八股文

1.Java String

- **String、StringBuffer、StringBuilder的区别**

  String是不可变类，任何对String的改变都会引发新的String对象产生。

  StringBuffer是可变类，其内部改变不会有新的对象产生。

  StringBuilder则是线程不安全的，在单线程下性能比StringBuffer好，区别在于StringBuffer的大多数方法都加了synchronized修饰。

- **String 的内部实现**

  JDK9之前内部通过字符数组

  ```
  private final char value[];
  ```

  JDK9开始改用

  ```
  private final byte[] value;
  ```

  改用原因：

  String类的当前实现将字符存储在char数组中，每个字符使用两个字节(16位)。从许多不同的应用程序收集的数据表明，字符串是堆使用的主要组成部分，而且，大多数字符串对象只包含拉丁字符。这些字符只需要一个字节的存储空间，因此这些字符串对象的内部char数组中有一半的空间将不会使用。

  我们建议改变字符串的内部表示classš从utf - 16字符数组到字节数组+一个encoding-flag字段。新的String类将根据字符串的内容存储编码为ISO-8859-1/Latin-1(每个字符一个字节)或UTF-16(每个字符两个字节)的字符。编码标志将指示使用哪种编码。

  在编译期，使用字符串字面量定义的String，会在常量池中存储，最大长度是65534字节。

  ```
  CONSTANT_Utf8_info {
      u1 tag;
      u2 length;
      u1 bytes[length];
  }
  ```

  可以看到length类型为u2，u2是无符号的16位整数，最大长度为65535.但受javac额外限制，当长度刚好是65535时会编译失败。因此最大只能65534个字节，eclipse使用自己的编译器，不用javac，因此最大为65535

  在运行期的String，length类型为int，那么String允许的最大长度就是Integer.MAX_VALUE，优于java中的字符是以16位存储的，因此大概需要4GB的内存才能存储最大长度的字符串。

- **字符串常量池的位置**

  Java 6及以前，字符串常量池存放在永久代

  Java 7中 oracle的工程师对字符串池的逻辑做了很大的改变，即将字符串常量池的位置调整到Java堆内

> 所有的字符串都保存在堆（Heap）中，和其他普通对象一样，这样可以让你在进行调优应用时仅需要调整堆大小就可以了。
>
> 字符串常量池概念原本使用得比较多，但是这个改动使得我们有足够的理由让我们重新考虑在Java 7中使用string.intern（）。

Java8元空间，字符串常量在堆

为什么StringTable从永久代调整到堆中

在JDK 7中，interned字符串不再在Java堆的永久生成中分配，而是在Java堆的主要部分(称为年轻代和年老代)中分配，与应用程序创建的其他对象一起分配。此更改将导致驻留在主Java堆中的数据更多，驻留在永久生成中的数据更少，因此可能需要调整堆大小。由于这一变化，大多数应用程序在堆使用方面只会看到相对较小的差异，但加载许多类或大量使用字符串的较大应用程序会出现这种差异。intern()方法会看到更显著的差异。

- 永久代的默认比较小
- 永久代垃圾回收频率低

- **字符串拼接操作**
  - 常量与常量的拼接结果在常量池，原理是编译期优化
  - 常量池中不会存在相同内容的变量
  - 只要其中有一个是变量，结果就在堆中。变量拼接的原理是StringBuilder
  - 如果拼接的结果调用intern()方法，则主动将常量池中还没有的字符串对象放入池中，并返回此对象地址

```
    public static void test1() {
        String s1 = "a" + "b" + "c";  // 得到 abc的常量池
        String s2 = "abc"; // abc存放在常量池，直接将常量池的地址返回
        /**
         * 最终java编译成.class，再执行.class
         */
        System.out.println(s1 == s2); // true，因为存放在字符串常量池
        System.out.println(s1.equals(s2)); // true
    }

    public static void test2() {
        String s1 = "javaEE";
        String s2 = "hadoop";
        String s3 = "javaEEhadoop";
        String s4 = "javaEE" + "hadoop";
        String s5 = s1 + "hadoop";
        String s6 = "javaEE" + s2;
        String s7 = s1 + s2;

        System.out.println(s3 == s4); // true
        System.out.println(s3 == s5); // false
        System.out.println(s3 == s6); // false
        System.out.println(s3 == s7); // false
        System.out.println(s5 == s6); // false
        System.out.println(s5 == s7); // false
        System.out.println(s6 == s7); // false

        String s8 = s6.intern();
        System.out.println(s3 == s8); // true
    }
```

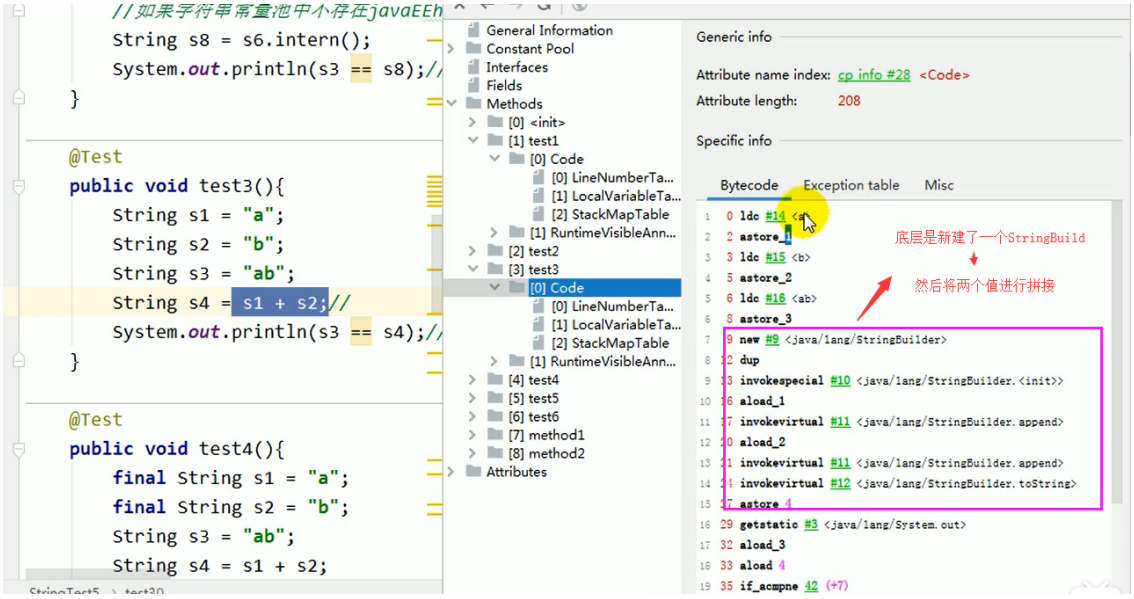
从上述的结果我们可以知道：

如果拼接符号的前后出现了变量，则相当于在堆空间中new String()，具体的内容为拼接的结果

而调用intern方法，则会判断字符串常量池中是否存在JavaEEhadoop值，如果存在则返回常量池中的值，否者就在常量池中创建

底层原理

拼接操作的底层其实使用了StringBuilder



s1 + s2的执行细节

- StringBuilder s = new StringBuilder();
- s.append(s1);
- s.append(s2);
- s.toString(); -> 类似于new String("ab");

在JDK5之后，使用的是StringBuilder，在JDK5之前使用的是StringBuffer

String	StringBuffer	StringBuilder
String的值是不可变的，这就导致每次对String的操作都会生成新的String对象，不仅效率低下，而且浪费大量优先的内存空间	StringBuffer是可变类，和线程安全的字符串操作类，任何对它指向的字符串的操作都不会产生新的对象。每个StringBuffer对象都有一定的缓冲区容量，当字符串大小没有超过容量时，不会分配新的容量，当字符串大小超过容量时，会自动增加容量	可变类，速度更快
不可变	可变	可变
	线程安全	线程不安全
	多线程操作字符串	单线程操作字符串

注意，我们左右两边如果是变量的话，就是需要new StringBuilder进行拼接，但是如果使用的是final修饰，则是从常量池中获取。所以说拼接符号左右两边都是字符串常量或常量引用 则仍然使用编译器优化。也就是说被final修饰的变量，将会变成常量，类和方法将不能被继承、

- 在开发中，能够使用final的时候，建议使用上

拼接操作和append性能对比

结论：

- 通过StringBuilder的append()方式添加字符串的效率，要远远高于String的字符串拼接方法

好处

- StringBuilder的append的方式，自始至终只创建一个StringBuilder的对象
- 对于字符串拼接的方式，还需要创建很多StringBuilder对象和 调用toString时候创建的String对象
- 内存中由于创建了较多的StringBuilder和String对象，内存占用过大，如果进行GC那么将会耗费更多的时间

改进的空间

- 我们使用的是StringBuilder的空参构造器，默认的字符串容量是16，然后将原来的字符串拷贝到新的字符串中， 我们也可以默认初始化更大的长度，减少扩容的次数
- 因此在实际开发中，我们能够确定，前前后后需要添加的字符串不高于某个限定值，那么建议使用构造器创建一个阈值的长度

- **面试题**

  new String("ab")会创建几个对象

  这里面就是两个对象

  - 一个对象是：new关键字在堆空间中创建
  - 另一个对象：字符串常量池中的对象

  new String("a") + new String("b") 会创建几个对象码文件为

  我们创建了6个对象

  - 对象1：new StringBuilder()
  - 对象2：new String("a")
  - 对象3：常量池的 a
  - 对象4：new String("b")
  - 对象5：常量池的 b
  - 对象6：toString中会创建一个 new String("ab")
    - 调用toString方法，不会在常量池中生成ab

- **intern**

  总结string的intern（）的使用：

  JDK1.6中，将这个字符串对象尝试放入串池。

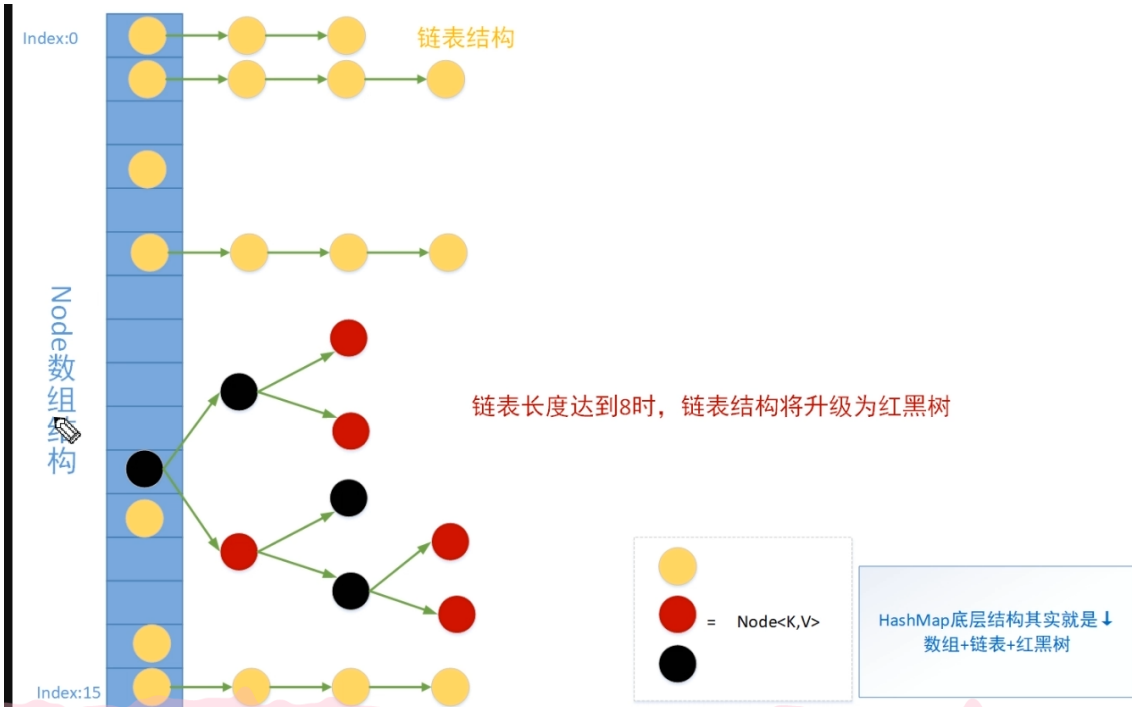
  - 如果串池中有，则并不会放入。返回已有的串池中的对象的地址
  - 如果没有，会把此**对象复制一份**，放入串池，并返回串池中的对象地址

  JDK1.7起，将这个字符串对象尝试放入串池。

  - 如果串池中有，则并不会放入。返回已有的串池中的对象的地址
  - 如果没有，则会把**对象的引用地址**复制一份，放入串池，并返回串池中的引用地址

2.java hashMap相关

- **HashMap底层原理**



- **静态常量**

```
/**
 * 默认的初始容量，必须是二的次方，后面计算index时用的是与运算而不是取模，与运算速度快，但要求长度为二的次方
 */
 static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
/**
 * 最大容量，当通过构造函数隐式指定了一个大于MAXIMUM_CAPACITY的时候使用
 */
static final int MAXIMUM_CAPACITY = 1 << 30;
/**
 * 加载因子，当构造函数没有指定加载因子的时候的默认值的时候使用
 */
static final float DEFAULT_LOAD_FACTOR = 0.75f;

/**
 * TREEIFY_THRESHOLD为当一个bin从list转化为tree的阈值，当一个bin中元素的总元素最低超过这个值的时候，bin才被转化为tree；
 * 为了满足转化为简单bin时的要求，TREEIFY_THRESHOLD必须比2大而且比8要小
 */
static final int TREEIFY_THRESHOLD = 8;
/**
 * bin反tree化时的最大值，应该比TREEIFY_THRESHOLD要小，
 * 为了在移除元素的时候能检测到移除动作，UNTREEIFY_THRESHOLD必须至少为6
 */
static final int UNTREEIFY_THRESHOLD = 6;

/**
 * 树化的另外一个阈值，table的长度(注意不是bin的长度)的最小得为64。为了避免扩容和树型结构化阈值之间的冲突，MIN_TREEIFY_CAPACITY 应该最小是 4 * TREEIFY_THRESHOLD
 */
static final int MIN_TREEIFY_CAPACITY = 64;
```

- **成员变量**

```
/**
```

```
 * table，第一次被使用的时候才进行加载
 */
transient Node<K,V>[] table;
/**
 * 键值对缓存，它们的映射关系集合保存在entrySet中。即使Key在外部修改导致hashCode变化，缓存中还可以找到映射关系
 */
transient Set<Map.Entry<K,V>> entrySet;

/**
 * table中 key-value 元素的个数
 */
transient int size;

/**
 * HashMap在结构上被修改的次数，结构上被修改是指那些改变HashMap中映射的数量或者以其他方式修改其内部结构的次数（例如，rehash）。
 * 此字段用于使HashMap集合视图上的迭代器快速失败。
 */
transient int modCount;

/**
 * 下一次resize扩容阈值，当前table中的元素超过此值时，触发扩容
 * threshold = capacity * load factor
 */
int threshold;

/**
 * 负载因子
 * @serial
 */
final float loadFactor;
```

- 构造方法

```
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                loadFactor);

    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}

/**
 *
 * 1.返回一个大于等于当前值cap的一个的数字，并且这个数字一定是2的次方数
 * 假如cap为10，那么n= 9 = 0b1001
 * 0b1001 | 0b0100 = 0b1101
 * 0b1101 | 0b0011 = 0b1111
 * 0b1111 | 0b0011 = 0b1111
 * ......
```

```
 * .....
 * n = 0b1111 = 15
 *
 * 2.这里的cap必须要减1，如果不减，并且如果传入的cap为16，那么算出来的值为32，防止二次扩容
 *
 * 3.这个方法就是为了把最高位1的后面都变为1
 * 0001 1101 1100 -> 0001 1111 1111 -> +1 -> 0010 1111 1111
 */
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

- put操作

```
 /**
  * 返回先前key对应的value值（如果value为null，也返回null），如果先前不存在这个key，那么返回的就是null；
  */
  public V put(K key, V value) {
     return putVal(hash(key), key, value, false, true);
   }
    /*
    * 在往haspmap中插入一个元素的时候，由元素的hashcode经过一个扰动函数之后再与table的长度进行与运算才找到插入位置，下面的这个hash()方法就是所谓的扰动函数
     * 作用：让key的hashCode值的高16位参与运算,hash()方法返回的值的低十六位是有hashCode的高低16位共同的特征的
     * 举例
     * hashCode = 0b 0010 0101 1010 1100  0011 1111 0010 1110
     *
     *     0b 0010 0101 1010 1100  0011 1111 0010 1110  ^
     *     0b 0000 0000 0000 0000  0010 0101 1010 1100
     *     0b 0010 0101 1010 1100  0001 1010 1000 0010
     */
    static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }


final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        // tab表示当前hashmap的table
        // p表示table的元素
        // n表示散列表的长度
        // i表示路由寻址结果
        Node<K,V>[] tab; Node<K,V> p; int n, i;

        // 延迟初始化逻辑，第一次调用putval()方法的时候才进行初始化hashmap中最耗内存的talbe
        if ((tab = table) == null || (n = tab.length) == 0)
```

```
            n = (tab = resize()).length;

        // 1.最简单的一种情况，寻找到的桶位，刚好是null，这个时候直接构建Node节点放进去
就行了
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);


        else {
            // e，如果key不为null，并且找到了当前要插入的key一致的node元素，就保存在e
中
            // k表示一个临时的key
            Node<K,V> e; K k;

            // 2.表示该桶位中的第一个元素与你当前插入的node元素的key一致，表示后序要进
行替换操作
            if (p.hash == hash &&
                    ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;

            // 3.表示当前桶位已经树化了
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
value);

            // 4.当前捅位是一个链表
            else {
                for (int binCount = 0; ; ++binCount) {
                    // 4.1 迭代到最后一个元素了也没有找到要插入的key一致的node
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }

                    // 4.1 找到了与要插入的key一致的node元素
                    if (e.hash == hash &&
                            ((k = e.key) == key || (key != null &&
key.equals(k))))
                        break;
                    p = e;
                }
            }
            // 如果找到了与要插入的key一致的node元素，那么进行替换
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        // nodeCount表示散列表table结构的修改次数，替换Node元素的value不算
        ++modCount;
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
```

```
        return null;
    }
```

- **HashMap 和 HashSet 的区别**

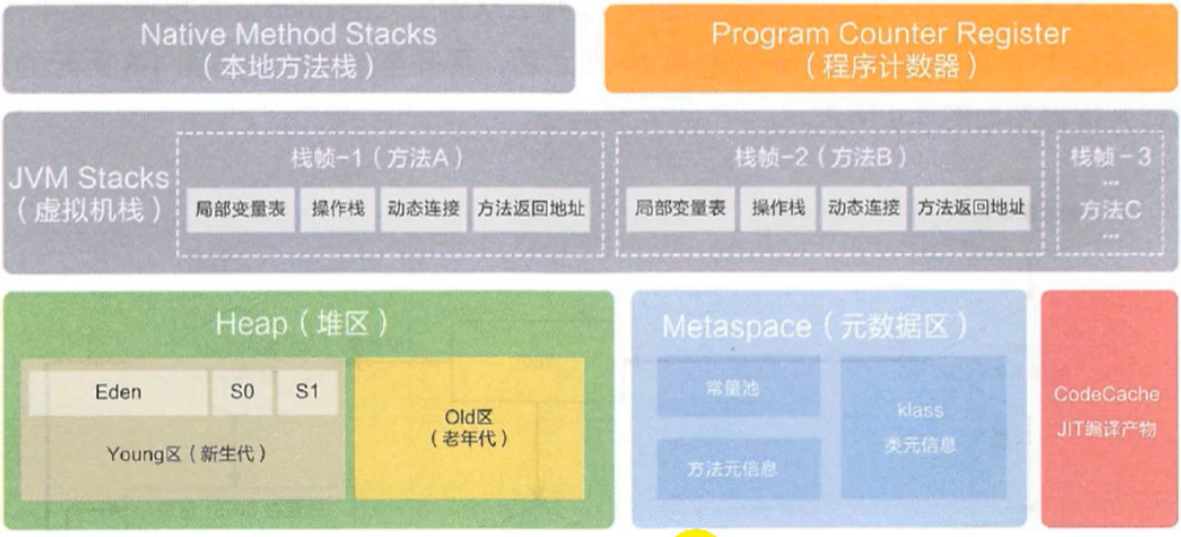
3.JVM相关

- **运行时数据区**

  Java虚拟机定义了若干种程序运行期间会使用到的运行时数据区，其中有一些会随着虚拟机启动而创建，随着虚拟机退出而销毁。另外一些则是与线程一一对应的，这些与线程对应的数据区域会随着线程开始和结束而创建和销毁。

  灰色的为单独线程私有的，红色的为多个线程共享的。即：

  - 每个线程：独立包括程序计数器、栈、本地栈。
  - 线程间共享：堆、堆外内存（永久代或元空间、代码缓存）



4.jvm 类与对象

- **创建对象的过程**
  - **加载类元信息**
  - **为对象分配内存**
  - **处理并发问题**
  - **属性的默认初始化（零值初始化）**
  - **设置对象头信息**
  - **属性的显示初始化、代码块中初始化、构造器中初始化**

    判断对象对应的类是否加载、链接、初始化

    虚拟机遇到一条new指令，首先去检查这个指令的参数能否在Metaspace的常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载，解析和初始化。（即判断类元信息是否存在）。如果没有，那么在双亲委派模式下，使用当前类加载器以ClassLoader + 包名 + 类名为key进行查找对应的 .class文件，如果没有找到文件，则抛出ClassNotFoundException异常，如果找到，则进行类加载，并生成对应的Class对象。

为对象分配内存

首先计算对象占用空间的大小，接着在堆中划分一块内存给新对象。如果实例成员变量是引用变量，仅分配引用变量空间即可，即4个字节大小

- 如果内存规整：指针碰撞
- 如果内存不规整
  - 虚拟表需要维护一个列表
  - 空闲列表分配

如果内存是规整的，那么虚拟机将采用的是指针碰撞法（Bump The Point）来为对象分配内存。

意思是所有用过的内存在一边，空闲的内存放另外一边，中间放着一个指针作为分界点的指示器，分配内存就仅仅是把指针指向空闲那边挪动一段与对象大小相等的距离罢了。如果垃圾收集器选择的是Serial，ParNew这种基于压缩算法的，虚拟机采用这种分配方式。一般使用带Compact（整理）过程的收集器时，使用指针碰撞。

如果内存不是规整的，已使用的内存和未使用的内存相互交错，那么虚拟机将采用的是空闲列表来为对象分配内存。意思是虚拟机维护了一个列表，记录上那些内存块是可用的，再分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的内容。这种分配方式成为了“空闲列表（Free List）”

选择哪种分配方式由Java堆是否规整所决定，而Java堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

处理并发问题

- 采用CAS配上失败重试保证更新的原子性
- 每个线程预先分配TLAB - 通过设置 -XX:+UseTLAB参数来设置（区域加锁机制）
  - 在Eden区给每个线程分配一块区域

初始化分配到的内存

给对象属性赋值的操作

- 属性的默认初始化
- 显示初始化
- 代码块中的初始化
- 构造器初始化
- 所有属性设置默认值，保证对象实例字段在不赋值可以直接使用

设置对象的对象头

将对象的所属类（即类的元数据信息）、对象的HashCode和对象的GC信息、锁信息等数据存储在对象的对象头中。这个过程的具体设置方式取决于JVM实现。

执行init方法进行初始化

在Java程序的视角看来，初始化才正式开始。初始化成员变量，执行实例化代码块，调用类的构造方法，并把堆内对象的首地址赋值给引用变量

因此一般来说（由字节码中跟随invokespecial指令所决定），new指令之后会接着就是执行方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完成创建出来。

5.jvm 垃圾回收

- **如何判断一个对象会被回收（标记算法）**
  - **引用计数法**

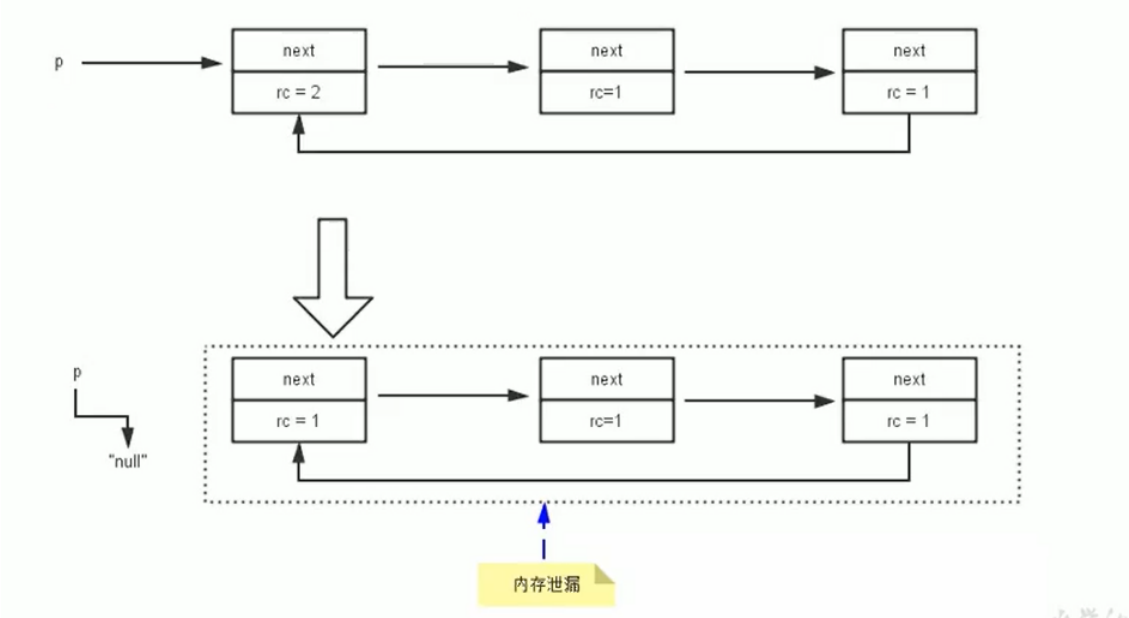
    对每个对象保存一个整型的引用计数器属性。用于记录对象被引用的情况。

    对于一个对象A，只要有任何一个对象引用了A，则A的引用计数器就加1；当引用失效时，引用计数器就减1。只要对象A的引用计数器的值为0，即表示对象A不可能再被使用，可进行回收。

    优点：实现简单，垃圾对象便于辨识；判定效率高，回收没有延迟性。

    缺点：它需要单独的字段存储计数器，这样的做法增加了存储空间的开销。

    > 每次赋值都需要更新计数器，伴随着加法和减法操作，这增加了时间开销。
    > 引用计数器有一个严重的问题，即无法处理循环引用的情况。这是一条致命缺陷，导致在Java的垃圾回收器中没有使用这类算法。python使用弱引用解决循环引用问题。



  - **可达性分析算法**

    基本思路：

    - 可达性分析算法是以根对象集合（GCRoots）为起始点，按照从上至下的方式搜索被根对象集合所连接的目标对象是否可达。
    - 使用可达性分析算法后，内存中的存活对象都会被根对象集合直接或间接连接着，搜索所走过的路径称为引用链（Reference Chain）
    - 如果目标对象没有任何引用链相连，则是不可达的，就意味着该对象已经死亡，可以标记为垃圾对象。
    - 在可达性分析算法中，只有能够被根对象集合直接或者间接连接的对象才是存活对象。

      GC Roots可以是哪些?

      - 虚拟机栈中引用的对象
        - 比如：各个线程被调用的方法中使用到的参数、局部变量等。
      - 本地方法栈内JNI（通常说的本地方法）引用的对象方法区中类静态属性引用的对象
        - 比如：Java类的引用类型静态变量
      - 方法区中常量引用的对象
        - 比如：字符串常量池（string Table）里的引用
      - 所有被同步锁synchronized持有的对象

        - Java虚拟机内部的引用。
            - 基本数据类型对应的Class对象，一些常驻的异常对象（如：Nu11PointerException、outofMemoryError），系统类加载器。
        - 反映java虚拟机内部情况的JMXBean、JVMTI中注册的回调、本地代码缓存等。

- **对象的finalize机制**

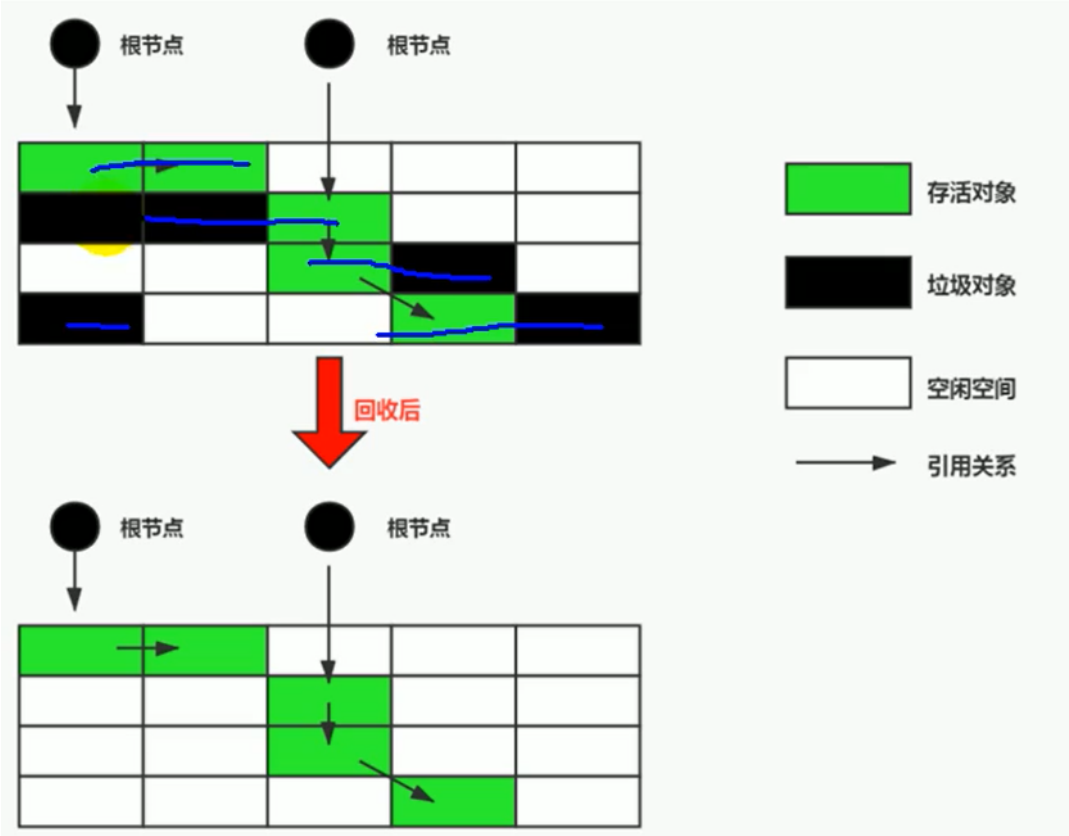
  判定一个对象objA是否可回收，至少要经历两次标记过程：

  - 如果对象objA到GC Roots没有引用链，则进行第一次标记。
  - 进行筛选，判断此对象是否有必要执行finalize（）方法
    - 如果对象objA没有重写finalize（）方法，或者finalize（）方法已经被虚拟机调用过，则虚拟机视为“没有必要执行”，objA被判定为不可触及的。
    - 如果对象objA重写了finalize（）方法，且还未执行过，那么objA会被插入到F-Queue队列中，由一个虚拟机自动创建的、低优先级的Finalizer线程触发其finalize（）方法执行。
    - finalize（）方法是对象逃脱死亡的最后机会，稍后GC会对F-Queue队列中的对象进行第二次标记。如果objA在finalize（）方法中与引用链上的任何一个对象建立了联系，那么在第二次标记时，objA会被移出“即将回收”集合。之后，对象会再次出现没有引用存在的情况。在这个情况下，finalize方法不会被再次调用，对象会直接变成不可触及的状态，也就是说，一个对象的finalize方法只会被调用一次。

  被标记的对象有机会复活！

- **垃圾回收的算法**

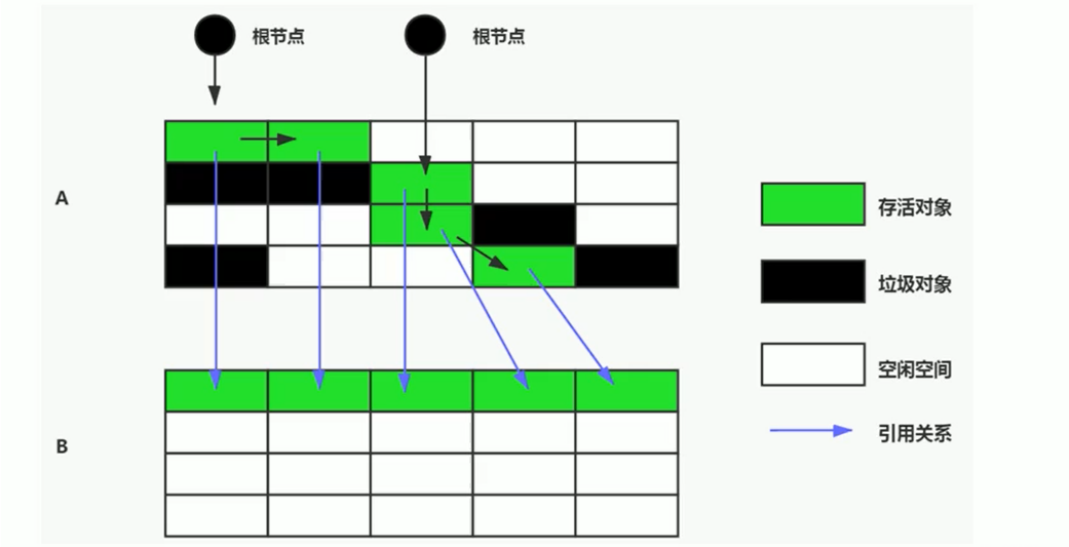
  - **标记清除**

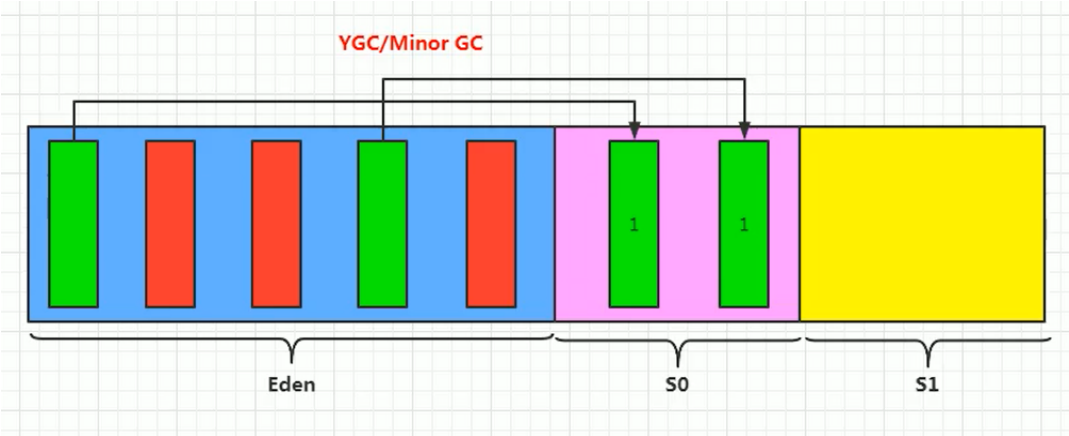
    虚拟机需要维护一个列表记录可用的地址，同时有碎片化的问题

  - **复制**

    将活着的内存空间分为两块，每次只使用其中一块，在垃圾回收时将正在使用的内存中的存活对象复制到未被使用的内存块中，之后清除正在使用的内存块中的所有对象，交换两个内存的角色，最后完成垃圾回收



把可达的对象，直接复制到另外一个区域中复制完成后，A区就没有用了，里面的对象可以直接清除掉，其实里面的新生代里面就用到了复制算法



优点

- 没有标记和清除过程，实现简单，运行高效
- 复制过去以后保证空间的连续性，不会出现"碎片"问题。

缺点

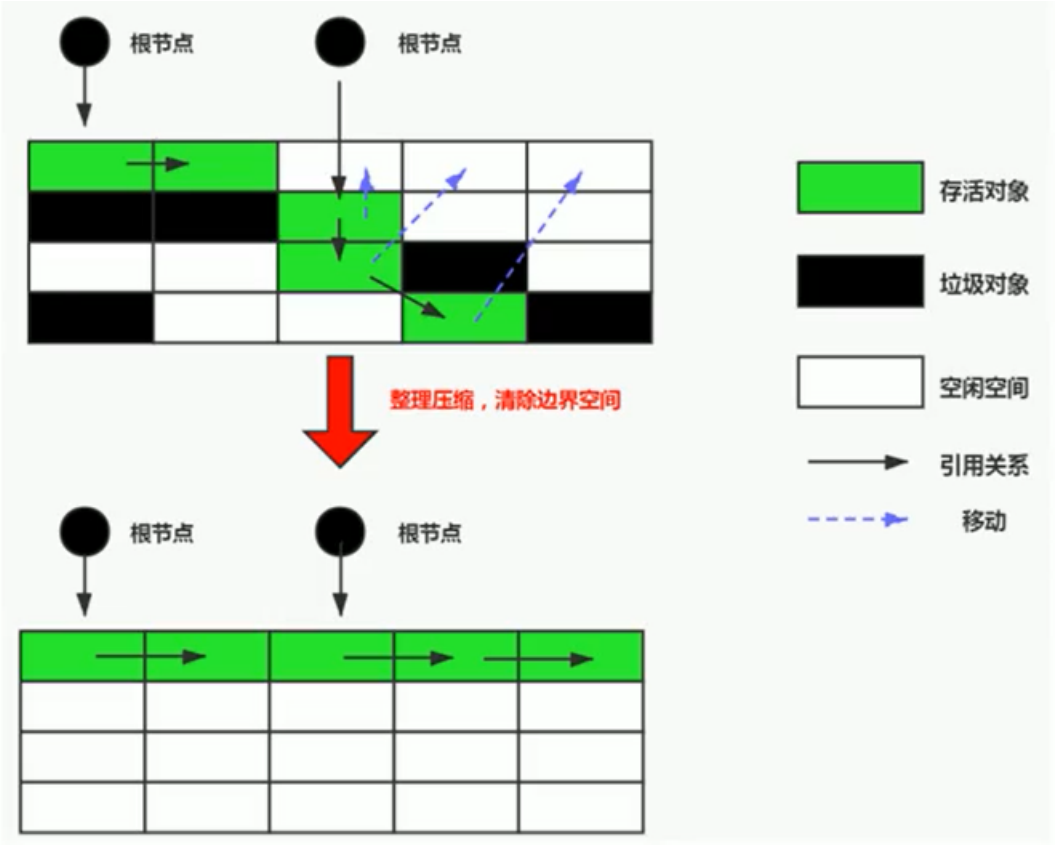
- 此算法的缺点也是很明显的，就是需要两倍的内存空间。
- 对于G1这种分拆成为大量region的GC，复制而不是移动，意味着GC需要维护region之间对象引用关系，不管是内存占用或者时间开销也不小

注意

如果系统中的垃圾对象很多，复制算法需要复制的存活对象数量并不会太大，或者说非常低才行（老年代大量的对象存活，那么复制的对象将会有很多，效率会很低）

在新生代，对常规应用的垃圾回收，一次通常可以回收70% - 99% 的内存空间。回收性价比很高。所以现在的商业虚拟机都是用这种收集算法回收新生代。

- **标记整理**



优点

- 消除了标记-清除算法当中，内存区域分散的缺点，我们需要给新对象分配内存时，JVM只需要持有一个内存的起始地址即可。
- 消除了复制算法当中，内存减半的高额代价。

缺点

- 从效率上来说，标记-整理算法要低于复制算法。
- 移动对象的同时，如果对象被其他对象引用，则还需要调整引用的地址
- 移动过程中，需要全程暂停用户应用程序。即：STW

- **分年代回收**
  - 年轻代（Young Gen）

  年轻代特点：区域相对老年代较小，对象生命周期短、存活率低，回收频繁。

  这种情况复制算法的回收整理，速度是最快的。复制算法的效率只和当前存活对象大小有关，因此很适用于年轻代的回收。而复制算法内存利用率不高的问题，通过hotspot中的两个survivor的设计得到缓解。

  - 老年代（Tenured Gen）

  老年代特点：区域较大，对象生命周期长、存活率高，回收不及年轻代频繁。

  这种情况存在大量存活率高的对象，复制算法明显变得不合适。一般是由标记-清除或者是标记-清除与标记-整理的混合实现。

- **内存管理机制与分配策略**

  如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到survivor空间中，并将对象年龄设为1。对象在survivor区中每熬过一次MinorGC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁，其实每个JVM、每个GC都有所不同）时，就会被晋升到老年代

对象晋升老年代的年龄阈值，可以通过选项-xx:MaxTenuringThreshold来设置

针对不同年龄段的对象分配原则如下所示：

- 优先分配到Eden
  - 开发中比较长的字符串或者数组，会直接存在老年代，但是因为新创建的对象 都是 朝生夕死的，所以这个大对象可能也很快被回收，但是因为老年代触发Major GC的次数比Minor GC要更少，因此可能回收起来就会比较慢
- 大对象直接分配到老年代
  - 尽量避免程序中出现过多的大对象
- 长期存活的对象分配到老年代
- 动态对象年龄判断
  - 如果survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代，无须等到MaxTenuringThreshold 中要求的年龄。

空间分配担保： -Xx:HandlePromotionFailure

- 也就是经过Minor GC后，所有的对象都存活，因为Survivor比较小，所以就需要将Survivor无法容纳的对象，存放到老年代中。

为对象分配内存：TLAB

6.操作系统 进程与线程

- **进程与线程的区别**
  - 进程

程序由指令和数据组成，但这些指令要运行，数据要读写，就必须将指令加载至 CPU，数据加载至内存。在指令运行过程中还需要用到磁盘、网络等设备。进程就是用来加载指令、管理内存、管理 IO 的。
当一个程序被运行，从磁盘加载这个程序的代码至内存，这时就开启了一个进程。
进程就可以视为程序的一个实例。大部分程序可以同时运行多个实例进程（例如记事本、画图、浏览器等），也有的程序只能启动一个实例进程（例如网易云音乐、360 安全卫士等）

- 线程

一个进程之内可以分为一到多个线程。
一个线程就是一个指令流，将指令流中的一条条指令以一定的顺序交给 CPU 执行 。
Java 中，线程作为小调度单位，进程作为资源分配的最小单位。 在 windows 中进程是不活动的，只是作 为线程的容器

- 两者对比

进程基本上相互独立的，而线程存在于进程内，是进程的一个子集 进程拥有共享的资源，如内存空间等，供其内部的线程共享
进程间通信较为复杂 同一台计算机的进程通信称为 IPC（Inter-process communication）
不同计算机之间的进程通信，需要通过网络，并遵守共同的协议，例如 HTTP
线程通信相对简单，因为它们共享进程内的内存，一个例子是多个线程可以访问同一个共享变量 线程更轻量，线程上下文切换成本一般上要比进程上下文切换低

- **线程的生命周期**
  - **初始状态**，仅仅是在语言层面上创建了线程对象，即Thead thread = new Thead();，还未与操作系统线程关联
  - **可运行状态**，也称就绪状态，指该线程已经被创建，与操作系统相关联，等待cpu给它分配时间片就可运行

- **运行状态**，指线程获取了CPU时间片，正在运行
  当CPU时间片用完，线程会转换至【可运行状态】，等待 CPU再次分配时间片，会导致我们前面讲到的上下文切换
- **阻塞状态**
  如果调用了阻塞API，如BIO读写文件，那么线程实际上不会用到CPU，不会分配CPU时间片，会导致上下文切换，进入【阻塞状态】
  等待BIO操作完毕，会由操作系统唤醒阻塞的线程，转换至【可运行状态】
  与【可运行状态】的区别是，只要操作系统一直不唤醒线程，调度器就一直不会考虑调度它们，CPU就一直不会分配时间片
- **终止状态**，表示线程已经执行完毕，生命周期已经结束，不会再转换为其它状态

7.计算机网络 HTTP

- **http缓存机制**

  Cache-Control
  Cache-Control 是最重要的规则。常见的取值有private、public、no-cache、max-age，no-store，默认为private。
  private:　　客户端可以缓存
  public:　　客户端和代理服务器都可缓存
  max-age=xxx: 缓存的内容将在 xxx 秒后失效
  no-cache:　　需要使用对比缓存来验证缓存数据
  no-store:　　所有内容都不会缓存，强制缓存，对比缓存都不会触发

  对比缓存，顾名思义，需要进行比较判断是否可以使用缓存。
  浏览器第一次请求数据时，服务器会将缓存标识与数据一起返回给客户端，客户端将二者备份至缓存数据库中。
  再次请求数据时，客户端将备份的缓存标识发送给服务器，服务器根据缓存标识进行判断，判断成功后，返回304状态码，通知客户端比较成功，可以使用缓存数据。

  Last-Modified：
  服务器在响应请求时，告诉浏览器资源的最后修改时间。

  If-Modified-Since：
  再次请求服务器时，通过此字段通知服务器上次请求时，服务器返回的资源最后修改时间。
  服务器收到请求后发现有头If-Modified-Since 则与被请求资源的最后修改时间进行比对。
  若资源的最后修改时间大于If-Modified-Since，说明资源又被改动过，则响应整片资源内容，返回状态码200；
  若资源的最后修改时间小于或等于If-Modified-Since，说明资源无新修改，则响应HTTP 304，告知浏览器继续使用所保存的cache。

  Etag：
  服务器响应请求时，告诉浏览器当前资源在服务器的唯一标识（生成规则由服务器决定）。

  If-None-Match：
  再次请求服务器时，通过此字段通知服务器客户段缓存数据的唯一标识。
  服务器收到请求后发现有头If-None-Match 则与被请求资源的唯一标识进行比对，
  不同，说明资源又被改动过，则响应整片资源内容，返回状态码200；
  相同，说明资源无新修改，则响应HTTP 304，告知浏览器继续使用所保存的cache。

- **浏览器中输入域名（url）后发生了什么**

  1. **查找域名的IP地址**

  我们在浏览器中输入一个网址（URL），首先，浏览器会根据输入的网址找到对应的IP地址。那么，怎样找到对应的IP地址呢？接下来我们就来看一下。

  （1）**URL的格式**

  一个URL包括协议，网络地址，资源路径；

协议，最常用的比如HTTP（超文本传输协议），FTP（文件传输协议）；

网络地址，可以是域名或IP地址，包括端口号，如果没有端口号，默认为80；

资源路径，可以是多种多样的。

（2）**DNS域名解析**

浏览器发现输入的网址不是IP地址，便向操作系统发送请求IP地址，操作系统启动DNS域名解析协议，接下来就开始DNS查询了。

第一步：先在各种缓存信息中查找

浏览器缓存——浏览器会缓存DNS一段时间，但是操作系统不会告诉浏览器缓存多长时间，这个缓存时间完全由浏览器自己决定。

系统缓存——如果在浏览器中没有找到，浏览器会做一个系统调用，获得系统缓存中的记录。

路由器缓存——接着会将请求发给路由器，路由器一般也有自己的DNS缓存。

如果在缓存信息中都没有查找到，则转第二步。

第二步：DNS服务器查找

全球所有的DNS服务器组成了一个DNS域名解析系统，在这个系统中，包含了全球所有的主机和IP地址的映射。所以，先在和它直接相连的DNS服务器中查找，一般情况下，在这个DNS服务器中都可以找到，但是也不排除特殊情况。

如果在和本地相连的服务器上没有找到想要的IP地址，则进行递归查找。本地服务器请求比他高一级的服务器或者根服务器，根服务器查询自己的数据库，如果知道对应的IP地址，则返回信息给本地服务器，本地服务器再将信息返回给浏览器；如果没有直接找到对应的IP地址，则告诉本地服务器应该在另外的哪一个服务器上询问，然后将询问到的信息返回给浏览器。（在根服务器上一定可以找到对应的IP地址）。

2. **TCP连接**

在知道对应的IP地址后，接下来我们就可以进行TCP连接请求了。TCP向服务器端发送SYN连接请求，经过TCP三次连接成功后，浏览器就和服务器端建立好连接了，就可以相互发送数据了。

3. **浏览器发起web服务器 HTTP请求**

根据HTTP协议的要求，组织一个HTTP数据包，HTTP请求的报头有请求行和报文，请求行包括三部分，请求方法，URL（服务器上的资源），版本。报文有一些其他信息，比如请求正文的有效载荷长度（Content_Length），缓存信息（Cache_Control）,Cookie等。

（1）**常见的请求方法**

方法	说明	支持的协议
GET	获取资源	1.0/1.1
POST	传输实体主体	1.0/1.1
PUT	传输文件	1.0/1.1
HEAD	获得报文首部	1.0/1.1
DELETE	删除文件	1．0/1.1
OPTIONS	访问支持方法	1．1
TRACE	追踪路径	1．1
CONNECT	要求用隧道协议连接代理	1．1
LINK	建立和资源之间的联系	1．1
UNLINK	断开连接关系	1．1

（2）**常见的HTTP版本**

有两种，为HTTP/1.1和HTTP/1.0。

注：HTTP报头在结束时，会向下留下空行，这个空行也是将报头和正文分开的依据。

4. **HTTP响应**

在通过HTTP请求服务后，服务器会向浏览器返回一个应答信息——HTTP响应。HTTP响应的报头包括三部分——版本，状态码，状态码描述。

常见的状态和状态码

	类型	原因短语
1XX	Informational(信息状态码)	接收的请求正在处理
2XX	Success（成功状态码）	请求正常处理完毕
3XX	Redirection（重定向状态码）	需要进行附加操作完成
4XX	Client Error(客户端错误状态码)	服务器无法处理请求
5XX	Server Error(服务器错误状态码)	服务器处理请求出错

注：如果服务器返回的响应信息为3XX，此时要转到第五步。

5. **浏览器跟踪重定向地址**

现在浏览器知道了真正要访问的目标服务器在哪里，便向此目标服务器发送和第三步相同的报文，请求响应。

6. **服务器处理请求**

服务器接收到获取请求，然后处理并返回一个响应。

这表面上看起来是一个顺向的任务，但其实这中间发生了很多有意思的东西：

Web 服务器软件
web服务器软件（像IIS和阿帕奇）接收到HTTP请求，然后确定执行什么请求处理来处理它。请求处理就是一个能够读懂请求并且能生成HTML来进行响应的程序（像ASP.NET,PHP,RUBY...）。
请求处理
请求处理阅读请求及它的参数和cookies。它会读取也可能更新一些数据，并讲数据存储在服务器上。然后，需求处理会生成一个HTML响应。

7. **浏览器解析HTML**

就像我们平常请求网页一样，浏览器会一个一个的响应出用户请求的页面，这个页面里面有表格，有图片，有文字，也可能有视频等等。

浏览器按顺序解析html文件，构建DOM树，在解析到外部的css和js文件时，向服务器发起请求下载资源，若是下载css文件，则解析器会在下载的同时继续解析后面的html来构建DOM树，则在下载js文件和执行它时，解析器会停止对html的解析。

8. **浏览器布局渲染**

布局：通过计算得到每个渲染对象在可视区域中的具体位置信息（大小和位置），这是一个递归的过程。
绘制：将计算好的每个像素点信息绘制在屏幕。

9. **TCP断开连接**

在完成所有的工作后，客户端就要发送断开连接请求了，TCP释放连接需要四次挥手。

8. jvm 强、弱等引用相关

- 软引用

  软引用是用来描述一些还有用，但非必需的对象。只被软引用关联着的对象，在系统将要发生内存溢出异常前，会把这些对象列进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。

  掉，这样就保证了使用缓存的同时，不会耗尽内存。

垃圾回收器在某个时刻决定回收软可达的对象的时候，会清理软引用，并可选地把引用存放到一个引用队列（Reference Queue）。

类似弱引用，只不过Java虚拟机会尽量让软引用的存活时间长一些，迫不得已才清理。

> 一句话概括：当内存足够时，不会回收软引用可达的对象。内存不够时，会回收软引用的可达对象

- 弱引用

> 发现即回收

弱引用也是用来描述那些非必需对象，被弱引用关联的对象只能生存到下一次垃圾收集发生为止。在系统GC时，只要发现弱引用，不管系统堆空间使用是否充足，都会回收掉只被弱引用关联的对象。

但是，由于垃圾回收器的线程通常优先级很低，因此，并不一定能很快地发现持有弱引用的对象。在这种情况下，弱引用对象可以存在较长的时间。

弱引用和软引用一样，在构造弱引用时，也可以指定一个引用队列，当弱引用对象被回收时，就会加入指定的引用队列，通过这个队列可以跟踪对象的回收情况。

软引用、弱引用都非常适合来保存那些可有可无的缓存数据。如果这么做，当系统内存不足时，这些缓存数据会被回收，不会导致内存溢出。而当内存资源充足时，这些缓存数据又可以存在相当长的时间，从而起到加速系统的作用。

- 虚引用

一个对象是否有虚引用的存在，完全不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它和没有引用几乎是一样的，随时都可能被垃圾回收器回收。

它不能单独使用，也无法通过虚引用来获取被引用的对象。当试图通过虚引用的get（）方法取得对象时，总是null

为一个对象设置虚引用关联的唯一目的在于跟踪垃圾回收过程。比如：能在这个对象被收集器回收时收到一个系统通知。

虚引用必须和引用队列一起使用。虚引用在创建时必须提供一个引用队列作为参数。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象后，将这个虚引用加入引用队列，以通知应用程序对象的回收情况。

由于虚引用可以跟踪对象的回收时间，因此，也可以将一些资源释放操作放置在虚引用中执行和记录。

> 虚引用无法获取到我们的数据

9.多线程

- **sleep和wait的区别**
  - Sleep 是 Thread 类的静态方法，Wait 是 Object 的方法，Object 又是所有类的父类，所以所有类都有Wait方法。
  - Sleep 在阻塞的时候不会释放锁，而 Wait 在阻塞的时候会释放锁，它们都会释放 CPU 资源。
  - Sleep 不需要与 synchronized 一起使用，而 Wait 需要与 synchronized 一起使用（对象被锁以后才能使用）
  - 使用 wait 一般需要搭配 notify 或者 notifyAll 来使用，不然会让线程一直等待。
- **死锁产生原因**
  - 互斥条件
    在一段时间内，一种资源只能被一个进程所使用
  - 请求和保持条件
    进程已经拥有了至少一种资源，同时又去申请其他资源。因为其他资源被别的进程所使用，该进程进入阻塞状态，并且不释放自己已有的资源

- 不可抢占条件
  进程对已获得的资源在未使用完成前不能被强占，只能在进程使用完后自己释放
- 循环等待条件
  发生死锁时，必然存在一个进程——资源的循环链。

- **线程池**
  - 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
  - 提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。
  - 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控

  java ThreadPoolExecutor

  - ThreadPoolExecutor 使用 int 的高 3 位来表示线程池状态，低 29 位表示线程数量，ThreadPoolExecutor 类中的线程状态变量如下

状态名称	高3位的值	描述
RUNNING	111	接收新任务，同时处理任务队列中的任务
SHUTDOWN	000	不接受新任务，但是处理任务队列中的任务
STOP	001	中断正在执行的任务，同时抛弃阻塞队列中的任务
TIDYING	010	任务执行完毕，活动线程为0时，即将进入终结阶段
TERMINATED	011	终结状态

```
//原子整型储存线程池状态，可以一次cas操作改变两个属性
private final AtomicInteger ctl ;
```

  - **构造参数**
    - corePoolSize：核心线程数
    - maximumPoolSize：最大线程数
      maximumPoolSize - corePoolSize = 救急线程数
    - keepAliveTime：救急线程空闲时的最大生存时间
    - unit：时间单位
    - workQueue：阻塞队列（存放任务）
    - 有界阻塞队列 ArrayBlockingQueue
    - 无界阻塞队列 LinkedBlockingQueue
    - 最多只有一个同步元素的队列 SynchronousQueue
    - 优先队列 PriorityBlockingQueue
    - threadFactory：线程工厂（给线程取名字）
    - handler：拒绝策略
  - **工作方式**
    - 线程池中刚开始没有线程，当一个任务提交给线程池后，线程池会创建一个新线程来执行任务。
    - 当线程数达到 corePoolSize 并没有线程空闲，这时再加入任务，新加的任务会被加入 workQueue 队列排 队，直到有空闲的线程。
    - 如果队列选择了有界队列，那么任务超过了队列大小时，会创建 maximumPoolSize - corePoolSize 数目的线 程来救急。
    - 如果线程到达 maximumPoolSize 仍然有新任务这时会执行拒绝策略。拒绝策略 jdk 提供了 下面的前 4 种实现，其它著名框架也提供了实现
    - ThreadPoolExecutor.AbortPolicy 让调用者抛出RejectedExecutionException 异常，这是默认策略
    - ThreadPoolExecutor.CallerRunsPolicy 让调用者运行任务
    - ThreadPoolExecutor.DiscardPolicy 放弃本次任务
    - ThreadPoolExecutor.DiscardOldestPolicy 放弃队列中最早的任务，本任务取而代之

- 当高峰过去后，超过 corePoolSize 的救急线程如果一段时间没有任务做，需要结束节省资源，这个时间由 keepAliveTime 和 unit 来控制。

跟据上述构造方法以及拒绝策略，很多框架有不同的线程池

10.计算机网络 网络安全

- **如何保障网络传输的数据安全**
  - 保密性

确保信息在存储、使用、传输过程中不会泄漏给非授权用户或实体。

- 完整性

确保信息在存储、使用、传输过程中不会被非授权用户篡改，同时还要防止授权用户对系统及信息进行不恰当的篡改，保持信息内、外部表示的一致性。

- 可用性

确保授权用户或实体对信息及资源的正常使用不会被异常拒绝，允许其可靠而及时地访问信息及资源。

加密方式

- 对称加密的原理很简单，就是数据的发送方和接收方共享一个加密数据的密钥，使用这个密钥加密的数据，可以使用这个密钥进行解密。而这个密钥是隐私的，只有数据的发送方和接收方知道，这也就意味着，其他人如果截获了数据，由于这个数据使用了密钥加密，而它没有这个密钥，所有无法解析出原始数据。
- 非对称加密系统中，参与加密解密的共有两个——**公钥**和**私钥**，使用私钥加密的数据，只能用公钥解密，而使用公钥加密的数据，只能用私钥解出。在非对称加密系统中，每一台主机都有自己的私钥和公钥，私钥只有自己知道，而公钥是公开的，可以让所有主机知道。发送方在发送数据时，使用接收方的公钥进行加密，而接收方使用自己的私钥进行解密，即可完成隐私的数据传输。如果数据被其它人截获，但是因为它没有接收方的私钥，所以无法解析出数据。

  非对称加密能够工作的一个前提是，必须确保发送方拿到的公钥，就是接收方的公钥，而不是其他人发送来的假公钥，如果公钥是假的，那么这个机制也就失去了意义。在实际应用中，解决这个问题的方式就是，每一台主机的公钥和私钥，都是由官方机构所分配的，这些机构被称为**认证中心**（`CA`）。`CA`在分配公钥私钥时，会严格地验证身份，然后对身份进行绑定，而我们在获取公钥时，通过`CA`获取，即可保证获取到的公钥就是接收方的。

  需要注意的一点是，**非对称加密的效率一般比较低，而对称加密的效率相对较高。**下面，开始正式讨论解决上面三个问题的方案。

解决数据机密性

（一）非对称加密

1. 发送方获取接受方的公钥，使用公钥对需要发送的数据进行加密，然后发送；
2. 接受方接收到后，使用自己的私钥进行解密，解析出数据；

> 总结：因为只有接受方知道自己的私钥，所以只有接受方能读出数据。但是，非对称加密的执行效率比较低，所以每一次数据传输都使用非对称加密，**响应速度将会比较慢**；

（二）非对称加密 + 对称加密（多次传输）

为了解决非对称加密效率较低的问题，我们可以使用对称加密，但是同步对称加密的密钥，却需要依赖于非对称加密：

1. 发送方随机生成一个密钥，然后获取接受方的公钥，使用公钥加密这个密钥，发送给接受方；
2. 接收方接收到加密的密钥后，使用自己的私钥解析出密钥，此时双方就完成了密钥同步；

3. 之后双方发送的所有数据，都可以使用这个密钥进行加密解密；

> 总结：由于私钥只有接收方自己知道，所以这个密钥不会被其他人截获；同时使用对称加密的速度，要高于非对称加密，所以解决了上一个方案效率不高的问题；**需要注意，一般密钥都比较短，所以使用非对称加密对密钥进行加密，一般比直接加密数据更快，而且只需要进行一次，所以速度能够显著提高。**

`HTTPS`依赖于`SSL`保证数据传输的安全性，而`SSL`就是使用类似机制。

（三）非对称加密 + 对称加密（单次传输）

如果发送方只是需要向接收方发送一次数据，那先进行一次密钥同步可能有些浪费时间，可以使用如下方案解决：

1. 发送方随机生成一个密钥，然后使用这个密钥对数据进行加密；
2. 发送方使用接收方的公钥对数据密钥进行加密，然后将加密的数据和加密的密钥发送；
3. 接收方首先使用自己的私钥解析出密钥，然后使用解析出的密钥将数据解析出来；

> 总结：此方案适合于进行单次数据发送，因为不需要进行密钥的同步，而是将密钥与数据一同发送；同时，这个密钥使用了接收方的公钥加密，所以这个密钥只有接收方自己能解析出来，而其他人解析不出密钥，自然无法解析数据；

上面的方式非常简单，就是将我们之前提过的加密，以及2.4中的方案组合，以此来同时解决三个问题。这是一个非常常用的方案，比如安全的邮件传输协议的实现就使用了类似方案。

- **MD5加密后可以解密吗**

  不能，MD5加密原理是散列算法，散列算法也称哈希算法。
  计算机专业学的数据结构就有哈希表这一知识点。
  比如10除以3余数为一，4除以3余数也为一，但余数为一的就不知道这个数是哪个了。
  所以md5不能解密。
  就算是设计这个加密算法的人都不知道。
  但是你的密码是怎么验证的呢？就是因为同一密码加密后一定相同。
  你输入密码加密后才能知道你的密码是否正确。
  也就是说，你的密码只有你自己知道。

- **如何规避DNS劫持**

  1.在不同的网络上运行分离的域名服务器来取得冗余性。

  2.将外部和内部域名服务器分开(物理上分开或运行BIND Views)并使用转发器(forwarders)。外部域名服务器应当接受来自几乎任何地址的查询，但是转发器则不接受。它们应当被配置为只接受来自内部地址的查询。关闭外部域名服务器上的递归功能(从根服务器开始向下定位DNS记录的过程)。这可以限制哪些DNS服务器与Internet联系。

  3.可能时，限制动态DNS更新。

  4.将区域传送仅限制在授权的设备上。

  5.利用事务签名对区域传送和区域更新进行数字签名。

  6.隐藏运行在服务器上的BIND版本。

  7.删除运行在DNS服务器上的不必要服务，如FTP、telnet和HTTP。

  8.在网络外围和DNS服务器上使用防火墙服务。将访问限制在那些DNS功能需要的端口/服务上。

- **规避泛洪攻击**

其实最常用的一个手段就是优化主机系统设置。比如降低SYN timeout时间，使得主机尽快释放半连接的占用或者采用SYN cookie设置，如果短时间内收到了某个IP的重复SYN请求，我们就认为受到了攻击。我们合理的采用防火墙设置等外部网络也可以进行拦截。也可以设置黑名单。