常量池、栈、堆的比较

JAVA中，有六个不同的地方可以存储数据，对于栈和常量池中的对象可以共享，对于堆中的对象不可以共享。

栈

存放基本类型的变量数据和对象的引用。JAVA编译器需知道存储在栈内所有数据的大小和生命周期。当没有引用指向数据时，这个数据就会消失。

堆

存放所有的JAVA对象，即使用new关键字创建的对象，堆进行存储需要一定的时间。堆中的对象的由垃圾回收器负责回收，因此大小和生命周期不需要确定，具有很大的灵活性。

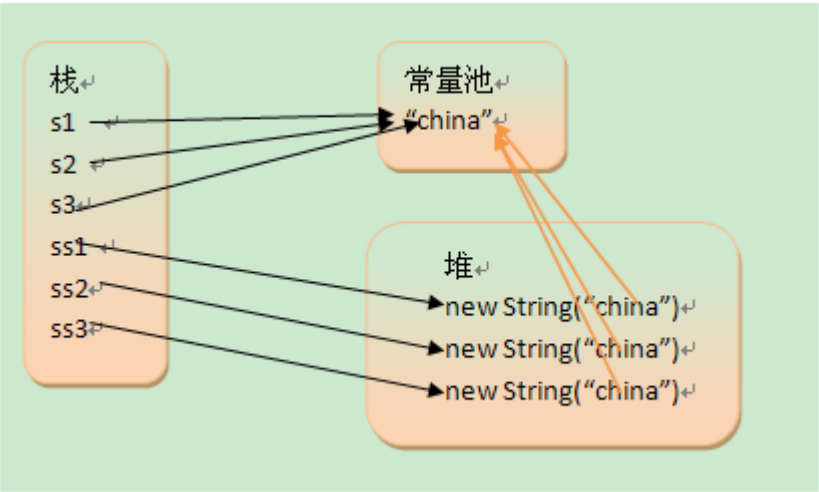
静态域

存放静态成员（static定义的）

常量池

存放字符串常量和基本类型常量（public static final）。String s1 = "china"; 这种创建的数据，就存储在常量池中，String ss1 = new String("china");存储在堆中，对于equals相等的字符串，在常量池中永远只有一份，在堆中有多份。

```
String s1 = "china";
String s2 = "china";
String s3 = "china";
String ss1 = new String("china");
String ss2 = new String("china");
String ss3 = new String("china");
```

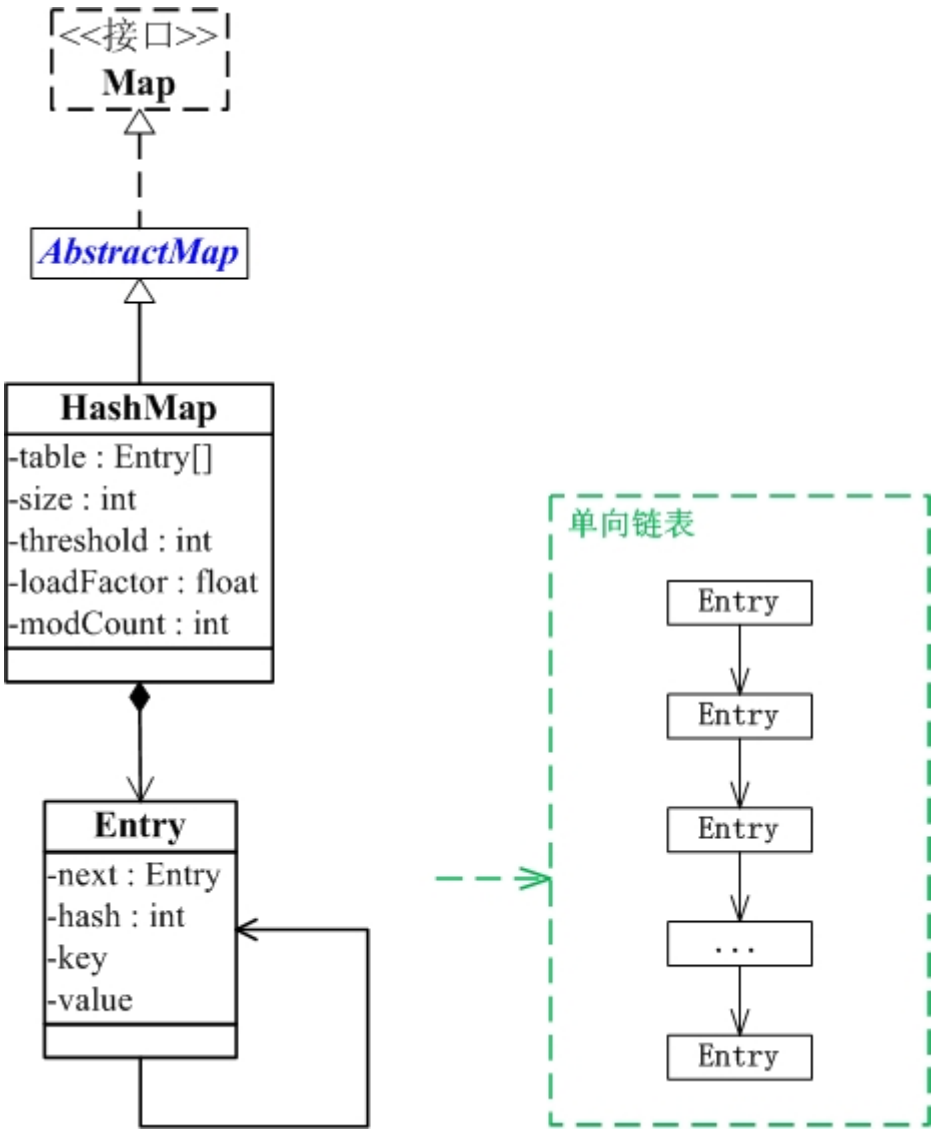


HashMap

HashMap 是一个散列表，存储的内容是键值对(key-value)映射。HashMap中的映射是无序的。线程不安全。key、value都可以为null。**通过“拉链法”解决哈希冲突的。**

集成-实现关系

```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable,
Serializable
```



构造函数

```
// 默认构造函数。
HashMap()
    //使用默认的初始容量
   static final int DEFAULT_INITIAL_CAPACITY = 16;
   // 默认加载因子
   static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

```
// 指定“容量大小”的构造函数
HashMap(int capacity)
    // 默认加载因子
   static final float DEFAULT_LOAD_FACTOR = 0.75f;
// 指定“容量大小”和“加载因子”的构造函数
HashMap(int capacity, float loadFactor)
// 包含“子Map”的构造函数
HashMap(Map<? extends K, ? extends V> map)
```

重要的成员变量

table 是一个Entry[]数组类型，而Entry实际上就是一个单向链表。哈希表的"key-value键值对"都是存储在Entry数组中的。 **size**是HashMap的大小，它是HashMap保存的键值对的数量。 **threshold**是HashMap的阈值，用于判断是否需要调整HashMap的容量。threshold的值="容量*加载因子"，当HashMap中存储数据的数量达到threshold时，就需要将HashMap的容量加倍。 **loadFactor**就是加载因子。 **modCount**是用来实现fail-fast机制的。

API

```
void                clear() //通过将所有的元素设为null来实现
Object              clone()
//首先通过getEntry(key)获取key对应的Entry，getEntry(key)判断key的hashcode在table中是否存在，如果存在则判断存在table中的key与当前key是否相等
boolean             containsKey(Object key)
boolean             containsValue(Object value)
Set<Entry<K, V>>    entrySet()
V                   get(Object key)
boolean             isEmpty()
Set<K>              keySet()
V                   put(K key, V value)
void                putAll(Map<? extends K, ? extends V> map)
V                   remove(Object key)
int                 size()
Collection<V>       values()
```

造成哈希冲突原因

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

public V put(K key, V value) {
        // 若“key为null”，则将该键值对添加到table[0]中。
        if (key == null)
            return putForNullKey(value);
        // 若“key不为null”，则计算该key的哈希值，然后将其添加到该哈希值对应的链表中。
        int hash = hash(key.hashCode());
        int i = indexFor(hash, table.length);
        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
```

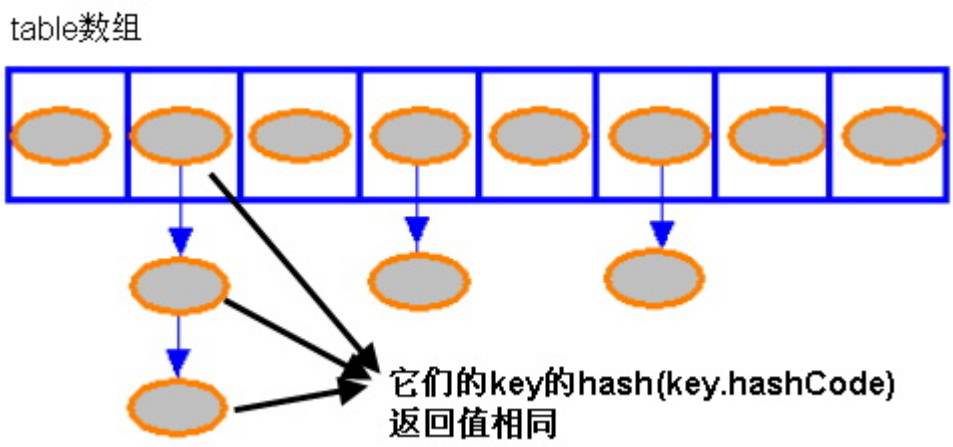
```
            Object k;
            // 若“该key”对应的键值对已经存在，则用新的value取代旧的value。然后退出！
            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
            }
        }

        // 若“该key”对应的键值对不存在，则将“key-value”添加到table中
        modCount++;
        addEntry(hash, key, value, i);
        return null;
    }


    // 新增Entry。将“key-value”插入指定位置，bucketIndex是位置索引。
    void addEntry(int hash, K key, V value, int bucketIndex) {
        // 保存“bucketIndex”位置的值到“e”中
        Entry<K,V> e = table[bucketIndex];
        // 设置“bucketIndex”位置的元素为“新Entry”，
        // 设置“e”为“新Entry的下一个节点”
        table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
        // 若HashMap的实际大小 不小于 “阈值”，则调整HashMap的大小
        if (size++ >= threshold)
            resize(2 * table.length);
    }
```

在hashmap中添加数据时，即put(K key,V value)，HashMap根据key值的hashcode来决定将Entry实例存放在table中的哪个位置【 int i = indexFor(hash, table.length);】，当key的hashcode相同时，，则会出现hash冲突。

使用单链表解决hash冲突的问题，当hashcode存在时，将当前bucketIndex的值存放在新创建的Entry的链表上，然后将新创建的Entry放在 table[bucketIndex] 中。



http安全性幂等行

安全性

一次请求或多次请求对数据资源不造成影响，即为安全

例如：GET GET请求获取数据资源，只是查看而不对数据进行操作即为安全。 DELETE DELETE请求删除数据资源，对数据资源造成了影响，即为不安全，同理POST PUT等 。

幂等性

一次请求或多次相同请求所达到的目的(对数据造成的影响)一致.

请求方式	请求地址	描述	幂等
GET	/emp	多次获取相同数据，不会对数据造成影响	幂等
PUT	/emp/5	对要修改的数据进行修改，一次多次执行修改对数据的影响相同，目的是将该条数据修改为请求数据	幂等
DELETE	/emp/5	对要删除的数据进行删除，一次多次执行删除对数据的影响相同，目的是将该条数据删除	幂等
POST	/emp	插入数据，一次执行及插入一条数据，多次执行则插入多条，对数据造成的影响不同	不幂等

ThreadPoolExecutor

类构造器

```
public class ThreadPoolExecutor extends AbstractExecutorService {
        .....
        public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long
keepAliveTime,TimeUnit unit,
                BlockingQueue<Runnable> workQueue);

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

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

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

构造器参数说明

corePoolSize：核心线程数，默认情况下,在创建了线程池后，线程池中的线程数为0,task1来请求,会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；

maximumPoolSize：线程池最大线程数，在线程池中最多能创建多少个线程

keepAliveTime：线程存活时间，表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用。但是如果调用allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；

workQueue：阻塞队列，用来存储等待执行的任务。一般使用LinkedBlockingQueue和SynchronousQueue。

threadFactory：线程工厂，主要用来创建线程

handler：表示拒绝处理任务时的策略，有以下四种取值：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。(常用)

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

继承关系

```
ThreadPoolExecutor extends AbstractExecutorService 、AbstractExecutorService implements
ExecutorService、 ExecutorService extends Executor
    public interface Executor {
        void execute(Runnable command);
    }
```

execute()方法：在ThreadPoolExecutor进行了具体的实现，通过这个方法可以向线程池提交一个任务，交由线程池去执行。

submit()方法：在AbstractExecutorService进行具体的实现，这个方法也是用来向线程池提交任务的，实际上是调用的execute()方法，它利用了Future来获取任务执行结果。

线程池状态

状态	描述
RUNNING	创建线程池初始状态
SHUTDOWN	调用了shutdown()方法，此时线程池不能够接受新的任务，它会等待所有任务执行完毕
STOP	调用了shutdownNow()方法，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务；
TERMINATED	当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态

重要成员变量

```
private final BlockingQueue<Runnable> workQueue;              //任务缓存队列，用来存放等待执行的任务
    private final ReentrantLock mainLock = new ReentrantLock();    //线程池的主要状态锁，对线程池状态（比如线程池大小、runState等）的改变都要使用这个锁
    private final HashSet<Worker> workers = new HashSet<Worker>();  //用来存放工作集

    private volatile long  keepAliveTime;    //线程存活时间
    private volatile boolean allowCoreThreadTimeOut;   //是否允许为核心线程设置存活时间
    private volatile int   corePoolSize;     //核心池的大小（即线程池中的线程数目大于这个参数时，提交的任务会被放进任务缓存队列）
    private volatile int   maximumPoolSize;   //线程池最大能容忍的线程数

    private volatile int   poolSize;       //线程池中当前的线程数

    private volatile RejectedExecutionHandler handler; //任务拒绝策略

    private volatile ThreadFactory threadFactory;   //线程工厂，用来创建线程

    private int largestPoolSize;   //用来记录线程池中曾经出现过的最大线程数

    private long completedTaskCount;   //用来记录已经执行完毕的任务个数
```

任务缓存队列及排队策略

workQueue的类型为BlockingQueue，通常可以取下面三种类型：

1）ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；

2）LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX_VALUE；

3）synchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

ThreadPoolExecutor运行原理

corePoolSize maximumPoolSize task(任务总数) poolSize(当前线程总数)

poolSize < corePoolSize 时， 当任务处理慢时，则创建新的线程处理新的task

poolSize == corePoolSize 时，将(task-corePoolSize)的任务放入workQueue中，等待处理

task > corePoolSize+workQueue 时,当新任务加入workQueue失败时(任务缓存队列已满),则将线程数增大至maximumPoolSize来处理添加缓存失败的任务

task > maximumPoolSize+workQueue 根据拒绝策略执行

TCP和UDP

TCP

Transmission Control Protocol，传输控制协议，是面向连接的协议。TCP连接时，三次握手，断开时，四次挥手。三次握手（类似打电话），A：喂 （此时A不能确定B是否能接通），B：喂（此时B听到了A的谈话，但不能确定自己说的A是否能正常接收到），A："B，你好"（此时A收到了B的回复，给B回复能正常接听），谈话开始.

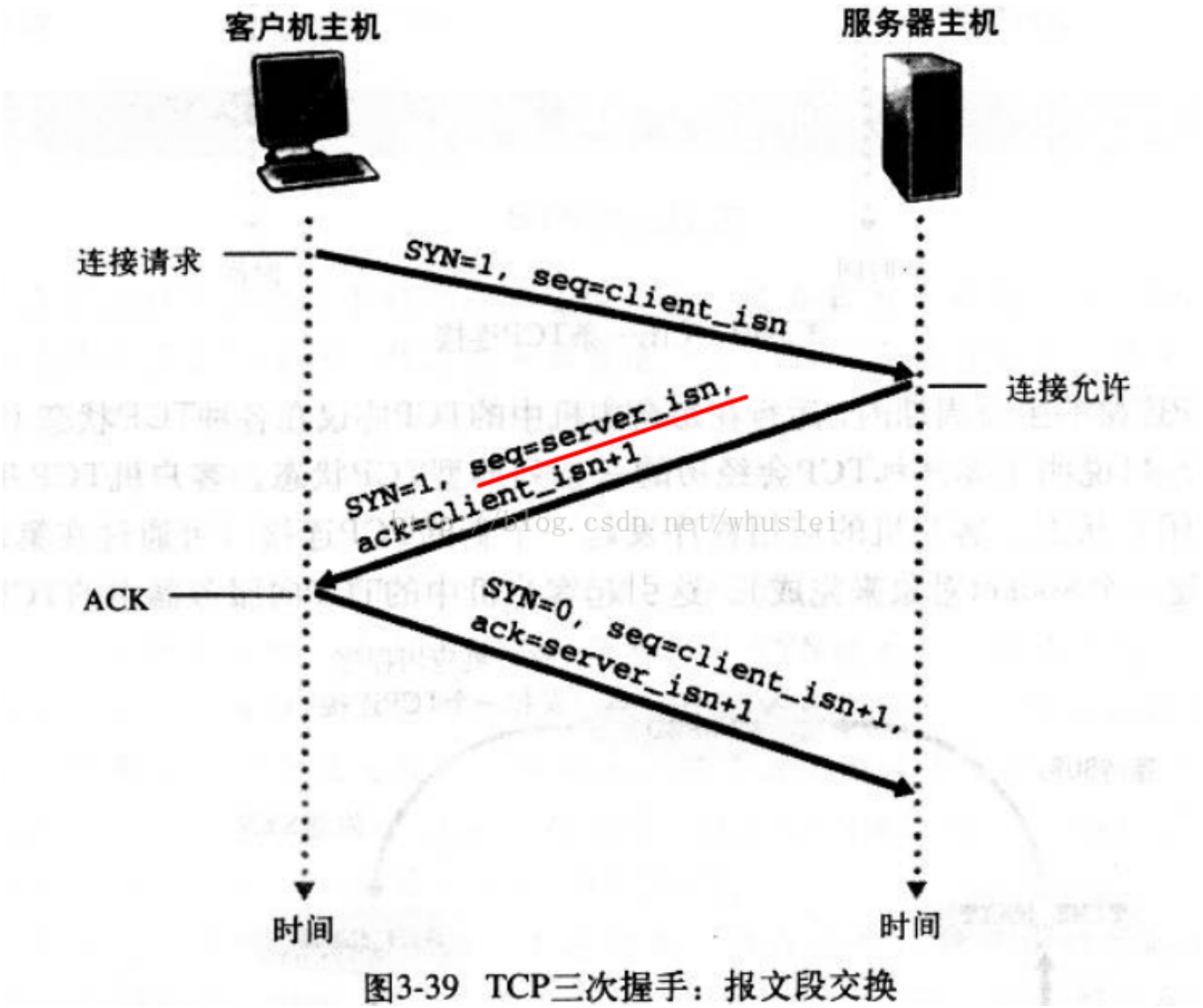


图3-39 TCP三次握手：报文段交换

四次挥手

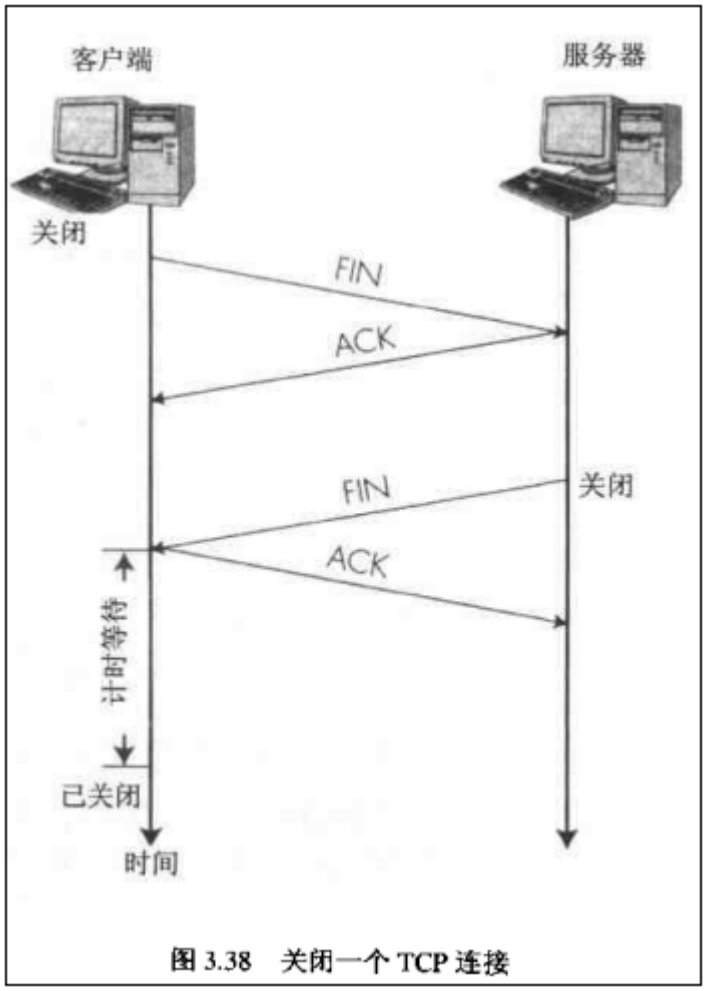


图 3.38　关闭一个 TCP 连接

UDP

UDP（User Data Protocol），用户数据报协议，UDP是一个非连接的协议，不需要维护连接状态。UDP尽最大努力交付，不保证可靠交付。UDP是面向报文的。ping命令的原理就是向对方主机发送UDP数据包。

TCP和UDP区别

	TCP	UDP
是否连接	面向连接	面向非连接
传输可靠性	可靠	不可靠
应用场合	传输大量数据	少量数据
速度	慢	快

SimpleDateFormat

参考网址：https://www.cnblogs.com/java1024/p/8594784.html

集成关系

```
public class SimpleDateFormat extends DateFormat {...}
```

造成线程不安全的原因

DateFormat类中有成员变量

```
protected Calendar calendar;
```

在SimpleDateFormat进行format()时，使用同一个calendar变量进行时间设置，在多线程访问的情况下，则有问题。

源码：

```
private StringBuffer format(Date date, StringBuffer toAppendTo,
                            FieldDelegate delegate) {
        // Convert input date to time field list
        calendar.setTime(date);//此处会造成多线程访问线程不安全问题
        boolean useDateFormatSymbols = useDateFormatSymbols();

        for (int i = 0; i < compiledPattern.length; ) {
            int tag = compiledPattern[i] >>> 8;
            int count = compiledPattern[i++] & 0xff;
            if (count == 255) {
                count = compiledPattern[i++] << 16;
                count |= compiledPattern[i++];
            }

            switch (tag) {
            case TAG_QUOTE_ASCII_CHAR:
                toAppendTo.append((char)count);
                break;

            case TAG_QUOTE_CHARS:
                toAppendTo.append(compiledPattern, i, count);
                i += count;
                break;

            default:
                subFormat(tag, count, delegate, toAppendTo, useDateFormatSymbols);
                break;
            }
        }
        return toAppendTo;
    }
```

解决方案

借用**ThreadLocal**类

代码实现：

```
public class ConcurrentDateUtil {
    private static ThreadLocal<DateFormat> threadLocal = new ThreadLocal<DateFormat>() {
            @Override
            protected DateFormat initialValue() {
                return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            }
        };

        public static Date parse(String dateStr) throws ParseException {
            return threadLocal.get().parse(dateStr);
        }

        public static String format(Date date) {
            return threadLocal.get().format(date);
        }
}
```

ThreadLocal

与Thread关系

Thread中定义了ThreadLocal.ThreadLocalMap 类型的私有变量

```
public class Thread implements Runnable {
    //Thread中定义了ThreadLocal.ThreadLocalMap 类型的变量
    ThreadLocal.ThreadLocalMap threadLocals = null;
    ...
}
```

ThreadLocalMap

ThreadLocalMap是ThreadLocal内部类

```
public class ThreadLocal<T> {
    static class ThreadLocalMap {....}
}
```

ThreadLocal方法

initialValue

```
 protected T initialValue() {
        return null;
    }
```

get

调用ThreadLocal中的get()方法，获取本线程的ThreadLocalMap threadLocals变量，如果threadLocals为空，空则调用setInitialValue()方法。如果threadLocals不为空，则获取threadLocals中以当前ThreadLocal对象为key的单链表Entry实例e，e不为空则返回e.value，e为空则调用setInitialValue()方法

```
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}
```

getMap(Thread t)

返回当前线程的ThreadLocalMap threadLocals变量

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

setInitialValue

在调用ThreadLocal中的get()方法是ThreadLocalMap map为空或map中get(key)为空则调用setInitialValue()方法

现获取初始化方法initialValue() v1中的值，若map不为空则将v1 set进map中，若map为空，则创建map并将v1 set进map中

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

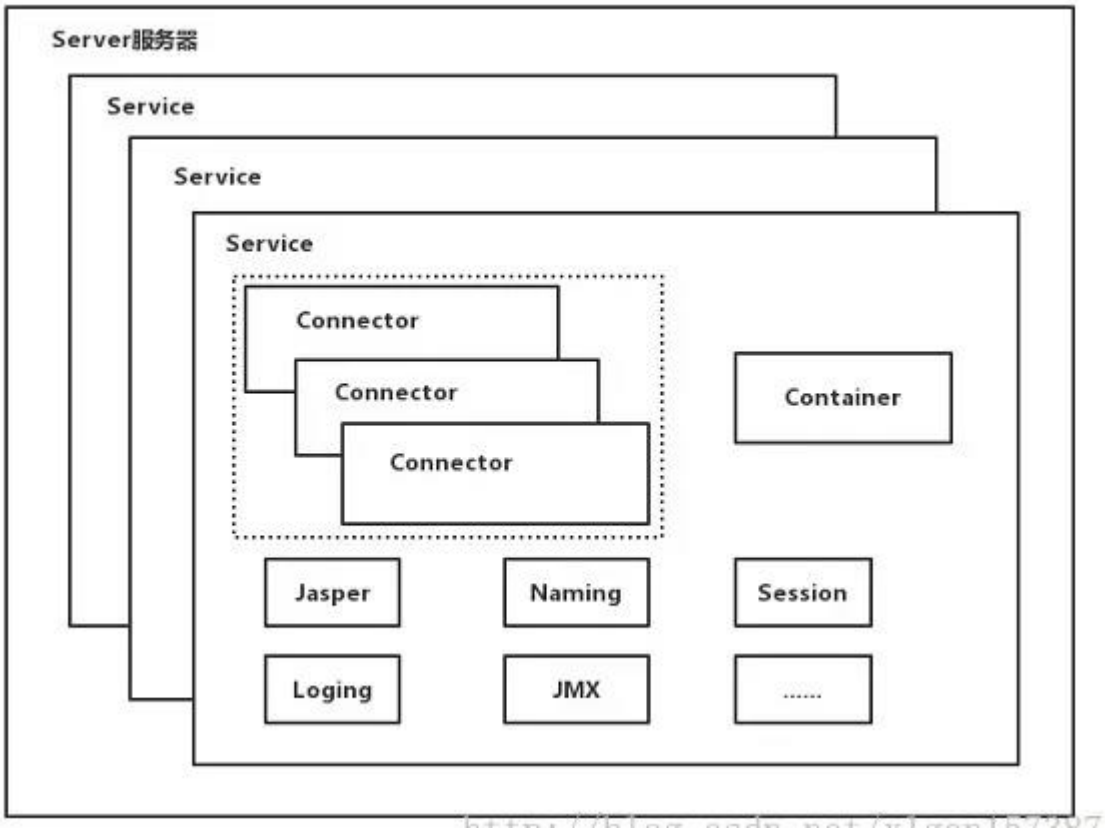
createMap

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

tomcat

Tomcat顶层架构

tomcat顶层结构图



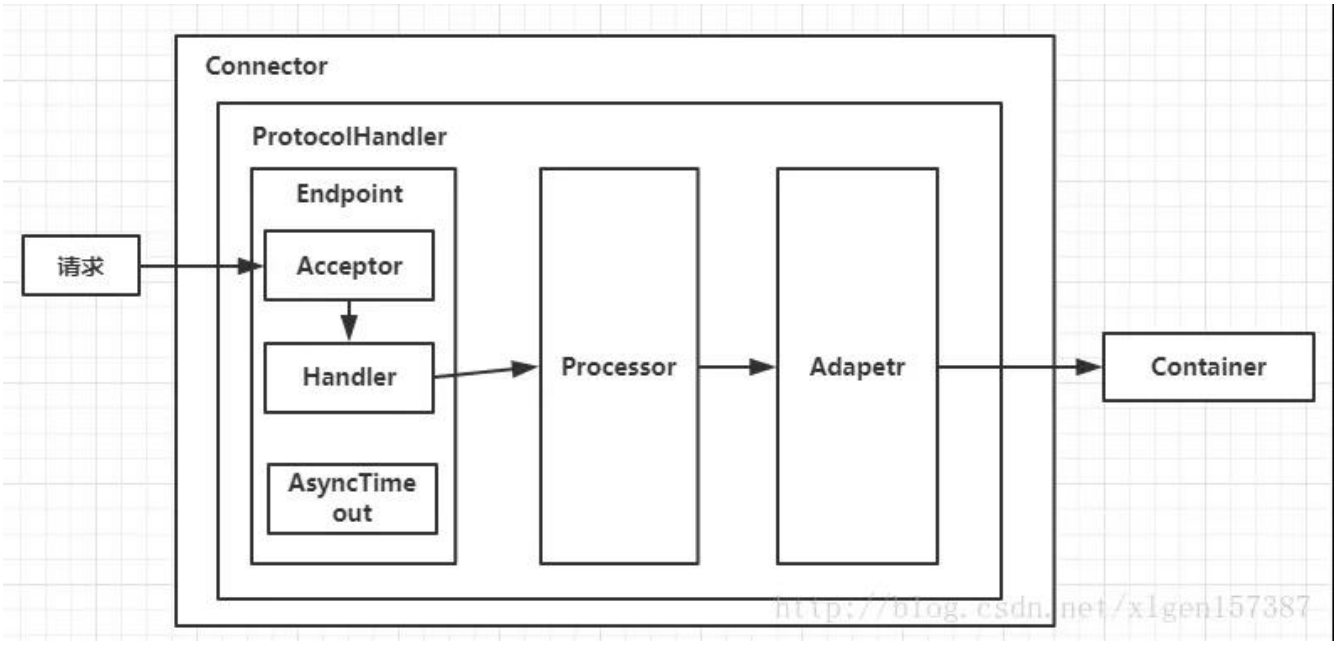
从上图可以看出一个Tomcat中只有一个Server，一个Server可以包含多个Service，一个Service只有一个Container，但是可以有多个Connectors，这是因为一个服务可以有多个连接，如同时提供Http和Https链接，也可以提供向相同协议不同端口的连接，多个 Connector 和一个 Container 就形成了一个 Service

Connector用于接受请求并将请求封装成Request和Response来具体处理；

Container用于封装和管理Servlet，以及具体处理request请求；

Connector的结构

Connector结构图



Connector -ProtocolHandler -Endpoint -Acceptor(用于监听请求) -AsyncTimeout(用于检查异步Request的超时) -Handler(Handler用于处理接收到的Socket，在内部调用Processor进行处理) -Processor -Adapter

ProtocolHandler处理请求，不同的ProtocolHandler代表不同的连接类型，比如：Http11Protocol使用的是普通Socket来连接的，Http11NioProtocol使用的是NioSocket来连接的。

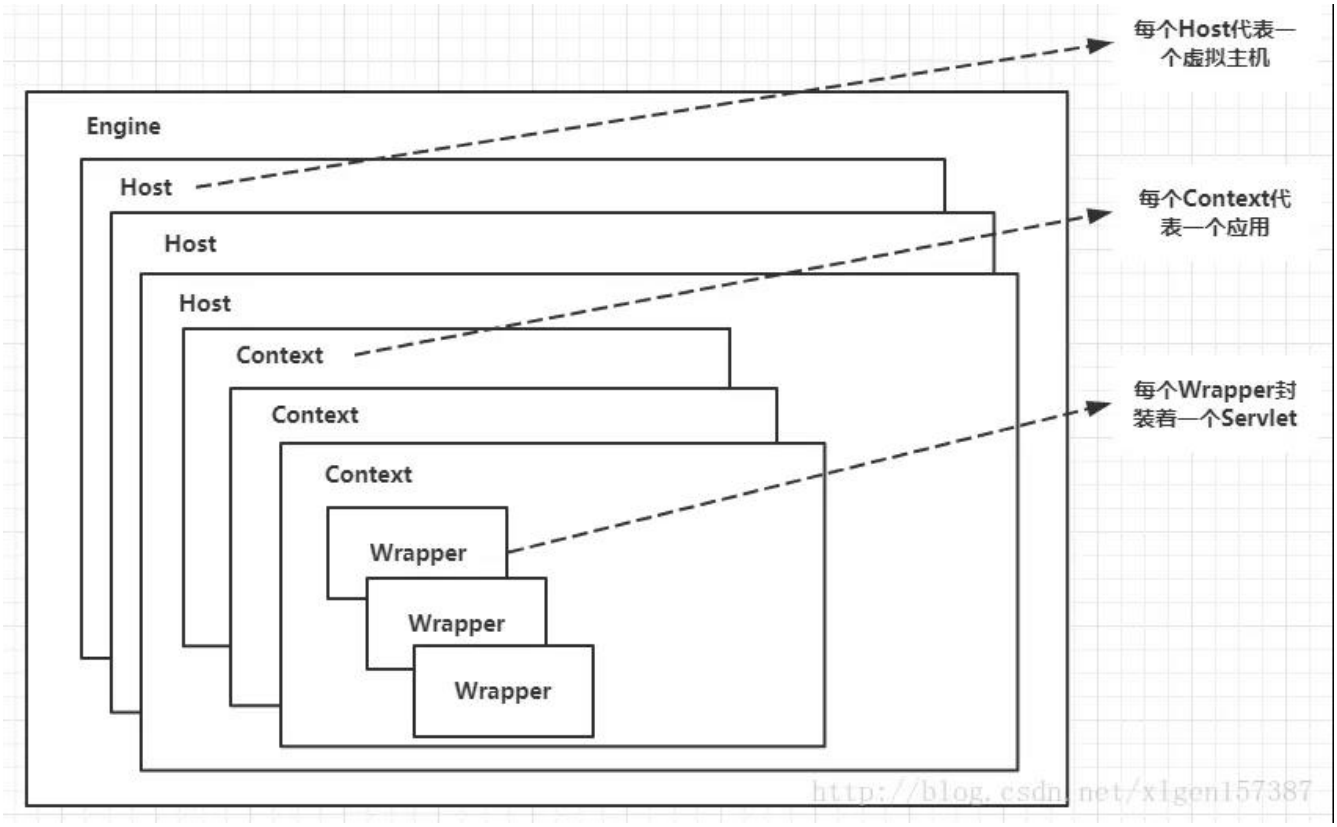
Endpoint用来处理底层Socket的网络连接,用来实现TCP/IP协议的。

Processor用于将Endpoint接收到的Socket封装成Request,用来实现HTTP协议的。

Adapter用于将Request交给Container进行具体的处理,将请求适配到Servlet容器进行具体的处理。

Container架构

Container架构图



Engine：引擎，用来管理多个站点，一个Service最多只能有一个Engine；

Host：代表一个站点，也可以叫虚拟主机，通过配置Host就可以添加站点；整个webapps就是一个Host站点。

Context：代表一个应用程序，对应着平时开发的一套程序，或者一个WEB-INF目录以及下面的web.xml文件；

Wrapper：每一Wrapper封装着一个Servlet；

Container处理请求是使用责任链模式(Pipeline-Valve)。责任链模式是指在一个请求处理的过程中有很多处理者依次对请求进行处理，每个处理者负责做自己相应的处理，处理完之后将处理后的请求返回，再让下一个处理者继续处理。

Container责任链调用流程：

EngineValve1...->StandardEngineValve->HostValve1...StandardHostValve->ContextValve1...StandardContextValve->WrapperValve1...->StandardWrapperValve

当执行到StandardWrapperValve的时候，会在StandardWrapperValve中创建FilterChain，并调用其doFilter方法来处理请求，这个FilterChain包含着我们配置的与请求相匹配的Filter和Servlet，其doFilter方法会依次调用所有的Filter的doFilter方法和Servlet的service方法，这样请求就得到了处理！

当所有的Pipeline-Valve都执行完之后，并且处理完了具体的请求，这个时候就可以将返回的结果交给Connector了，Connector在通过Socket的方式将结果返回给客户端。

Object

Java最基础和核心的类，在编译时会自动导入；

hashCode()

```
public native int hashCode();
```

hashCode根据一定的规则和对象相关的信息生成的一个散列值；

重写hashCode()方法的基本规则：

同一个对象多次调用hashCode()，返回的值应相同；

两个对象通过equals()比较相同时，hashCode()返回的值也应相同

equals(Object obj)

```
public boolean equals(Object obj)
```

比较两个对象的内存地址是否相等

toString()

```
public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
```

finalize()

```
protected void finalize() throws Throwable { }
```

垃圾回收器准备释放内存的时候，会先调用finalize()。

(1).对象不一定会被回收。

(2).垃圾回收不是析构函数。

(3).垃圾回收只与内存有关。

(4).垃圾回收和finalize()都是靠不住的，只要JVM还没有快到耗尽内存的地步，它是不会浪费时间进行垃圾回收的。

clone()

```
protected native Object clone() throws CloneNotSupportedException;
```

快速创建一个已有对象的副本

Object.clone()方法返回一个Object对象

clone方法首先会判对象是否实现了Cloneable接口，若无则抛出CloneNotSupportedException

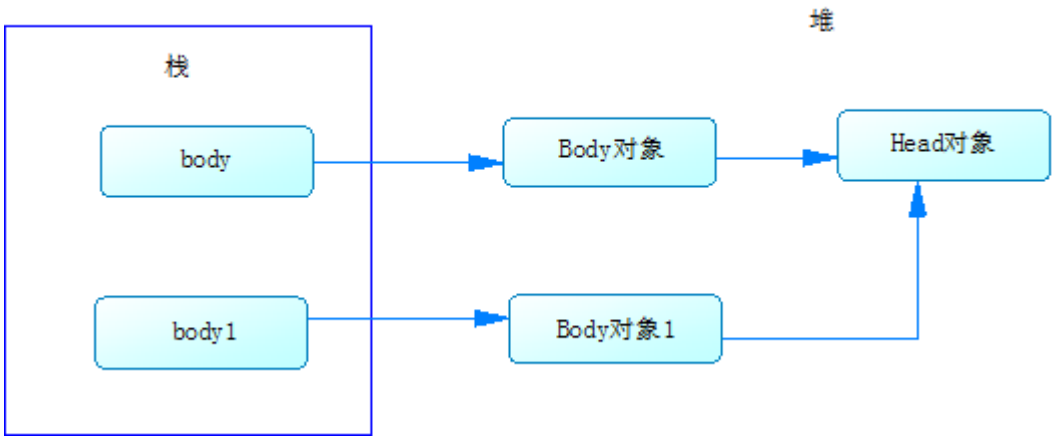
object类默认的拷贝为浅拷贝

浅拷贝

最终指向了源对象属性的地址

```
static class Body implements Cloneable{
    public Head head;
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
static class Head /*implements Cloneable*/{
    public  Face face;
    public Head() {}
    public Head(Face face){this.face = face;}
}
```

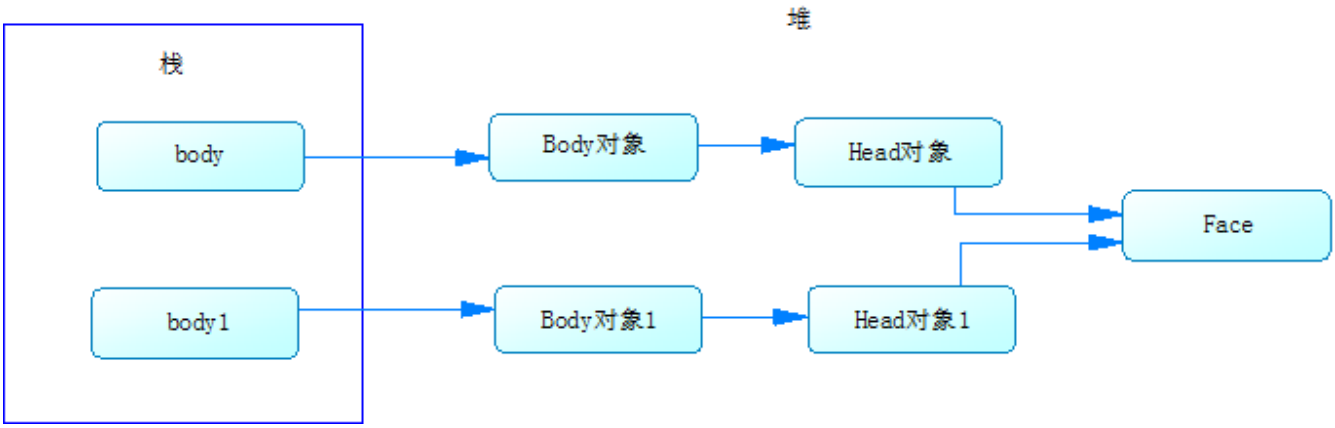
Body重写clone()，Head不重写clone()



不彻底的深拷贝

```
static class Body implements Cloneable{
    public Head head;
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
static class Head /*implements Cloneable*/{
    public  Face face;
    public Head() {}
    public Head(Face face){this.face = face;}
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
static class Face{}
```

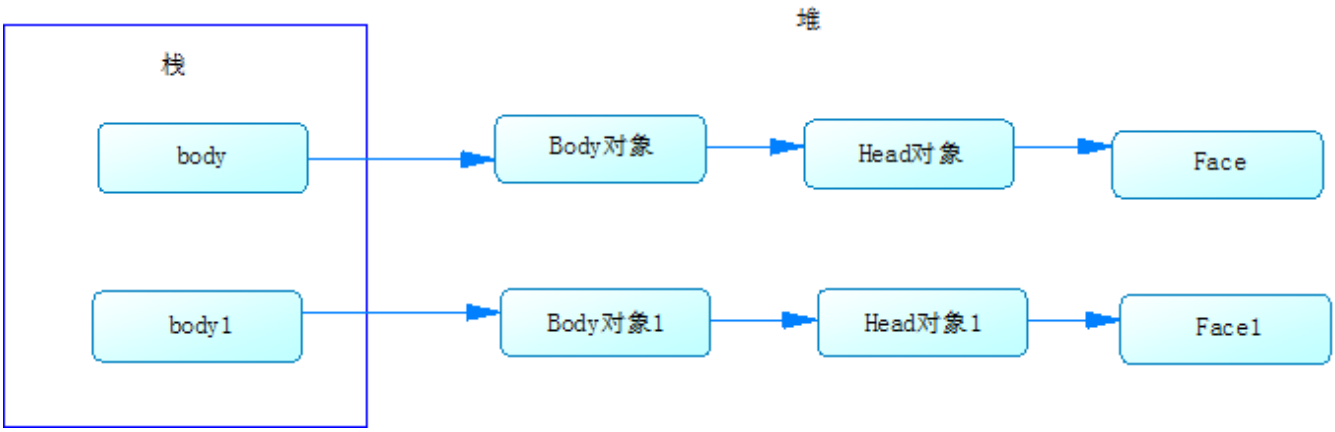
Body重写clone()，Head重写clone() Face不重写clone()



深拷贝

```
static class Body implements Cloneable{
    public Head head;
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
static class Head /*implements Cloneable*/{
    public  Face face;
    public Head() {}
    public Head(Face face){this.face = face;}
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
static class Face{
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

Body重写clone()，Head重写clone() Face重写clone()



try-catch-finally

```
private int doTry() {
    int x = 1;
    try {
        return ++x;
    } catch (Exception e) {
    } finally {
        ++x;
        System.out.println("finally - " + x);
    }
    return x;
}

@Test
public void doTryTest(){
    int i = doTry();
    System.out.println(i);
}
打印结果:
finally - 3
2
```

在try-catch-finally代码块中不管try中是否有break,return 等都会执行finally模块。也就是说，return前会执行finally语句。执行完finally语句才执行 return。

如果try语句里有return，那么代码的行为如下： 1.如果有返回值，就把返回值保存到局部变量中 2.执行到finally语句里执行 3.执行完finally语句后，返回之前保存在局部变量表里的值