timeout, int nanos)`：导致当前的线程等待，直到其他线程调用此对象的 `notify()` 方法或 `notifyAll()` 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量。

76. `sleep()` 和 `wait()` 的区别



- `sleep()` 是Thread类的方法， `wait()` 是Object类中定义的方法。
- `Thread.sleep()` 不会导致锁行为的改变，如果当前线程是拥有锁的，那么 `Thread.sleep()` 不会让线程释放锁。
- `Thread.sleep()` 和 `Object.wait()` 都会暂停当前的线程，对于CPU资源来说，不管是哪种方式暂停的线程，都表示它暂时不再需要CPU的执行时间。区别是，调用 `wait()` 后，需要别的线程执行 `notify()` / `notifyAll()` 才能够重新获得CPU执行时间。

77. Java版本历史与特性

77.1. Java 8

- Lambda表达式；
- Pipelines和Streams；
- Date和Time API；
- Default方法；

- Type注解；
- Nashhorn JavaScript引擎；
- 并发计数器；
- Parallel操作；
- 移除PermGen Error；
- TLS SNI。

77.2. Java 7

- switch语句块中允许以字符串作为分支条件；
- 在创建泛型对象时应用类型推断；
- 在一个语句块中捕获多种异常；
- 支持动态语言；
- 支持try-with-resources；
- 引入Java NIO.2开发包；
- 数值类型可以用2进制字符串表示，并且可以在字符串表示中添加下划线；
- 钻石型语法；
- null值的自动处理。

77.3. Java 6

- 支持脚本语言；
- 引入JDBC 4.0 API；
- 引入Java Compiler API；
- 可插拔注解；
- 增加对Native PKI(Public Key Infrastructure)、Java GSS(Generic Security Service)、Kerberos和LDAP(Lightweight Directory Access Protocol)的支持；
- 继承Web Services；
- 做了很多优化。

77.4. Java 5

- 引入泛型；
- 增强循环，可以使用迭代方式；
- 自动装箱与自动拆箱；
- 类型安全的枚举；
- 可变参数；
- 静态引入；
- 元数据（注解）；

- 引入Instrumentation。

78. ThreadLocal原理

ThreadLocal类似Map。

```
class Thread implements Runnable {
    ThreadLocal.ThreadLocalMap threadLocals = null;
}
```

```
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}
```

```
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}
```

```
void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

79. HashMap（Java 7）

79.1. 构造函数

主要是对如下几个变量的初始化：

```
int threshold;              // 所能容纳的key-value对极限
final float loadFactor;     // 负载因子
int modCount;
int size;
```

- `loadFactor`：Load factor为负载因子(默认值是0.75)。
- `threshold`：`Entry`数组初始化长度`length`默认值为16，`threshold`是HashMap所能容纳的

最大数据量的 `Entry` (键值对)个数。 `threshold = length * Load factor` 。

- `size` ： HashMap中实际存在的键值对数量。
- `modCount` ： 主要用来记录HashMap内部结构发生变化的次数，主要用于迭代的快速失败。内部结构发生变化指的是结构发生变化，例如 `put()` 新键值对，但是某个key对应的value值被覆盖不属于结构变化。

79.2. 确定索引位置

HashMap定位数组索引位置，直接决定了 `hash()` 方法的离散性能。Hash算法本质上就是三步：取key的hashCode值、高位运算、取模运算。

```
int hash(Object k) {
    int h = k.hashCode();
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

```
int indexFor(int h, int length) {
    return h & (length-1);
}
```

`hash()` 主要是用来“扰动”， `indexFor()` 直接取hash的低位作为数组索引，所以扰动的目的就是混合 `hashCode()` 的高位和低位，以此来加大低位的随机性。

79.3. `put()`

```
public V put(K key, V value) {
    // 1. table为空则创建
    if (table == EMPTY_TABLE) {
        inflateTable(threshold);
    }
    // 2. key为null则单独处理
    if (key == null)
        return putForNullKey(value);
    // 3. 计算hash并得到数组索引
    int hash = hash(key);
    int i = indexFor(hash, table.length);
    // 4. key存在则直接覆盖value
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }

    modCount++;
    // 5. key不存在则添加
    addEntry(hash, key, value, i);
    return null;
}
```

79.3.1. `inflateTable()`

```
private void inflateTable(int toSize) {
    int capacity = roundUpToPowerOf2(toSize);

    threshold = (int) Math.min(capacity * loadFactor, MAXIMUM_CAPACITY + 1);
    table = new Entry[capacity];
}
```

`inflateTable()` 创建table，table大小永远是2的幂次，是为了计算 `indexFor()` 方便。

79.3.2. `putForNullKey()`

`putForNullKey()` 与正常的 `put()` 非常相似，只不过将数组索引指定为0。

79.3.3. `addEntry()`

```
void addEntry(int hash, K key, V value, int bucketIndex) {
    if ((size >= threshold) && (null != table[bucketIndex])) {
        resize(2 * table.length);
        hash = (null != key) ? hash(key) : 0;
        bucketIndex = indexFor(hash, table.length);
    }

    createEntry(hash, key, value, bucketIndex);
}
```

```
void createEntry(int hash, K key, V value, int bucketIndex) {
    Entry<K,V> e = table[bucketIndex];
    table[bucketIndex] = new Entry<>(hash, key, value, e);
    size++;
}
```

```
Entry(int h, K k, V v, Entry<K,V> n) {
    value = v;
    next = n;
    key = k;
    hash = h;
}
```

`addEntry()` 首先判断是否需要扩容（`size >= threshold`），若需要则首先 `resize()` 扩容，重新计算数组索引，最后 `createEntry()` 插入到table中。 `createEntry()` 即采用头插法将新的 `Entry` 插入到table中。

79.4. 扩容机制

```
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;
    }

    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable);
    table = newTable;
    threshold = (int)Math.min(newCapacity * loadFactor, MAXIMUM_CAPACITY + 1);
}
```

```
void transfer(Entry[] newTable) {
    int newCapacity = newTable.length;
    for (Entry<K,V> e : table) {
        while(null != e) {
            Entry<K,V> next = e.next;
            e.hash = null == e.key ? 0 : hash(e.key);
            int i = indexFor(e.hash, newCapacity);
            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        }
    }
}
```

`resize()` 创建一个大小为 `2*table.length` 的 `Entry` 数组，然后通过 `transfer()` 将原 `Entry` 数组中的元素重新hash到新的 `Entry` 数组中。这里插入到新table仍然采用头插法。

79.5. `get()`

```
public V get(Object key) {
    if (key == null)
        return getForNullKey();
    Entry<K,V> entry = getEntry(key);

    return null == entry ? null : entry.getValue();
}
```

```
private V getForNullKey() {
    if (size == 0) {
        return null;
    }
    for (Entry<K,V> e = table[0]; e != null; e = e.next) {
        if (e.key == null)
            return e.value;
    }
    return null;
}
```

```
final Entry<K,V> getEntry(Object key) {
    if (size == 0) {
        return null;
    }

    int hash = (key == null) ? 0 : hash(key);
    for (Entry<K,V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))
            return e;
    }
    return null;
}
```

`get()`时，通过key找到入口`Entry`，再采用单链表遍历的方式找到真正的Entry（`e.hash == hash && (e.key == key || (key != null && key.equals(e.key))`），最后返回value。

参考：Java 8系列之重新认识HashMap

80. Java 8对HashMap的改进

80.1. `hash()`

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

Java 8中`hash()`仅扰动一次，而Java 7中扰动四次。

80.2. 红黑树

Java 7中HashMap采用的是位桶+链表的方式。而Java 8中采用的是位桶+链表 / 红黑树的方式，当某个位桶的链表的长度超过8的时候，这个链表就将转换成红黑树。

80.3. `resize()`

Java 7在扩容时会重新计算 `Entry` 的数组索引，而在Java 8中只需要看看原来的hash值新增的那个bit是1还是0就好了（table数组大小每次扩容乘2），是0的话索引没变，是1的话索引变成“原索引+oldCap”。

81. 如何理解NIO

81.1. 什么是NIO

NIO包（ `java.nio.*` ）引入了四个关键的抽象数据类型，它们共同解决传统的I/O类中的一些问题：

- `Buffer` ： 它是包含数据且用于读写的线形表结构。其中还提供了一个特殊类用于内存映射文件的I/O操作。
- `Charset` ： 它提供Unicode字符串映射到字节序列以及逆映射的操作。
- `Channels` ： 包含socket，file和pipe三种管道，它实际上是双向交流的通道。
- `Selector` ： 它将多元异步I/O操作集中到一个或多个线程中（它可以被看成是Unix中 `select()` 函数或Win32中 `WaitForSingleEvent()` 函数的面向对象版本）。

81.2. NIO与IO的区别

Java NIO和IO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

Java IO的各种流是阻塞的。这意味着，当一个线程调用 `read()` 或 `write()` 时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

81.3. 为什么要使用NIO

NIO 的创建目的是为了让Java程序员可以实现高速I/O而无需编写自定义的本机代码。NIO将最耗时的I/O操作(即填充和提取缓冲区)转移回操作系统，因而可以极大地提高速度。

82. concurrent包

concurrent包主要包含：

- 大部分关于并发的接口和类：BlockingQueue、Callable、ConcurrentMap、Executor、ExecutorService、Future、Semaphore等。
- 所有原子操作的类：AtomicInteger、AtomicLong等。
- 锁相关的类：Lock、ReentrantLock、ReadWriteLock等。

83. 当前线程 `wait()` 后会立即阻塞吗？其他线程能够进入同步块吗?

当调用 `wait()` 时，当前线程会放弃已经获得的锁，接着会将自己park住，放弃CPU。而在notify()中会选择一个 `wait()` 的线程进行unpark，被unpark的线程还需要竞争锁。

84. 为何调用 `wait()` 可能抛出InterruptedException异常?

当调用线程的 `interrupt()` 方法时会抛出InterruptedException，因此即使当前线程因 `wait()` 一直被阻塞，当被唤醒时也会去检查其状态，如果其被interrupt了，就会抛出InterruptedException。

85. 调用 `notify()` 后等待的线程会被立刻唤醒吗?

有不同的策略。默认策略是调用 `notify()` 会将一个等待队列中的线程放到锁池中，等到退出同步块时再释放锁，由锁池中的线程竞争。这里“唤醒”的定义不明确，可以说“线程由等待队列移动到锁池”是唤醒，也可以说“线程得到CPU时间”是唤醒。

86. `notify()` 和 `notifyAll()` 有什么区别?

注意： `synchronized()` 会使线程进入锁池， `wait()` 会使线程进入等待队列。只有锁池中的线程会竞争锁，等待队列中的线程不会竞争。

`notify()` 会将**一个**等待队列中的线程移动到锁池中， `notifyAll()` 则会将**所有**等待队列中的线程移动到

锁池中。

87. `notify()` 可能引发死锁。

```
class PubSub {
    boolean flag;
    int count;

    synchronized void pub() throws InterruptedException {
        while (!flag) {
            wait();
        }
        flag = false;
        count++;
        notify();
        System.out.println("pub count " + count);
    }

    synchronized void sub() throws InterruptedException {
        while (flag) {
            wait();
        }
        flag = true;
        count++;
        notify();
        System.out.println("sub count " + count);
    }
}

public class Main {
    public static void main(String[] args) {
        final PubSub pb = new PubSub();
        for (int i = 0; i < 5; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    while (true) {
                        try {
                            pb.sub();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
```

```
            }).start();

            new Thread(new Runnable() {
                @Override
                public void run() {
                    while (true) {
                        try {
                            pb.pub();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }).start();
        }
    }
}
```

如上代码会发生死锁。`synchronized()`会使线程进入锁池，`wait()`会使线程进入等待队列，`notify()`会将一个等待队列中的线程移动到锁池中。如上代码会出现多个多次调用`wait()`导致所有线程全部处于等待队列，而无线程在锁池中的情况，导致死锁。

88. 线程的`sleep()`、`yield()`和`join()`有什么区别?

- `sleep()`：线程进入阻塞状态；
- `yield()`：线程进入就绪状态；
- `join()`：线程进入阻塞状态。

89. `类名.class`与`类名.this`的区别

`类名.this`的语法在Java语言中叫做“qualified this”。这个语法的主要用途是：在内部类的方法中，要指定某个嵌套层次的外围类的`this`引用时，使用`外围类名.this`语法。

`类型名.class`的语法在Java语言中叫做“Class Literal”，类字面量。类字面量的类型是`java.lang.Class<类型名>`。例如说`Foo.class`的类型就是`Class<Foo>`，是一个引用，指向Foo类唯一对应的那个Class对象。当需要通过Class对象来做一些反射操作的时候，常常会用到类字面量的语法。