

## 2.1. Java 基础

### 2.1.1. 面向对象和面向过程的区别

- **面向过程**：面向过程性能比面向对象高。因为类调用时需要实例化，开销比较大，比较消耗资源，所以当性能是最重要的考量因素的时候，比如单片机、嵌入式开发、Linux/Unix 等一般采用面向过程开发。但是，面向过程没有面向对象易维护、易复用、易扩展。
- **面向对象**：面向对象易维护、易复用、易扩展。因为面向对象有封装、继承、多态性的特性，所以可以设计出低耦合的系统，使系统更加灵活、更加易于维护。但是，面向对象性能比面向过程低。

参见 issue：[面向过程：面向过程性能比面向对象高？](#)

这个并不是根本原因，面向过程也需要分配内存，计算内存偏移量，Java 性能差的主要原因并不是因为它是面向对象语言，而是 Java 是半编译语言，最终的执行代码并不是可以直接被 CPU 执行的二进制机械码。

而面向过程语言大多都是直接编译成机械码在电脑上执行，并且其它一些面向过程的脚本语言性能也并不一定比 Java 好。

### 2.1.2. Java 语言有哪些特点？

1. 简单易学；
2. 面向对象（封装，继承，多态）；
3. 平台无关性（Java 虚拟机实现平台无关性）；
4. 可靠性；
5. 安全性；
6. 支持多线程（C++ 语言没有内置的多线程机制，因此必须调用操作系统的多线程功能来进行多线程程序设计，而 Java 语言却提供了多线程支持）；
7. 支持网络编程并且很方便（Java 语言诞生本身就是为简化网络编程设计的，因此 Java 语言不仅支持网络编程而且很方便）；
8. 编译与解释并存；

修正（参见：[issue#544](#)）：C++11 开始（2011 年的时候），C++ 就引入了多线程库，在 windows、linux、macos 都可以使用 std::thread 和 std::async 来创建线程。参考链接：<http://www.cplusplus.com/reference/thread/thread/?kw=thread>

## 2.1.3. 关于 JVM JDK 和 JRE 最详细通俗的解答

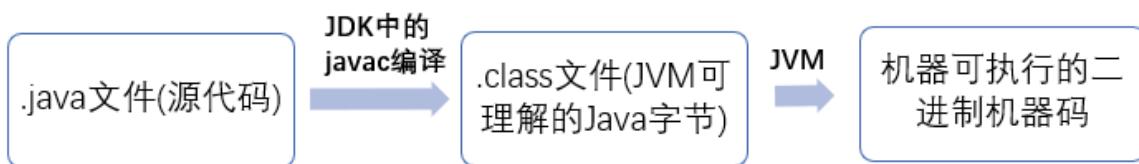
### 2.1.3.1. JVM

Java 虚拟机 (JVM) 是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现 (Windows, Linux, macOS)，目的是使用相同的字节码，它们都会给出相同的结果。

什么是字节码?采用字节码的好处是什么?

在 Java 中，JVM 可以理解的代码就叫做 字节码 (即扩展名为 .class 的文件)，它不面向任何特定的处理器，只面向虚拟机。Java 语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以 Java 程序运行时比较高效，而且，由于字节码并不针对一种特定的机器，因此，Java 程序无须重新编译便可在多种不同操作系统的计算机上运行。

Java 程序从源代码到运行一般有下面 3 步：



我们需要格外注意的是 .class->机器码 这一步。在这一步 JVM 类加载器首先加载字节码文件，然后通过解释器逐行解释执行，这种方式的执行速度会相对比较慢。而且，有些方法和代码块是经常需要被调用的(也就是所谓的热点代码)，所以后面引进了 JIT 编译器，而 JIT 属于运行时编译。当 JIT 编译器完成第一次编译后，其会将字节码对应的机器码保存下来，下次可以直接使用。而我们知道，机器码的运行效率肯定是高于 Java 解释器的。这也解释了我们为什么经常会说 Java 是编译与解释共存的语言。

HotSpot 采用了惰性评估(Lazy Evaluation)的做法，根据二八定律，消耗大部分系统资源的只有那一小部分的代码 (热点代码)，而这也正是 JIT 所需要编译的部分。JVM 会根据代码每次被执行的情况收集信息并相应地做出一些优化，因此执行的次数越多，它的速度就越快。JDK 9 引入了一种新的编译模式 AOT(Ahead of Time Compilation)，它是直接将字节码编译成机器码，这样就避免了 JIT 预热等各方面的开销。JDK 支持分层编译和 AOT 协作使用。但是，AOT 编译器的编译质量是肯定比不上 JIT 编译器的。

总结：

Java 虚拟机（JVM）是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现（Windows, Linux, macOS），目的是使用相同的字节码，它们都会给出相同的结果。字节码和不同系统的 JVM 实现是 Java 语言“一次编译，随处可以运行”的关键所在。

### 2.1.3.2. JDK 和 JRE

JDK 是 Java Development Kit，它是功能齐全的 Java SDK。它拥有 JRE 所拥有的一切，还有编译器（javac）和工具（如 javadoc 和 jdb）。它能够创建和编译程序。

JRE 是 Java 运行时环境。它是运行已编译 Java 程序所需的所有内容的集合，包括 Java 虚拟机（JVM），Java 类库，java 命令和其他的一些基础构件。但是，它不能用于创建新程序。

如果你只是为了运行一下 Java 程序的话，那么你只需要安装 JRE 就可以了。如果你需要进行一些 Java 编程方面的工作，那么你就需要安装 JDK 了。但是，这不是绝对的。有时，即使您不打算在计算机上进行任何 Java 开发，仍然需要安装 JDK。例如，如果要使用 JSP 部署 Web 应用程序，那么从技术上讲，您只是在应用程序服务器中运行 Java 程序。那你为什么需要 JDK 呢？因为应用程序服务器会将 JSP 转换为 Java servlet，并且需要使用 JDK 来编译 servlet。

### 2.1.4. Oracle JDK 和 OpenJDK 的对比

可能在看这个问题之前很多人和我一样并没有接触和使用过 OpenJDK。那么 Oracle 和 OpenJDK 之间是否存在重大差异？下面我通过收集到的一些资料，为你解答这个被很多人忽视的问题。

对于 Java 7，没什么关键的地方。OpenJDK 项目主要基于 Sun 捐赠的 HotSpot 源代码。此外，OpenJDK 被选为 Java 7 的参考实现，由 Oracle 工程师维护。关于 JVM, JDK, JRE 和 OpenJDK 之间的区别，Oracle 博客帖子在 2012 年有一个更详细的答案：

问：OpenJDK 存储库中的源代码与用于构建 Oracle JDK 的代码之间有什么区别？

答：非常接近 - 我们的 Oracle JDK 版本构建过程基于 OpenJDK 7 构建，只添加了几个部分，例如部署代码，其中包括 Oracle 的 Java 插件和 Java WebStart 的实现，以及一些封闭的源代码派对组件，如图形光栅化器，一些开源的第三方组件，如 Rhino，以及一些零碎的东西，如附加文档或第三方字体。展望未来，我们的目的是开源 Oracle JDK 的所有部分，除了我们考虑商业功能的部分。

总结：

1. Oracle JDK 大概每 6 个月发一次主要版本，而 OpenJDK 版本大概每三个月发布一次。但这不是固定的，我觉得了解这个没啥用处。详情参见：<https://blogs.oracle.com/java-platform-group/update-and-faq-on-the-java-se-release-cadence>。
2. OpenJDK 是一个参考模型并且是完全开源的，而 Oracle JDK 是 OpenJDK 的一个实现，并

不是完全开源的；

3. Oracle JDK 比 OpenJDK 更稳定。OpenJDK 和 Oracle JDK 的代码几乎相同，但 Oracle JDK 有更多的类和一些错误修复。因此，如果您想开发企业/商业软件，我建议您选择 Oracle JDK，因为它经过了彻底的测试和稳定。某些情况下，有些人提到在使用 OpenJDK 可能会遇到了许多应用程序崩溃的问题，但是，只需切换到 Oracle JDK 就可以解决问题；
4. 在响应性和 JVM 性能方面，Oracle JDK 与 OpenJDK 相比提供了更好的性能；
5. Oracle JDK 不会为即将发布的版本提供长期支持，用户每次都必须通过更新到最新版本获得支持来获取最新版本；
6. Oracle JDK 根据二进制代码许可协议获得许可，而 OpenJDK 根据 GPL v2 许可获得许可。

## 2.1.5. Java 和 C++的区别？

我知道很多人没学过 C++，但是面试官就是没事喜欢拿咱们 Java 和 C++ 比呀！没办法！！！就算没学过 C++，也要记下来！

- 都是面向对象的语言，都支持封装、继承和多态
- Java 不提供指针来直接访问内存，程序内存更加安全
- Java 的类是单继承的，C++ 支持多重继承；虽然 Java 的类不可以多继承，但是接口可以多继承。
- Java 有自动内存管理机制，不需要程序员手动释放无用内存
- 在 C 语言中，字符串或字符数组最后都会有一个额外的字符'\0'来表示结束。但是，Java 语  
言中没有结束符这一概念。这是一个值得深度思考的问题，具体原因推荐看这篇文章：  
<https://blog.csdn.net/sszgg2006/article/details/49148189>

作者：Guide 哥。

介绍：Github 90k Star 项目 **JavaGuide**（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取 Java 工程师必备学习资料+面试突击 pdf。

## 2.1.6. 字符型常量和字符串常量的区别？

1. 形式上：字符常量是单引号引起的一个字符；字符串常量是双引号引起的若干个字符
2. 含义上：字符常量相当于一个整型值( ASCII 值)，可以参加表达式运算；字符串常量代表一个地址值(该字符串在内存中存放位置)
3. 占内存大小：字符常量只占 2 个字节；字符串常量占若干个字节 (注意： char 在 Java 中占两个字节)

## java 编程思想第四版：2.2.2 节

Java要确定每种基本类型所占存储空间的大小。它们的大小并不像其他大多数语言那样随机器硬件架构的变化而变化。这种所占存储空间大小的不变性是Java程序比用其他大多数语言编写的程序更具可移植性的原因之一。

| 基本类型           | 大小      | 最小值       | 最大值                | 包装器类型            |
|----------------|---------|-----------|--------------------|------------------|
| <b>boolean</b> | —       | —         | —                  | <b>Boolean</b>   |
| <b>char</b>    | 16-bit  | Unicode 0 | Unicode $2^{16}-1$ | <b>Character</b> |
| <b>byte</b>    | 8 bits  | -128      | +127               | <b>Byte</b>      |
| <b>short</b>   | 16 bits | $-2^{15}$ | $+2^{15}-1$        | <b>Short</b>     |
| <b>int</b>     | 32 bits | $-2^{31}$ | $+2^{31}-1$        | <b>Integer</b>   |
| <b>long</b>    | 64 bits | $-2^{63}$ | $+2^{63}-1$        | <b>Long</b>      |
| <b>float</b>   | 32 bits | IEEE754   | IEEE754            | <b>Float</b>     |
| <b>double</b>  | 64 bits | IEEE754   | IEEE754            | <b>Double</b>    |
| <b>void</b>    | —       | —         | —                  | <b>Void</b>      |

## 2.1.7. 构造器 Constructor 是否可被 override?

Constructor 不能被 override (重写),但是可以 overload (重载),所以你可以看到一个类中有多个构造函数的情况。

## 2.1.8. 重载和重写的区别

重载就是同样的一个方法能够根据输入数据的不同，做出不同的处理

重写就是当子类继承自父类的相同方法，输入数据一样，但要做出有别于父类的响应时，你就要覆盖父类方法

**重载：**

发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同。

下面是《Java 核心技术》对重载这个概念的介绍：

#### 4.6.1 重载

有些类有多个构造器。例如，可以如下构造一个空的 `StringBuilder` 对象：

```
StringBuilder messages = new StringBuilder();
```

或者，可以指定一个初始字符串：

```
StringBuilder todoList = new StringBuilder("To do:\n");
```

这种特征叫做 **重载** (overloading)。如果多个方法（比如，`StringBuilder` 构造器方法）有相同的名字、不同的参数，便产生了**重载**。编译器必须挑选出具体执行哪个方法，它通过用各个方法给出的参数类型与特定方法调用所使用的值类型进行匹配来挑选出相应的方法。如果编译器找不到匹配的参数，就会产生编译时错误，因为根本不存在匹配，或者没有一个比其他的更好。（这个过程被称为**重载解析** (overloading resolution)。）

**注释：**Java 允许**重载**任何方法，而不只是构造器方法。因此，要完整地描述一个方法，需要指出方法名以及参数类型。这叫做方法的签名 (signature)。例如，`String` 类有 4 个称为 `indexOf` 的公有方法。它们的签名是

```
indexOf(int)  
indexOf(int, int)  
indexOf(String)  
indexOf(String, int)
```

返回类型不是方法签名的一部分。也就是说，不能有两个名字相同、参数类型也相同却返回不同类型值的方法。

综上：重载就是同一个类中多个同名方法根据不同的传参来执行不同的逻辑处理。

#### 重写：

重写发生在运行期，是子类对父类的允许访问的方法的实现过程进行重新编写。

1. 返回值类型、方法名、参数列表必须相同，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类。
2. 如果父类方法访问修饰符为 `private/final/static` 则子类就不能重写该方法，但是被 `static` 修饰的方法能够被再次声明。
3. 构造方法无法被重写

综上：重写就是子类对父类方法的重新改造，外部样子不能改变，内部逻辑可以改变

暖心的 Guide 哥最后再来个图表总结一下！

| 区别点   | 重载方法 | 重写方法                             |
|-------|------|----------------------------------|
| 发生范围  | 同一个类 | 子类                               |
| 参数列表  | 必须修改 | 一定不能修改                           |
| 返回类型  | 可修改  | 子类方法返回值类型应比父类方法返回值类型更小或相等        |
| 异常    | 可修改  | 子类方法声明抛出的异常类应比父类方法声明抛出的异常类更小或相等； |
| 访问修饰符 | 可修改  | 一定不能做更严格的限制（可以降低限制）              |
| 发生阶段  | 编译期  | 运行期                              |

方法的重写要遵循“两同两小一大”（以下内容摘录自《疯狂 Java 讲义》,issue#892）：

- “两同”即方法名相同、形参列表相同；
- “两小”指的是子类方法返回值类型应比父类方法返回值类型更小或相等，子类方法声明抛出的异常类应比父类方法声明抛出的异常类更小或相等；
- “一大”指的是子类方法的访问权限应比父类方法的访问权限更大或相等。

★ 关于 重写的返回值类型 这里需要额外多说明一下，上面的表述不太清晰准确：如果方法的返回类型是void和基本数据类型，则返回值重写时不可修改。但是如果方法的返回值是引用类型，重写时是可以返回该引用类型的子类的。

```

public class Hero {
    public String name() {
        return "超级英雄";
    }
}

public class SuperMan extends Hero{
    @Override
    public String name() {
        return "超人";
    }
    public Hero hero() {
        return new Hero();
    }
}

public class SuperSuperMan extends SuperMan {

```

```
public String name() {
    return "超级超级英雄";
}

@Override
public SuperMan hero() {
    return new SuperMan();
}
}
```

## 2.1.9. Java 面向对象编程三大特性: 封装 继承 多态

### 2.1.9.1. 封装

封装把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法，如果属性不想被外界访问，我们大可不必提供方法给外界访问。但是如果一个类没有提供给外界访问的方法，那么这个类也没有什么意义了。

### 2.1.9.2. 继承

继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承我们能够非常方便地复用以前的代码。

关于继承如下 3 点请记住：

1. 子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，**只是拥有**。
2. 子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
3. 子类可以用自己的方式实现父类的方法。（以后介绍）。

### 2.1.9.3. 多态

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

在 Java 中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口（实现接口并覆盖接口中同一方法）。

## 2.1.10. String StringBuffer 和 StringBuilder 的区别是什么？

### String 为什么是不可变的？

可变性

简单的来说：String 类中使用 final 关键字修饰字符数组来保存字符串，  
private final char value[]，所以 String 对象是不可变的。

补充（来自[issue 675](#)）：在 Java 9 之后，String 类的实现改用 byte 数组存储字符串  
private final byte[] value

而 StringBuilder 与 StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中也是使用字符数组保存字符串 char[] value 但是没有用 final 关键字修饰，所以这两种对象都是可变的。

StringBuilder 与 StringBuffer 的构造方法都是调用父类构造方法也就是 AbstractStringBuilder 实现的，大家可以自行查阅源码。

AbstractStringBuilder.java

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {

    /**
     * The value is used for character storage.
     */
    char[] value;

    /**
     * The count is the number of characters used.
     */
    int count;

    AbstractStringBuilder(int capacity) {
        value = new char[capacity];
    }
}
```

线程安全性

`String` 中的对象是不可变的，也就可以理解为常量，线程安全。`AbstractStringBuilder` 是 `StringBuilder` 与 `StringBuffer` 的公共父类，定义了一些字符串的基本操作，如 `expandCapacity`、`append`、`insert`、`indexOf` 等公共方法。`StringBuffer` 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。`StringBuilder` 并没有对方法进行加同步锁，所以是非线程安全的。

## 性能

每次对 `String` 类型进行改变的时候，都会生成一个新的 `String` 对象，然后将指针指向新的 `String` 对象。`StringBuffer` 每次都会对 `StringBuffer` 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用 `StringBuilder` 相比使用 `StringBuffer` 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

对于三者使用的总结：

1. 操作少量的数据：适用 `String`
2. 单线程操作字符串缓冲区下操作大量数据：适用 `StringBuilder`
3. 多线程操作字符串缓冲区下操作大量数据：适用 `StringBuffer`

## 2.1.11. 自动装箱与拆箱

- 装箱：将基本类型用它们对应的引用类型包装起来；
- 拆箱：将包装类型转换为基本数据类型；

更多内容见：[深入剖析 Java 中的装箱和拆箱](#)

## 2.1.12. 在一个静态方法内调用一个非静态成员为什么是非法的？

由于静态方法可以不通过对象进行调用，因此在静态方法里，不能调用其他非静态变量，也不可以访问非静态变量成员。

## 2.1.13. 在 Java 中定义一个不做事且没有参数的构造方法的作用

Java 程序在执行子类的构造方法之前，如果没有用 `super()` 来调用父类特定的构造方法，则会调用父类中“没有参数的构造方法”。因此，如果父类中只定义了有参数的构造方法，而在子类的构造方法中又没有用 `super()` 来调用父类中特定的构造方法，则编译时将发生错误，因为 Java 程序在父类中找不到没有参数的构造方法可供执行。解决办法是在父类里加上一个不做事且没有参数的构造方法。

## 2.1.14. 接口和抽象类的区别是什么？

1. 接口的方法默认是 `public`，所有方法在接口中不能有实现(Java 8 开始接口方法可以有默认实现)，而抽象类可以有非抽象的方法。
2. 接口中除了 `static`、`final` 变量，不能有其他变量，而抽象类中则不一定。
3. 一个类可以实现多个接口，但只能实现一个抽象类。接口自己本身可以通过 `extends` 关键字扩展多个接口。
4. 接口方法默认修饰符是 `public`，抽象方法可以有 `public`、`protected` 和 `default` 这些修饰符（抽象方法就是为了被重写所以不能使用 `private` 关键字修饰！）。
5. 从设计层面来说，抽象是对类的抽象，是一种模板设计，而接口是对行为的抽象，是一种行为的规范。

备注：

1. 在 JDK8 中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，则必须重写，不然会报错。(详见 issue:<https://github.com/Snailclimb/JavaGuide/issues/146>)。
2. jdk9 的接口被允许定义私有方法。

总结一下 jdk7~jdk9 Java 中接口概念的变化 ([相关阅读](#))：

1. 在 jdk 7 或更早版本中，接口里面只能有常量变量和抽象方法。这些接口方法必须由选择实现接口的类实现。
2. jdk 8 的时候接口可以有默认方法和静态方法功能。
3. Jdk 9 在接口中引入了私有方法和私有静态方法。

## 2.1.15. 成员变量与局部变量的区别有哪些？

1. 从语法形式上看：成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被 `public`、`private`、`static` 等修饰符所修饰，而局部变量不能被访问控制修饰符及 `static` 所修饰；但是，成员变量和局部变量都能被 `final` 所修饰。
2. 从变量在内存中的存储方式来看：如果成员变量是使用 `static` 修饰的，那么这个成员变量是属于类的，如果没有使用 `static` 修饰，这个成员变量是属于实例的。对象存于堆内存，如果局部变量类型为基本数据类型，那么存储在栈内存，如果为引用数据类型，那存放的是指向堆内存对象的引用或者是指向常量池中的地址。
3. 从变量在内存中的生存时间上看：成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动消失。
4. 成员变量如果没有被赋初值：则会自动以类型的默认值而赋值（一种情况例外：被 `final` 修饰的成员变量也必须显式地赋值），而局部变量则不会自动赋值。

## 2.1.16. 创建一个对象用什么运算符?对象实体与对象引用有何不同?

new 运算符, new 创建对象实例 (对象实例在堆内存中), 对象引用指向对象实例 (对象引用存放在栈内存中)。一个对象引用可以指向 0 个或 1 个对象 (一根绳子可以不系气球, 也可以系一个气球);一个对象可以有 n 个引用指向它 (可以用 n 条绳子系住一个气球)。

## 2.1.17. 什么是方法的返回值?返回值在类的方法里的作用是什么?

方法的返回值是指我们获取到的某个方法体中的代码执行后产生的结果! (前提是该方法可能产生结果)。返回值的作用:接收到结果, 使得它可以用于其他的操作!

## 2.1.18. 一个类的构造方法的作用是什么?若一个类没有声明构造方法, 该程序能正确执行吗?为什么?

主要作用是完成对类对象的初始化工作。可以执行。因为一个类即使没有声明构造方法也会有默认的不带参数的构造方法。

## 2.1.19. 构造方法有哪些特性?

1. 名字与类名相同。
2. 没有返回值, 但不能用 void 声明构造函数。
3. 生成类的对象时自动执行, 无需调用。

## 2.1.20. 静态方法和实例方法有何不同

1. 在外部调用静态方法时, 可以使用"类名.方法名"的方式, 也可以使用"对象名.方法名"的方式。而实例方法只有后面这种方式。也就是说, 调用静态方法可以无需创建对象。
2. 静态方法在访问本类的成员时, 只允许访问静态成员 (即静态成员变量和静态方法), 而不允许访问实例成员变量和实例方法; 实例方法则无此限制。

## 2.1.21. 对象的相等与指向他们的引用相等,两者有什么不同?

对象的相等, 比的是内存中存放的内容是否相等。而引用相等, 比较的是他们指向的内存地址是否相等。

## 2.1.22. 在调用子类构造方法之前会先调用父类没有参数的构造方法, 其目的是?

帮助子类做初始化工作。

## 2.1.23. == 与 equals(重要)

**==**：它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象(基本数据类型==比较的是值，引用数据类型==比较的是内存地址)。

**equals()**：它的作用也是判断两个对象是否相等。但它一般有两种使用情况：

- 情况 1：类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时，等价于通过“**==**”比较这两个对象。
- 情况 2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来比较两个对象的内容是否相等；若它们的内容相等，则返回 true (即，认为这两个对象相等)。

举个例子：

```
public class test1 {  
    public static void main(String[] args) {  
        String a = new String("ab"); // a 为一个引用  
        String b = new String("ab"); // b 为另一个引用，对象的内容一样  
        String aa = "ab"; // 放在常量池中  
        String bb = "ab"; // 从常量池中查找  
        if (aa == bb) // true  
            System.out.println("aa==bb");  
        if (a == b) // false, 非同一对象  
            System.out.println("a==b");  
        if (a.equals(b)) // true  
            System.out.println("aEQb");  
        if (42 == 42.0) { // true  
            System.out.println("true");  
        }  
    }  
}
```

说明：

- String 中的 equals 方法是被重写过的，因为 object 的 equals 方法是比较的对象的内存地址，而 String 的 equals 方法比较的是对象的值。
- 当创建 String 类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个 String 对象。

## 2.1.24. hashCode 与 equals (重要)

面试官可能会问你：“你重写过 `hashcode` 和 `equals` 么，为什么重写 `equals` 时必须重写 `hashCode` 方法？”

### 1) hashCode()介绍:

`hashCode()` 的作用是获取哈希码，也称为散列码；它实际上是返回一个 `int` 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。`hashCode()` 定义在 JDK 的 `Object` 类中，这就意味着 Java 中的任何类都包含有 `hashCode()` 函数。另外需要注意的是：`Object` 的 `hashcode` 方法是本地方法，也就是用 c 语言或 c++ 实现的，该方法通常用来将对象的内存地址转换为整数之后返回。

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

散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）

### 2)为什么要有 hashCode?

我们以“`HashSet` 如何检查重复”为例子来说明为什么要有 `hashCode`？

当你把对象加入 `HashSet` 时，`HashSet` 会先计算对象的 `hashcode` 值来判断对象加入的位置，同时也会与其他已经加入的对象的 `hashcode` 值作比较，如果没有相符的 `hashcode`，`HashSet` 会假设对象没有重复出现。但是如果发现有相同 `hashcode` 值的对象，这时会调用 `equals()` 方法来检查 `hashcode` 相等的对象是否真的相同。如果两者相同，`HashSet` 就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。（摘自我的 Java 启蒙书《Head First Java》第二版）。这样我们就大大减少了 `equals` 的次数，相应就大大提高了执行速度。

### 3)为什么重写 `equals` 时必须重写 `hashCode` 方法?

如果两个对象相等，则 `hashcode` 一定也是相同的。两个对象相等，对两个对象分别调用 `equals` 方法都返回 `true`。但是，两个对象有相同的 `hashcode` 值，它们也不一定是相等的。因此，`equals` 方法被覆盖过，则 `hashCode` 方法也必须被覆盖。

`hashCode()` 的默认行为是对堆上的对象产生独特值。如果没有重写 `hashCode()`，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

### 4)为什么两个对象有相同的 `hashcode` 值，它们也不一定是相等的?

在这里解释一位小伙伴的问题。以下内容摘自《Head First Java》。

因为 `hashCode()` 所使用的杂凑算法也许刚好会让多个对象传回相同的杂凑值。越糟糕的杂凑算法越容易碰撞，但这也与数据值域分布的特性有关（所谓碰撞也就是指的是不同的对象得到相同的 `hashCode`）。

我们刚刚也提到了 `HashSet`，如果 `HashSet` 在对比的时候，同样的 `hashcode` 有多个对象，它会使用 `equals()` 来判断是否真的相同。也就是说 `hashcode` 只是用来缩小查找成本。

更多关于 `hashcode()` 和 `equals()` 的内容可以查看：[Java hashCode\(\) 和 equals\(\) 的若干问题解答](#)

## 2.1.25. 为什么 Java 中只有值传递？

首先回顾一下在程序设计语言中有关将参数传递给方法（或函数）的一些专业术语。按值调用（**call by value**）表示方法接收的是调用者提供的值，而按引用调用（**call by reference**）表示方法接收的是调用者提供的变量地址。一个方法可以修改传递引用所对应的变量值，而不能修改传递值调用所对应的变量值。它用来描述各种程序设计语言（不只是 Java）中方法参数传递方式。

Java 程序设计语言总是采用按值调用。也就是说，方法得到的是所有参数值的一个拷贝，也就是说，方法不能修改传递给它的任何参数变量的内容。

下面通过 3 个例子来给大家说明

### example 1

```
public static void main(String[] args) {
    int num1 = 10;
    int num2 = 20;

    swap(num1, num2);

    System.out.println("num1 = " + num1);
    System.out.println("num2 = " + num2);
}

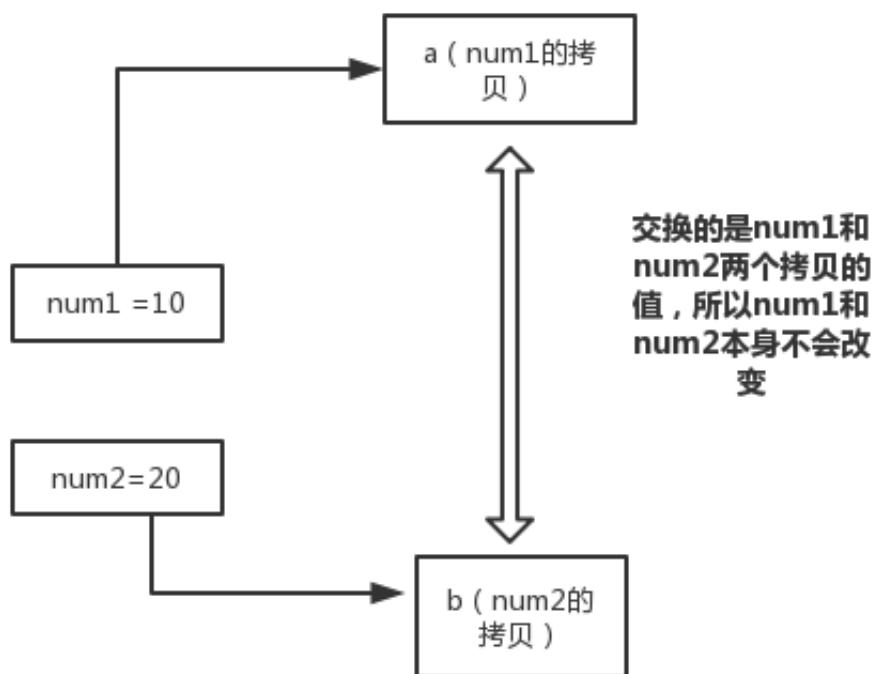
public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
```

```
    System.out.println("a = " + a);
    System.out.println("b = " + b);
}
```

结果：

```
a = 20
b = 10
num1 = 10
num2 = 20
```

解析：



在 swap 方法中，a、b 的值进行交换，并不会影响到 num1、num2。因为，a、b 中的值，只是从 num1、num2 的复制过来的。也就是说，a、b 相当于 num1、num2 的副本，副本的内容无论怎么修改，都不会影响到原件本身。

通过上面例子，我们已经知道了一个方法不能修改一个基本数据类型的参数，而对象引用作为参数就不一样，请看 example2.

## example 2

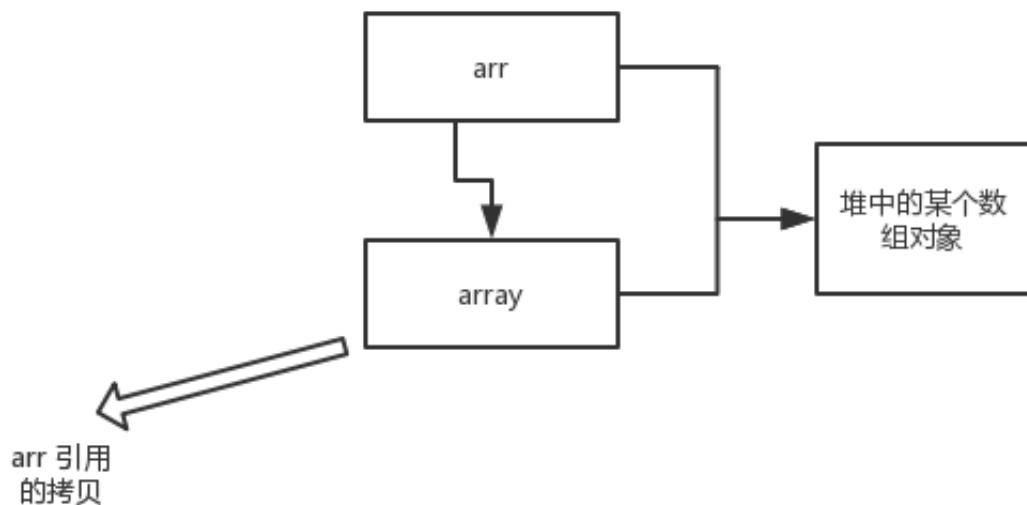
```
public static void main(String[] args) {  
    int[] arr = { 1, 2, 3, 4, 5 };  
    System.out.println(arr[0]);  
    change(arr);  
    System.out.println(arr[0]);  
}  
  
public static void change(int[] array) {  
    // 将数组的第一个元素变为0  
    array[0] = 0;  
}
```

结果：

```
1
```

```
0
```

解析：



array 被初始化 arr 的拷贝也是一个对象的引用，也就是说 array 和 arr 指向的是同一个数组对象。因此，外部对引用对象的改变会反映到所对应的对象上。

通过 example2 我们已经看到，实现一个改变对象参数状态的方法并不是一件难事。理由很简单，方法得到的是对象引用的拷贝，对象引用及其他的拷贝同时引用同一个对象。

很多程序设计语言（特别是，C++和Pascal）提供了两种参数传递的方式：值调用和引用调用。有些程序员（甚至本书的作者）认为 Java 程序设计语言对对象采用的是引用调用，实际上，这种理解是不对的。由于这种误解具有一定的普遍性，所以下面给出一个反例来详细地阐述一下这个问题。

### example 3

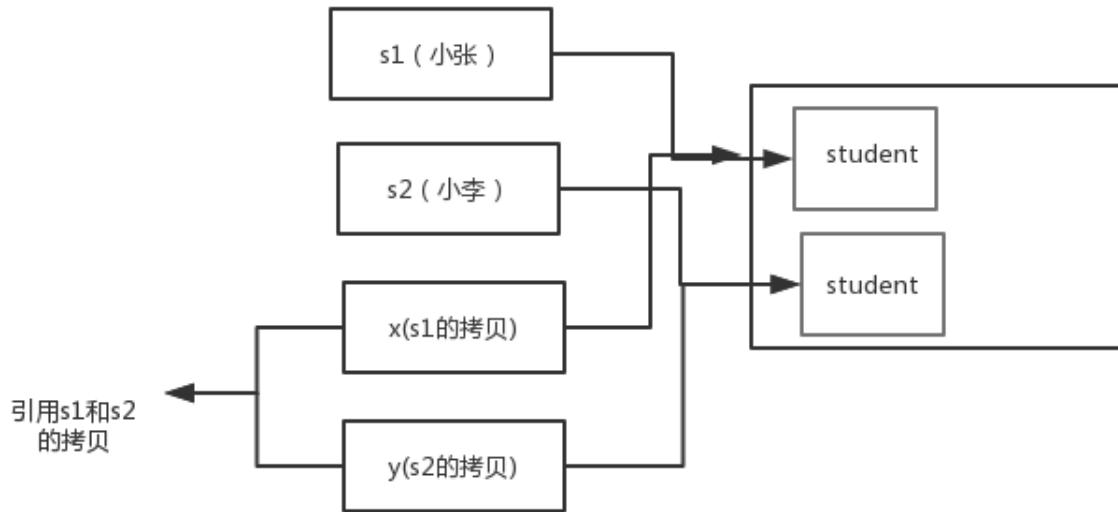
```
public class Test {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Student s1 = new Student("小张");  
        Student s2 = new Student("小李");  
        Test.swap(s1, s2);  
        System.out.println("s1:" + s1.getName());  
        System.out.println("s2:" + s2.getName());  
    }  
  
    public static void swap(Student x, Student y) {  
        Student temp = x;  
        x = y;  
        y = temp;  
        System.out.println("x:" + x.getName());  
        System.out.println("y:" + y.getName());  
    }  
}
```

结果：

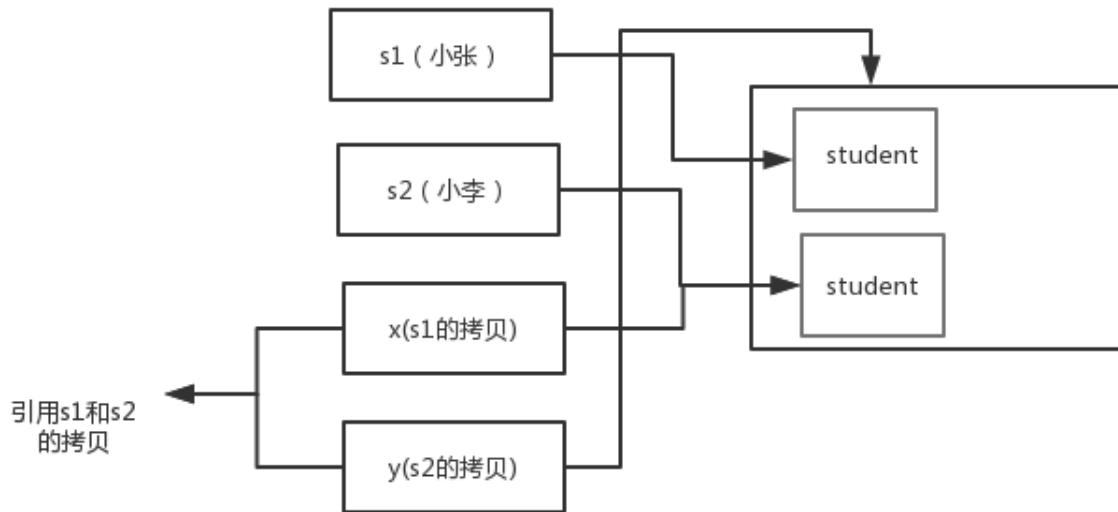
```
x:小李  
y:小张  
s1:小张  
s2:小李
```

解析：

交换之前：



交换之后：



通过上面两张图可以很清晰的看出：方法并没有改变存储在变量 `s1` 和 `s2` 中的对象引用。`swap` 方法的参数 `x` 和 `y` 被初始化为两个对象引用的拷贝，这个方法交换的是这两个拷贝

总结

Java 程序设计语言对对象采用的不是引用调用，实际上，对象引用是按值传递的。

下面再总结一下 Java 中方法参数的使用情况：

- 一个方法不能修改一个基本数据类型的参数（即数值型或布尔型）。
- 一个方法可以改变一个对象参数的状态。
- 一个方法不能让对象参数引用一个新的对象。

参考：

《Java 核心技术卷 I》基础知识第十版第四章 4.5 小节

## 2.1.26. 简述线程、程序、进程的基本概念。以及他们之间关系是什么？

线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

程序是含有指令和数据的文件，被存储在磁盘或其他的数据存储设备中，也就是说程序是静态的代码。

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。简单来说，一个进程就是一个执行中的程序，它在计算机中一个指令接着一个指令地执行着，同时，每个进程还占有某些系统资源如 CPU 时间，内存空间，文件，输入输出设备的使用权等等。换句话说，当程序在执行时，将会被操作系统载入内存中。

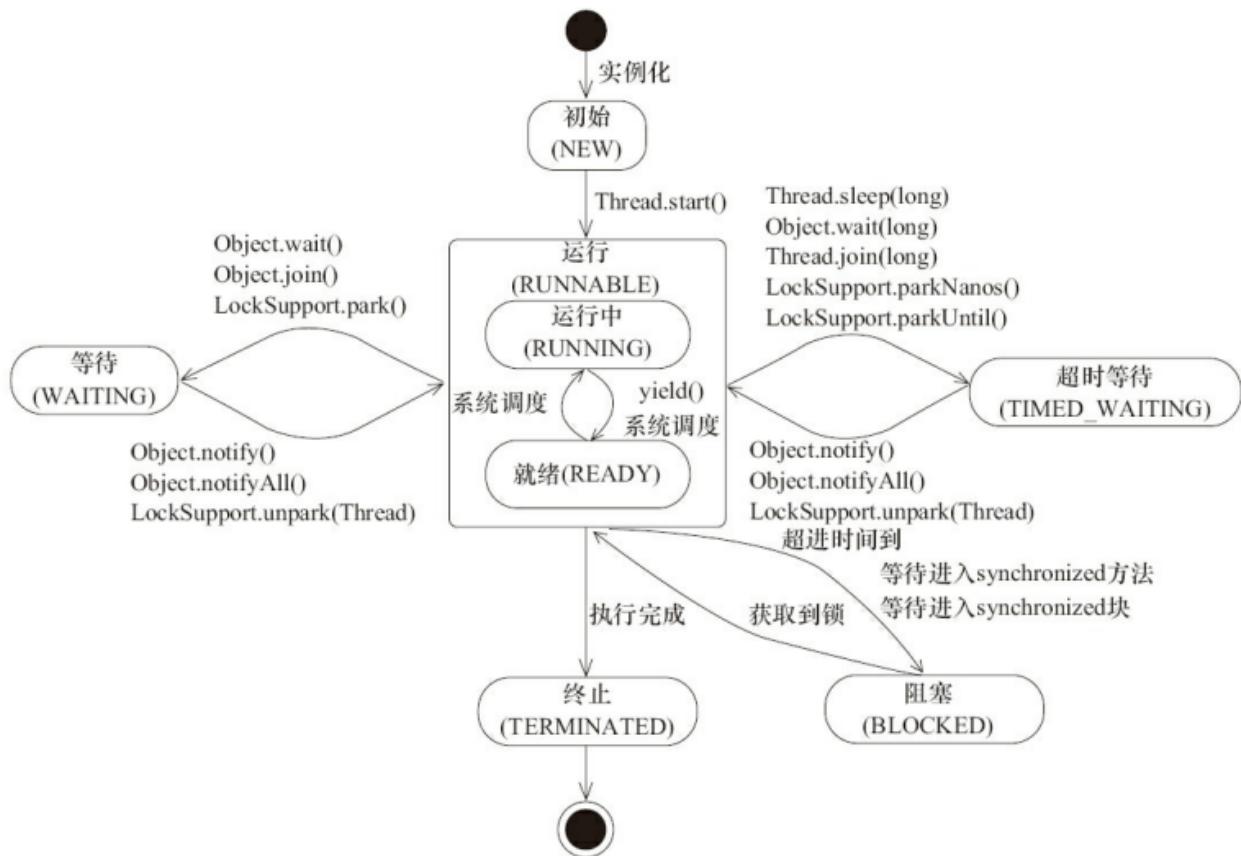
线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。从另一角度来说，进程属于操作系统的范畴，主要是同一段时间内，可以同时执行一个以上的程序，而线程则是在同一程序内几乎同时执行一个以上的程序段。

## 2.1.27. 线程有哪些基本状态？

Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态（图源《Java 并发编程艺术》4.1.4 节）。

| 状态名称         | 说    明   |
|--------------|--|
| NEW          | 初始状态，线程被构建，但是还没有调用 start() 方法                      |
| RUNNABLE     | 运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”             |
| BLOCKED      | 阻塞状态，表示线程阻塞于锁                                      |
| WAITING      | 等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断） |
| TIME_WAITING | 超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的              |
| TERMINATED   | 终止状态，表示当前线程已经执行完毕                                  |

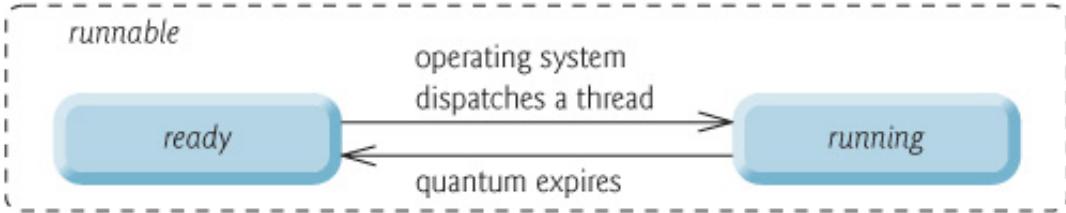
线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。Java 线程状态变迁如下图所示（图源《Java 并发编程艺术》4.1.4 节）：



由上图可以看出：

线程创建之后它将处于 **NEW**（新建）状态，调用 `start()` 方法后开始运行，线程这时候处于 **READY**（可运行）状态。可运行状态的线程获得了 cpu 时间片（timeslice）后就处于 **RUNNING**（运行）状态。

操作系统隐藏 Java 虚拟机（JVM）中的 READY 和 RUNNING 状态，它只能看到 RUNNABLE 状态（图源：[HowToDoInJava: Java Thread Life Cycle and Thread States](#)），所以 Java 系统一般将这两个状态统称为 **RUNNABLE**（运行中）状态。



当线程执行 `wait()` 方法之后，线程进入 **WAITING**（等待）状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 **TIME\_WAITING**（超时等待）状态相当于在等待状态的基础上增加了超时限制，比如通过 `sleep (long millis)` 方法或 `wait (long millis)` 方法可以将 Java 线程置于 **TIMED\_WAITING** 状态。当超时时间到达后 Java 线程将会返回到 **RUNNABLE** 状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到 **BLOCKED**（阻塞）状态。线程在执行 `Runnable` 的 `run()` 方法之后将会进入到 **TERMINATED**（终止）状态。

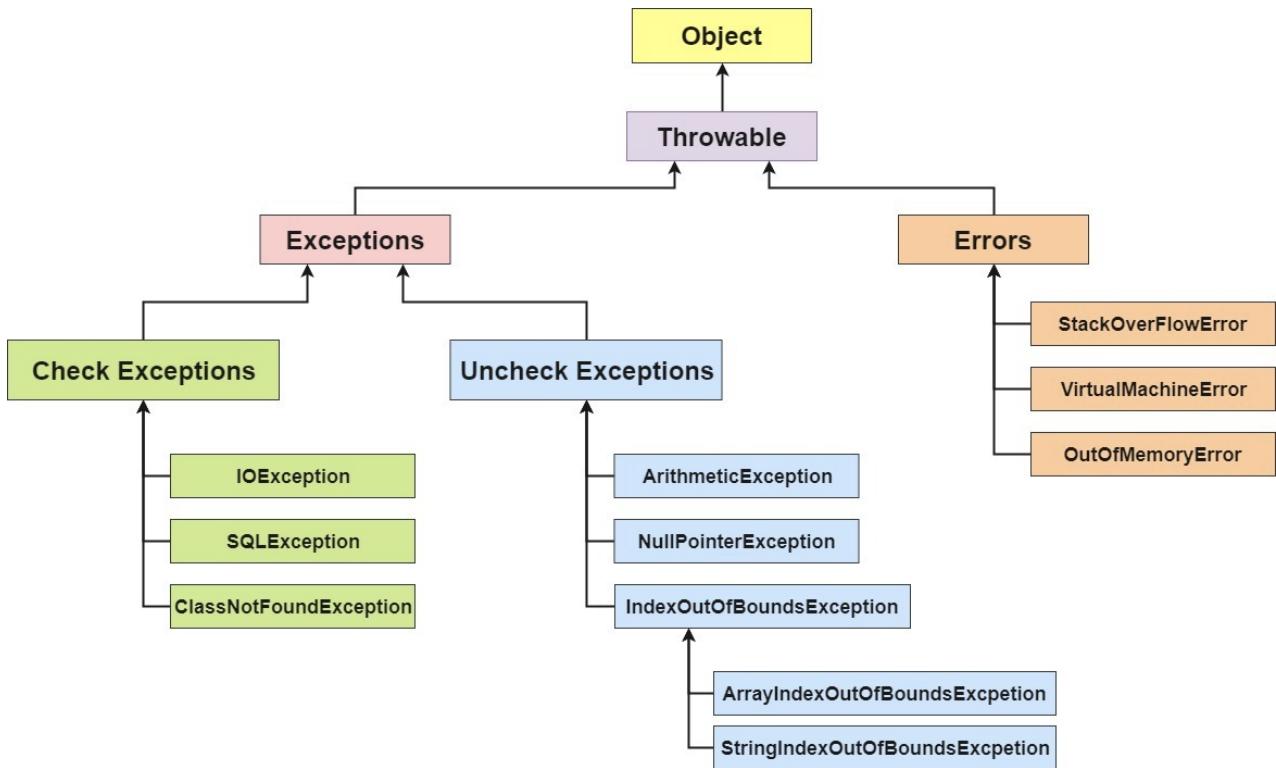
## 2.1.28. 关于 final 关键字的一些总结

`final` 关键字主要用在三个地方：变量、方法、类。

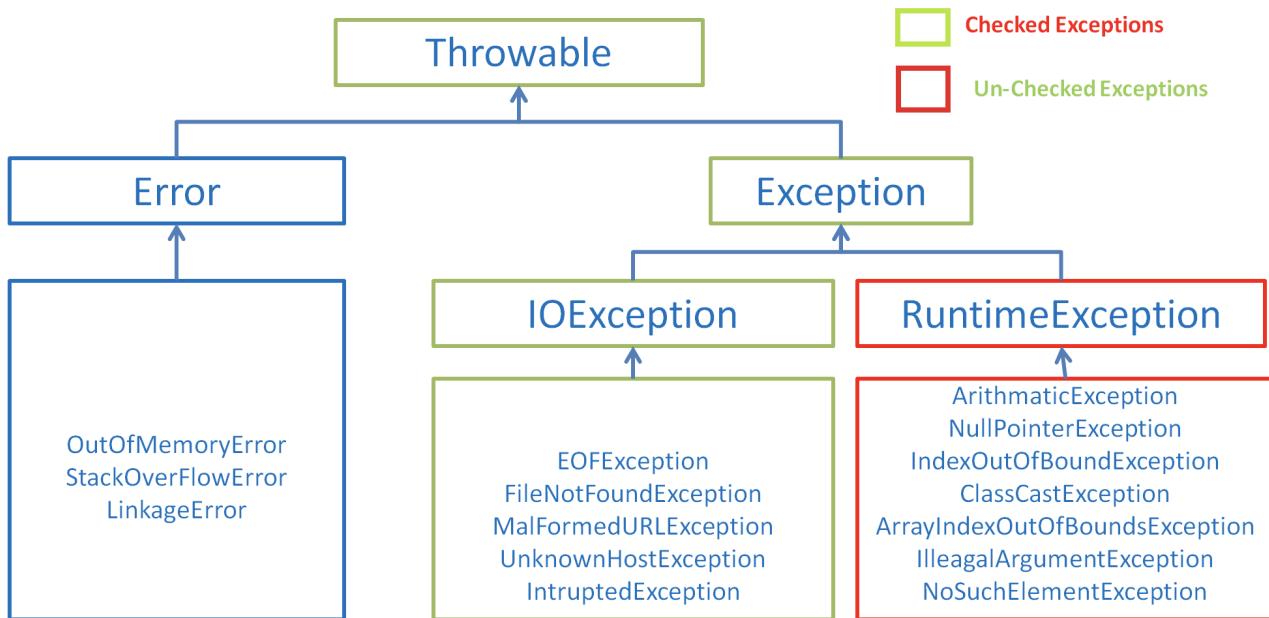
- 对于一个 `final` 变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。
- 当用 `final` 修饰一个类时，表明这个类不能被继承。`final` 类中的所有成员方法都会被隐式地指定为 `final` 方法。
- 使用 `final` 方法的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；第二个原因是效率。在早期的 Java 实现版本中，会将 `final` 方法转为内嵌调用。但是如果方法过于庞大，可能看不到内嵌调用带来的任何性能提升（现在的 Java 版本已经不需要使用 `final` 方法进行这些优化了）。类中所有的 `private` 方法都隐式地指定为 `final`。

## 2.1.29. Java 中的异常处理

### 2.1.29.1. Java 异常类层次结构图



图片来自：<https://simplesnippets.tech/exception-handling-in-java-part-1/>



图片来自：<https://chercher.tech/java-programming/exceptions-java>

在 Java 中，所有的异常都有一个共同的祖先 `java.lang` 包中的 `Throwable` 类。 `Throwable` 类有两个重要的子类 `Exception`（异常）和 `Error`（错误）。 `Exception` 能被程序本身处理(`try-catch`)，`Error` 是无法处理的(只能尽量避免)。

`Exception` 和 `Error` 二者都是 Java 异常处理的重要子类，各自都包含大量子类。

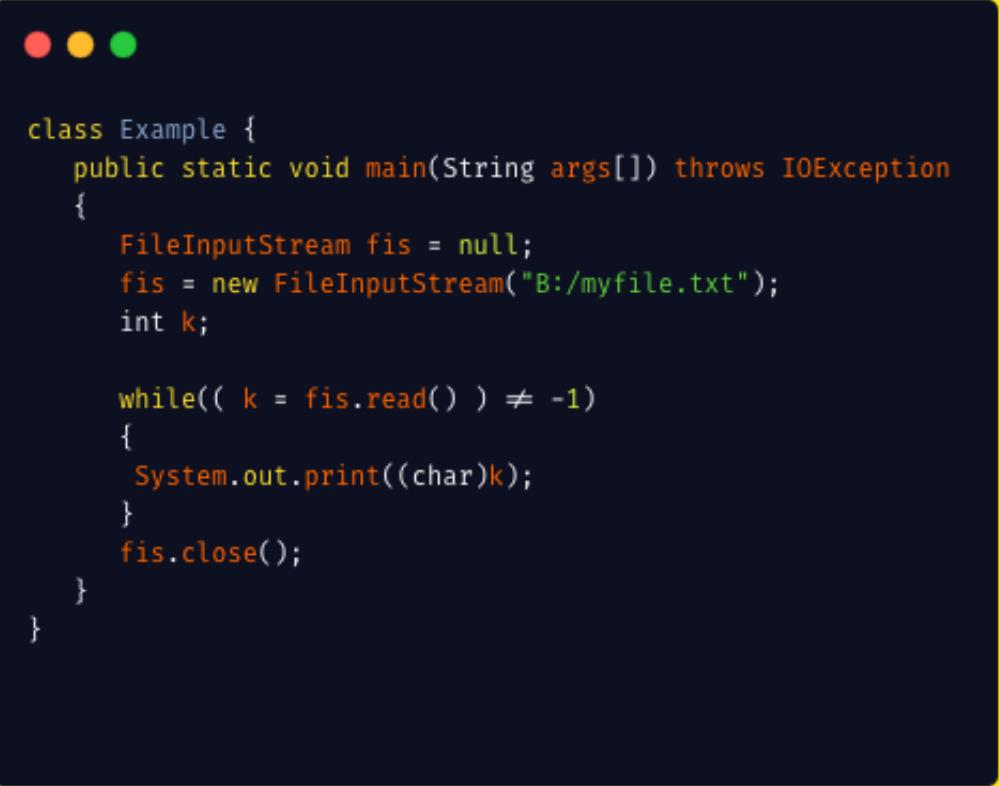
- `Exception` :程序本身可以处理的异常，可以通过 `catch` 来进行捕获。`Exception` 又可以分

为 受检查异常(必须处理) 和 不受检查异常(可以不处理)。

- **Error** : `Error` 属于程序无法处理的错误，我们没办法通过 `catch` 来进行捕获。例如，Java 虚拟机运行错误（`Virtual MachineError`）、虚拟机内存不够错误（`OutOfMemoryError`）、类定义错误（`NoClassDefFoundError`）等。这些异常发生时，Java 虚拟机（JVM）一般会选择线程终止。

## 受检查异常

Java 代码在编译过程中，如果受检查异常没有被 `catch / throw` 处理的话，就没办法通过编译。比如下面这段 IO 操作的代码。



```
class Example {
    public static void main(String args[]) throws IOException
    {
        FileInputStream fis = null;
        fis = new FileInputStream("B:/myfile.txt");
        int k;

        while(( k = fis.read() ) != -1)
        {
            System.out.print((char)k);
        }
        fis.close();
    }
}
```

除了 `RuntimeException` 及其子类以外，其他的 `Exception` 类及其子类都属于检查异常。常见的受检查异常有：IO 相关的异常、`ClassNotFoundException`、`SQLException` ...。

## 不受检查异常

Java 代码在编译过程中，我们即使不处理不受检查异常也可以正常通过编译。

`RuntimeException` 及其子类都统称为非受检查异常，例如：`NullPointerException`、`NumberFormatException`（字符串转换为数字）、`ArrayIndexOutOfBoundsException`（数组越界）、`ClassCastException`（类型转换错误）、`ArithmaticException`（算术错误）等。

## 2.1.29.2. `Throwable` 类常用方法

- `public string getMessage()` : 返回异常发生时的简要描述
- `public string toString()` : 返回异常发生时的详细信息
- `public string getLocalizedMessage()` : 返回异常对象的本地化信息。使用 `Throwable` 的子类覆盖这个方法，可以生成本地化信息。如果子类没有覆盖该方法，则该方法返回的信息与 `getMessage()` 返回的结果相同
- `public void printStackTrace()` : 在控制台上打印 `Throwable` 对象封装的异常信息

## 2.1.29.3. 异常处理总结

- `try` 块：用于捕获异常。其后可接零个或多个 `catch` 块，如果没有 `catch` 块，则必须跟一个 `finally` 块。
- `catch` 块：用于处理 `try` 捕获到的异常。
- `finally` 块：无论是否捕获或处理异常，`finally` 块里的语句都会被执行。当在 `try` 块或 `catch` 块中遇到 `return` 语句时，`finally` 语句块将在方法返回之前被执行。

在以下 3 种特殊情况下，`finally` 块不会被执行：

1. 在 `try` 或 `finally` 块中用了 `System.exit(int)` 退出程序。但是，如果 `System.exit(int)` 在异常语句之后，`finally` 还是会被执行
2. 程序所在的线程死亡。
3. 关闭 CPU。

下面这部分内容来自 issue:<https://github.com/Snailclimb/JavaGuide/issues/190>。

注意：当 `try` 语句和 `finally` 语句中都有 `return` 语句时，在方法返回之前，`finally` 语句的内容将被执行，并且 `finally` 语句的返回值将会覆盖原始的返回值。如下：

```
public static int f(int value) {  
    try {  
        return value * value;  
    } finally {  
        if (value == 2) {  
            return 0;  
        }  
    }  
}
```

如果调用 `f(2)`，返回值将是 0，因为 `finally` 语句的返回值覆盖了 `try` 语句块的返回值。

### 2.1.30. Java 序列化中如果有些字段不想进行序列化，怎么办？

对于不想进行序列化的变量，使用 `transient` 关键字修饰。

`transient` 关键字的作用是：阻止实例中那些用此关键字修饰的变量序列化；当对象被反序列化时，被 `transient` 修饰的变量值不会被持久化和恢复。`transient` 只能修饰变量，不能修饰类和方法。

### 2.1.31. 获取用键盘输入常用的两种方法

方法 1：通过 `Scanner`

```
Scanner input = new Scanner(System.in);  
String s = input.nextLine();  
input.close();
```

方法 2：通过 `BufferedReader`

```
BufferedReader input = new BufferedReader(new InputStreamReader(System.in));  
String s = input.readLine();
```

## 2.1.32. Java 中 IO 流

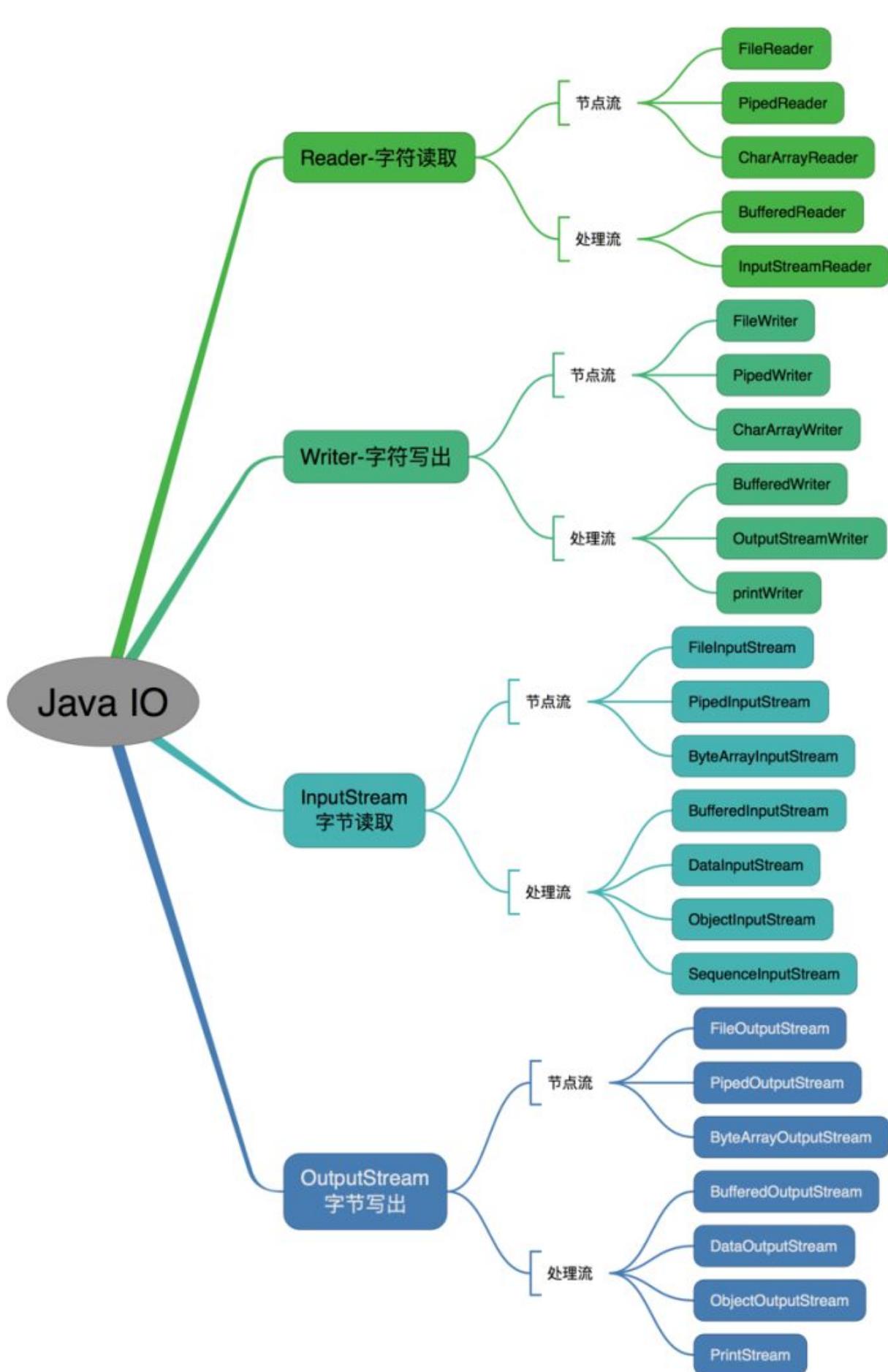
### 2.1.32.1. Java 中 IO 流分为几种？

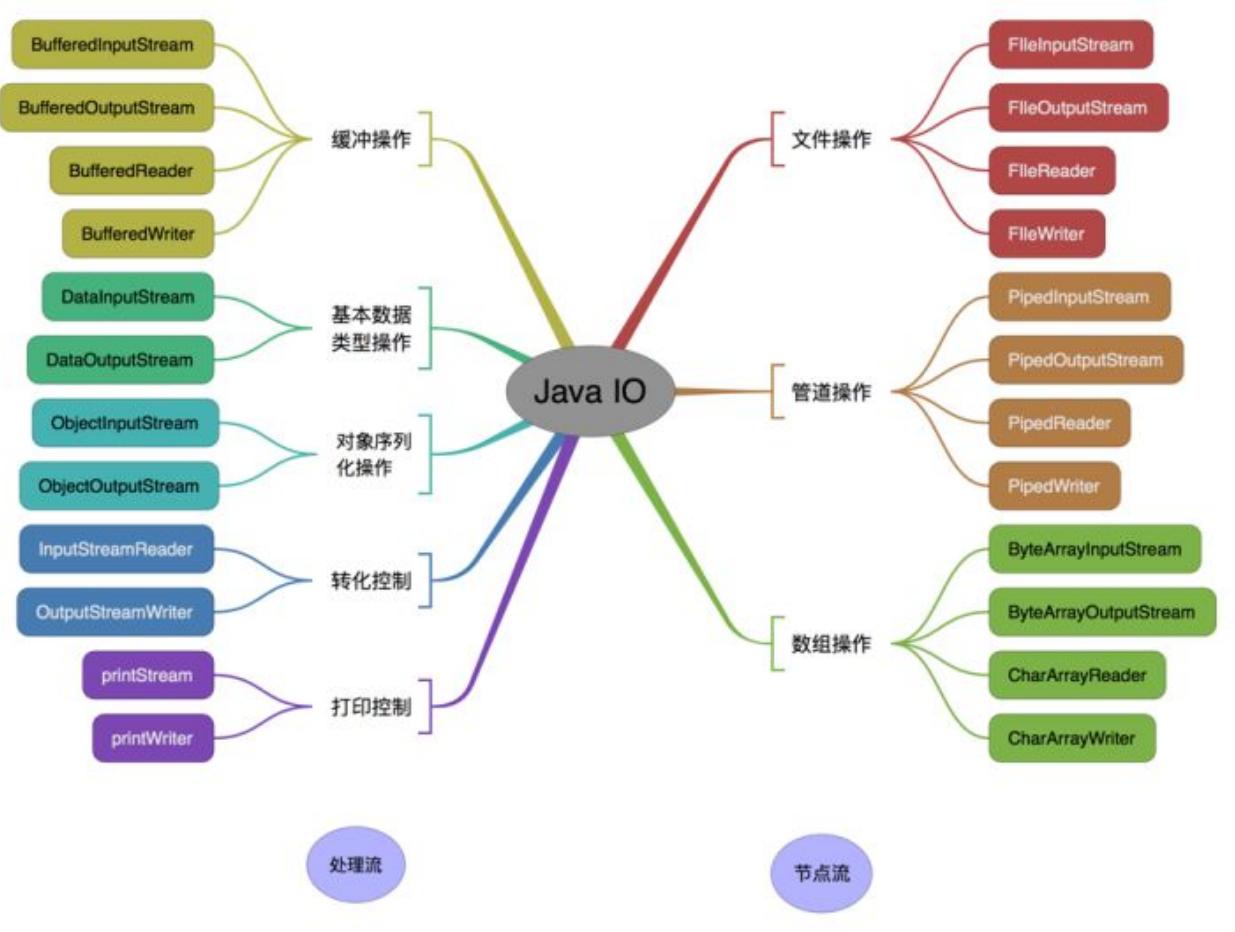
- 按照流的流向分，可以分为输入流和输出流；
- 按照操作单元划分，可以划分为字节流和字符流；
- 按照流的角色划分为节点流和处理流。

Java Io 流共涉及 40 多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存在非常紧密的联系，Java Io 流的 40 多个类都是从如下 4 个抽象类基类中派生出来的。

- `InputStream/Reader`: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。
- `OutputStream/Writer`: 所有输出流的基类，前者是字节输出流，后者是字符输出流。

按操作方式分类结构图：





### 2.1.32.2. 既然有了字节流,为什么还要有字符流?

问题本质想问：不管是文件读写还是网络发送接收，信息的最小存储单元都是字节，那为什么 I/O 流操作要分为字节流操作和字符流操作呢？

回答：字符流是由 Java 虚拟机将字节转换得到的，问题就出在这个过程还算是非常耗时，并且，如果我们不知道编码类型就很容易出现乱码问题。所以，I/O 流就干脆提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。如果音频文件、图片等媒体文件用字节流比较好，如果涉及到字符的话使用字符流比较好。

### 2.1.32.3. BIO,NIO,AIO 有什么区别?

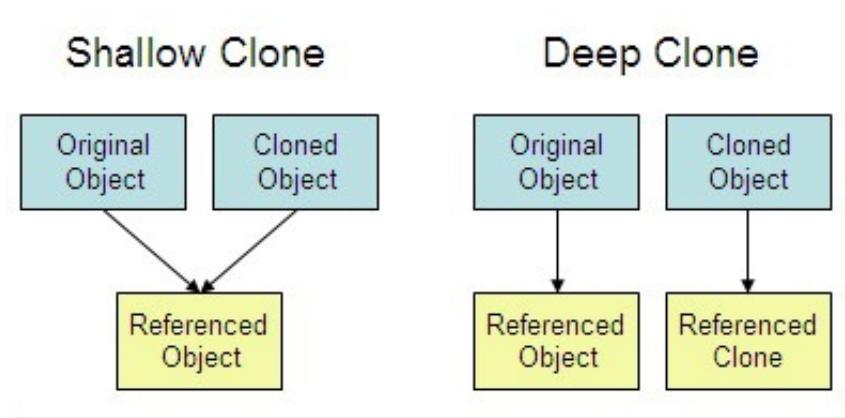
- **BIO (Blocking I/O):** 同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机 1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高的 I/O 处理模型来应对更高的并发量。
- **NIO (Non-blocking/New I/O):** NIO 是一种同步非阻塞的 I/O 模型，在 Java 1.4 中引入了 NIO 框架，对应 java.nio 包，提供了 Channel , Selector , Buffer 等抽象。NIO 中的 N 可以理解为 Non-blocking，不单纯是 New。它支持面向缓冲的，基于通道的 I/O 操作方法。NIO 提供了与传统 BIO 模型中的 Socket 和 ServerSocket 相对应的 SocketChannel 和

ServerSocketChannel 两种不同的套接字通道实现,两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样,比较简单,但是性能和可靠性都不好;非阻塞模式正好与之相反。对于低负载、低并发的应用程序,可以使用同步阻塞 I/O 来提升开发速率和更好的维护性;对于高负载、高并发的(网络)应用,应使用 NIO 的非阻塞模式来开发

- **AIO (Asynchronous I/O):** AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的 IO 模型。异步 IO 是基于事件和回调机制实现的,也就是应用操作之后会直接返回,不会堵塞在那里,当后台处理完成,操作系统会通知相应的线程进行后续的操作。AIO 是异步 IO 的缩写,虽然 NIO 在网络操作中,提供了非阻塞的方法,但是 NIO 的 IO 行为还是同步的。对于 NIO 来说,我们的业务线程是在 IO 操作准备好时,得到通知,接着就由这个线程自行进行 IO 操作,IO 操作本身是同步的。查阅网上相关资料,我发现就目前来说 AIO 的应用还不是很广泛,Netty 之前也尝试使用过 AIO,不过又放弃了。

### 2.1.33. 深拷贝 vs 浅拷贝

1. 浅拷贝: 对基本数据类型进行值传递,对引用数据类型进行引用传递般的拷贝,此为浅拷贝。
2. 深拷贝: 对基本数据类型进行值传递,对引用数据类型,创建一个新的对象,并复制其内容,此为深拷贝。



### 2.1.34. 参考

- <https://stackoverflow.com/questions/1906445/what-is-the-difference-between-jdk-and-jre>
- <https://www.educba.com/oracle-vs-openjdk/>
- <https://stackoverflow.com/questions/22358071/differences-between-oracle-jdk-and-openjdk?answertab=active#tab-top>

### 2.1.35. 公众号

如果大家想要实时关注我更新的文章以及分享的干货的话,可以关注我的公众号。

《JavaGuide 面试突击版》:由本文档衍生的专为面试而生的《JavaGuide 面试突击版》版本  
公众号后台回复 "Java 面试突击" 即可免费领取!

Java 工程师必备学习资源: 一些 Java 工程师常用学习资源公众号后台回复关键字“1”即可免费无套路获取。



## 2.2. Java集合

作者: Guide哥。

介绍: Github 70k Star 项目 [JavaGuide](#) (公众号同名) 作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

### 2.2.1. 说说List,Set,Map三者的区别?

- List (对付顺序的好帮手): 存储的元素是有序的、可重复的。
- Set (注重独一无二的性质): 存储的元素是无序的、不可重复的。
- Map (用 Key 来搜索的专家): 使用键值对 (key-value) 存储, 类似于数学上的函数  $y=f(x)$ , “x”代表 key, “y”代表 value, Key 是无序的、不可重复的, value 是无序的、可重复的, 每个键最多映射到一个值。

### 2.2.2. ArrayList 与 LinkedList 区别?

1. 是否保证线程安全: ArrayList 和 LinkedList 都是不同步的, 也就是不保证线程安全;
2. 底层数据结构: ArrayList 底层使用的是 Object 数组; LinkedList 底层使用的是 双向链表 数据结构 (JDK1.6 之前为循环链表, JDK1.7 取消了循环。注意双向链表和双向循环链表的区别, 下面有介绍到! )
3. 插入和删除是否受元素位置的影响: ① ArrayList 采用数组存储, 所以插入和删除元素的时间复杂度受元素位置的影响。比如: 执行 add(E e) 方法的时候, ArrayList 会默认在将

指定的元素追加到此列表的末尾，这种情况时间复杂度就是  $O(1)$ 。但是如果要在指定位置  $i$  插入和删除元素的话（`add(int index, E element)`）时间复杂度就为  $O(n-i)$ 。因为在进行上述操作的时候集合中第  $i$  和第  $i$  个元素之后的  $(n-i)$  个元素都要执行向后位/向前移一位的操作。②

`LinkedList` 采用链表存储，所以对于 `add(E e)` 方法的插入，删除元素时间复杂度不受元素位置的影响，近似  $O(1)$ ，如果是要在指定位置  $i$  插入和删除元素的话（`(add(int index, E element))` 时间复杂度近似为  $O(n)$ ）因为需要先移动到指定位置再插入。

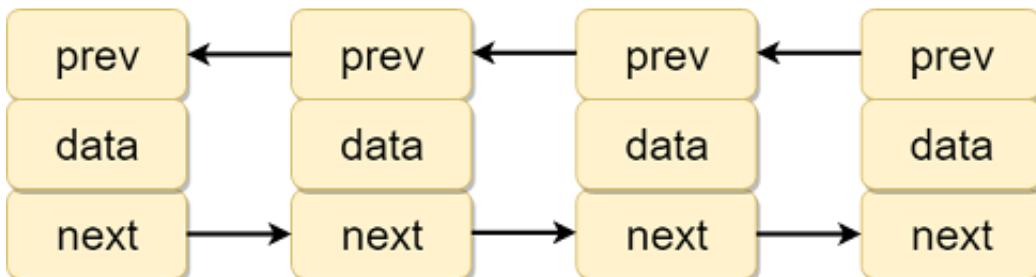
4. 是否支持快速随机访问： `LinkedList` 不支持高效的随机元素访问，而 `ArrayList` 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于 `get(int index)` 方法)。
5. 内存空间占用： `ArrayList` 的空间浪费主要体现在在 `list` 列表的结尾会预留一定的容量空间，而 `LinkedList` 的空间花费则体现在它的每一个元素都需要消耗比 `ArrayList` 更多的空间（因为要存放直接后继和直接前驱以及数据）。

### 2.2.2.1. 补充内容:双向链表和双向循环链表

双向链表：包含两个指针，一个 `prev` 指向前一个节点，一个 `next` 指向后一个节点。

另外推荐一篇把双向链表讲清楚的文章：<https://juejin.im/post/5b5d1a9af265da0f47352f14>

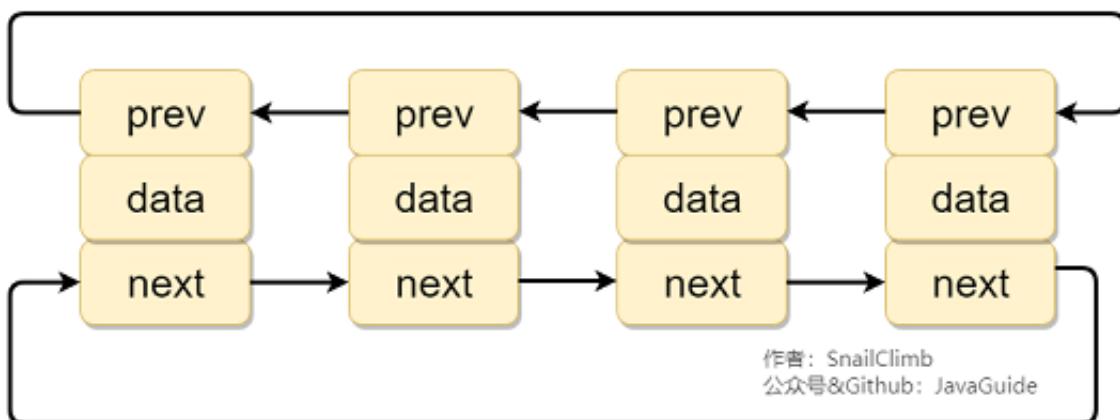
## 双向链表



作者: SnailClimb  
公众号&Github: JavaGuide

双向循环链表：最后一个节点的 `next` 指向 `head`，而 `head` 的 `prev` 指向最后一个节点，构成一个环。

## 双向循环链表



### 2.2.2.2. 补充内容:RandomAccess 接口

```
public interface RandomAccess {  
}
```

查看源码我们发现实际上 `RandomAccess` 接口中什么都没有定义。所以，在我看来 `RandomAccess` 接口不过是一个标识罢了。标识什么？标识实现这个接口的类具有随机访问功能。

在 `binarySearch()` 方法中，它要判断传入的 `list` 是否 `RandomAccess` 的实例，如果是，调用 `indexedBinarySearch()` 方法，如果不是，那么调用 `iteratorBinarySearch()` 方法

```
public static <T>  
int binarySearch(List<? extends Comparable<? super T>> list, T key) {  
    if (list instanceof RandomAccess || list.size() < BINARYSEARCH_THRESHOLD)  
        return Collections.indexedBinarySearch(list, key);  
    else  
        return Collections.iteratorBinarySearch(list, key);  
}
```

`ArrayList` 实现了 `RandomAccess` 接口，而 `LinkedList` 没有实现。为什么呢？我觉得还是和底层数据结构有关！`ArrayList` 底层是数组，而 `LinkedList` 底层是链表。数组天然支持随机访问，时间复杂度为  $O(1)$ ，所以称为快速随机访问。链表需要遍历到特定位置才能访问特定位置的元素，时间复杂度为  $O(n)$ ，所以不支持快速随机访问。, `ArrayList` 实现了 `RandomAccess` 接口，就表明了他具有快速随机访问功能。`RandomAccess` 接口只是标识，并不是说 `ArrayList`

实现 RandomAccess 接口才具有快速随机访问功能的！

**2.2.3. ArrayList 与 Vector 区别呢?为什么要用ArrayList取代Vector呢?**

- `ArrayList` 是 `List` 的主要实现类，底层使用 `Object[]` 存储，适用于频繁的查找工作，线程不安全；
  - `Vector` 是 `List` 的古老实现类，底层使用 `Object[]` 存储，线程安全的。

#### 2.2.4. 说一说 ArrayList 的扩容机制吧

详见笔主的这篇文章:通过源码一步一步分析 ArrayList 扩容机制

## 2.2.5. HashMap 和 Hashtable 的区别

- 线程是否安全：** `HashMap` 是非线程安全的，`HashTable` 是线程安全的，因为 `HashTable` 内部的方法基本都经过 `synchronized` 修饰。（如果你要保证线程安全的话就使用 `ConcurrentHashMap` 吧！）；
  - 效率：** 因为线程安全的问题，`HashMap` 要比 `HashTable` 效率高一点。另外，`HashTable` 基本被淘汰，不要在代码中使用它；
  - 对 Null key 和 Null value 的支持：** `HashMap` 可以存储 null 的 key 和 value，但 null 作为键只能有一个，null 作为值可以有多个；`HashTable` 不允许有 null 键和 null 值，否则会抛出 `NullPointerException`。
  - 初始容量大小和每次扩充容量大小的不同：** ① 创建时如果不指定容量初始值，`Hashtable` 默认的初始大小为 11，之后每次扩充，容量变为原来的  $2n+1$ 。`HashMap` 默认的初始化大小为 16。之后每次扩充，容量变为原来的 2 倍。② 创建时如果给定了容量初始值，那么 `Hashtable` 会直接使用你给定的大小，而 `HashMap` 会将其扩充为 2 的幂次方大小（`HashMap` 中的 `tableSizeFor()` 方法保证，下面给出了源代码）。也就是说 `HashMap` 总是使用 2 的幂作为哈希表的大小，后面会介绍到为什么是 2 的幂次方。
  - 底层数据结构：** JDK1.8 以后的 `HashMap` 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间。`Hashtable` 没有这样的机制。

**HashMap** 中带有初始容量的构造函数：

```
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);
}
public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}
```

下面这个方法保证了 `HashMap` 总是使用2的幂作为哈希表的大小。

```
/**
 * Returns a power of two size for the given target capacity.
 */
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;
}
```

## 2.2.6. `HashMap` 和 `HashSet`区别

如果你看过 `HashSet` 源码的话就应该知道：`HashSet` 底层就是基于 `HashMap` 实现的。

(`HashSet` 的源码非常非常少，因为除了 `clone()`、`writeObject()`、`readObject()` 是 `HashSet` 自己不得不实现之外，其他方法都是直接调用 `HashMap` 中的方法。

| HashMap                             | HashSet   |
|-------------------------------------|---|
| 实现了 Map 接口                          | 实现 Set 接口   |
| 存储键值对                               | 仅存储对象   |
| 调用 put() 向 map 中<br>添加元素            | 调用 add() 方法向 Set 中添加元素  |
| HashMap 使用键<br>(Key) 计算<br>hashcode | HashSet 使用成员对象来计算 hashcode 值, 对于两个对象来说<br>hashcode 可能相同, 所以 equals() 方法用来判断对象的相等性 |

## 2.2.7. HashSet如何检查重复

以下内容摘自我的 Java 启蒙书《Head fist java》第二版：

当你把对象加入 HashSet 时, HashSet 会先计算对象的 hashcode 值来判断对象加入的位置, 同时也会与其他加入的对象的 hashcode 值作比较, 如果没有相符的 hashcode , HashSet 会假设对象没有重复出现。但是如果发现有相同 hashcode 值的对象, 这时会调用 equals() 方法来检查 hashcode 相等的对象是否真的相同。如果两者相同, HashSet 就不会让加入操作成功。

### hashCode() 与 equals() 的相关规定:

1. 如果两个对象相等, 则 hashcode 一定也是相同的
2. 两个对象相等, 对两个 equals() 方法返回 true
3. 两个对象有相同的 hashcode 值, 它们也不一定是相等的
4. 综上, equals() 方法被覆盖过, 则 hashCode() 方法也必须被覆盖
5. hashCode() 的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode() , 则该 class 的两个对象无论如何都不会相等 (即使这两个对象指向相同的数据) 。

### ==与 equals 的区别

对于基本类型来说, == 比较的是值是否相等;

对于引用类型来说, == 比较的是两个引用是否指向同一个对象地址 (两者在内存中存放的地址 (堆内存地址) 是否指向同一个地方) ;

对于引用类型 (包括包装类型) 来说, equals 如果没有被重写, 对比它们的地址是否相等; 如果 equals()方法被重写 (例如 String) , 则比较的是地址里的内容。

作者：Guide哥。

介绍：Github 90k Star 项目 [JavaGuide](#)（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

## 2.2.8. HashMap的底层实现

### 2.2.8.1. JDK1.8 之前

JDK1.8 之前 HashMap 底层是 数组和链表 结合在一起使用也就是 链表散列。HashMap 通过 key 的 hashCode 经过扰动函数处理过后得到 hash 值，然后通过  $(n - 1) \& hash$  判断当前元素存放的位置（这里的 n 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。

所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的 hashCode() 方法 换句话说使用扰动函数之后可以减少碰撞。

JDK 1.8 HashMap 的 hash 方法源码：

JDK 1.8 的 hash 方法相比于 JDK 1.7 hash 方法更加简化，但是原理不变。

```
static final int hash(Object key) {
    int h;
    // key.hashCode(): 返回散列值也就是hashcode
    // ^ : 按位异或
    // >>>:无符号右移, 忽略符号位, 空位都以0补齐
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

对比一下 JDK1.7 的 HashMap 的 hash 方法源码.

```
static int hash(int h) {  
    // This function ensures that hashCode values differ only by  
    // constant multiples at each bit position have a bounded  
    // number of collisions (approximately 8 at default load factor).  
  
    h ^= (h >>> 20) ^ (h >>> 12);  
    return h ^ (h >>> 7) ^ (h >>> 4);  
}
```

相比于 JDK1.8 的 hash 方法，JDK 1.7 的 hash 方法的性能会稍差一点点，因为毕竟扰动了 4 次。

所谓“拉链法”就是：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。



### 2.2.8.2. JDK1.8 之后

相比于之前的版本，JDK1.8 之后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间。



TreeMap、TreeSet 以及 JDK1.8 之后的 HashMap 底层都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

### 2.2.9. HashMap 的长度为什么是2的幂次方

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。我们上面也讲到了过了，Hash 值的范围值-2147483648到2147483647，前后加起来大概40亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个40亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法是“ $(n - 1) \& hash$ ”。(n代表数组长度)。这也就解释了 HashMap 的长度为什么是2的幂次方。

这个算法应该如何设计呢？

我们首先可能会想到采用%取余的操作来实现。但是，重点来了：“取余(%)操作中如果除数是2的幂次则等价于与其除数减一的与(&)操作（也就是说  $hash \% length == hash \& (length - 1)$  的前提是  $length$  是2的n次方；）。”并且采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 `HashMap` 的长度为什么是2的幂次方。

## 2.2.10. `HashMap` 多线程操作导致死循环问题

主要原因在于并发下的Rehash会造成元素之间会形成一个循环链表。不过，jdk 1.8 后解决了这个问题，但是还是不建议在多线程下使用 `HashMap`,因为多线程下使用 `HashMap` 还是会存在其他问题比如数据丢失。并发环境下推荐使用 `ConcurrentHashMap` 。

详情请查看：<https://coolshell.cn/articles/9606.html>

## 2.2.11. `ConcurrentHashMap` 和 `Hashtable` 的区别

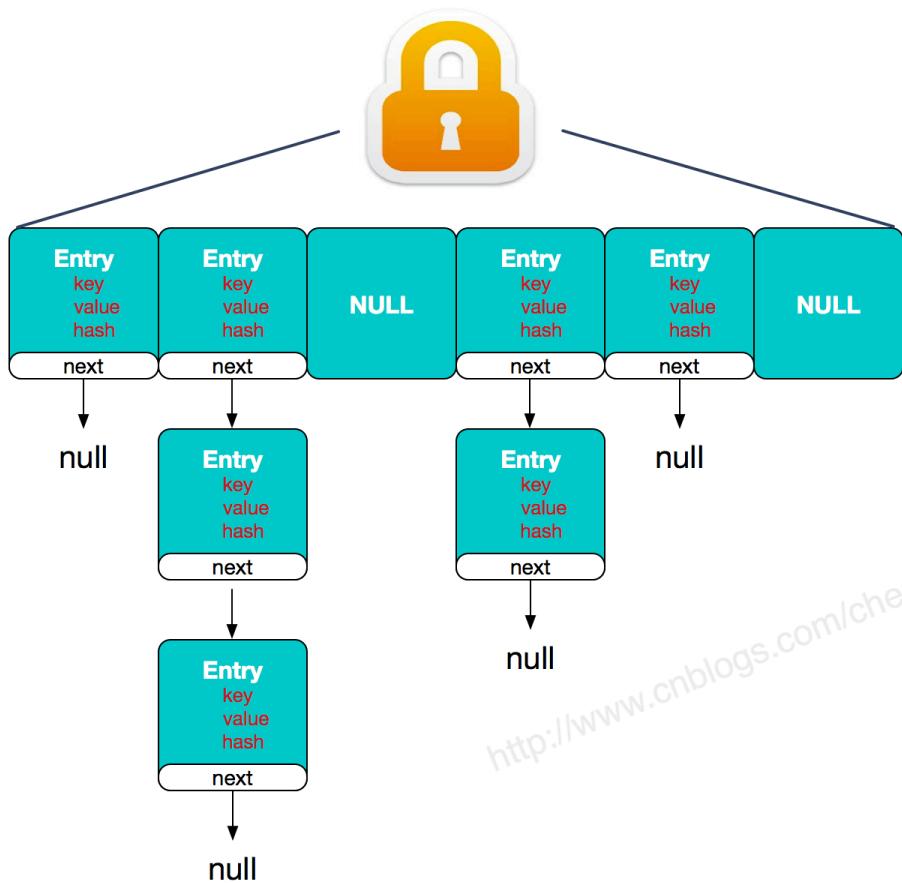
`ConcurrentHashMap` 和 `Hashtable` 的区别主要体现在实现线程安全的方式上不同。

- **底层数据结构：** JDK1.7 的 `ConcurrentHashMap` 底层采用 分段的数组+链表 实现，JDK1.8 采用的数据结构跟 `HashMap` 的结构一样，数组+链表/红黑二叉树。`Hashtable` 和 JDK1.8 之前的 `HashMap` 的底层数据结构类似都是采用 数组+链表 的形式，数组是 `HashMap` 的主体，链表则是主要为了解决哈希冲突而存在的；
- **实现线程安全的方式（重要）：** ① 在 JDK1.7 的时候，`ConcurrentHashMap` （分段锁）对整个桶数组进行了分割分段( Segment )，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。到了 JDK1.8 的时候已经摒弃了 Segment 的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 `synchronized` 和 CAS 来操作。（JDK1.6 以后对 `synchronized` 锁做了很多优化）整个看起来就像是优化过且线程安全的 `HashMap`，虽然在 JDK1.8 中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；② **Hashtable (同一把锁)**：使用 `synchronized` 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 `put` 添加元素，另一个线程不能使用 `put` 添加元素，也不能使用 `get`，竞争会越来越激烈效率越低。

两者的对比图：

**HashTable:**

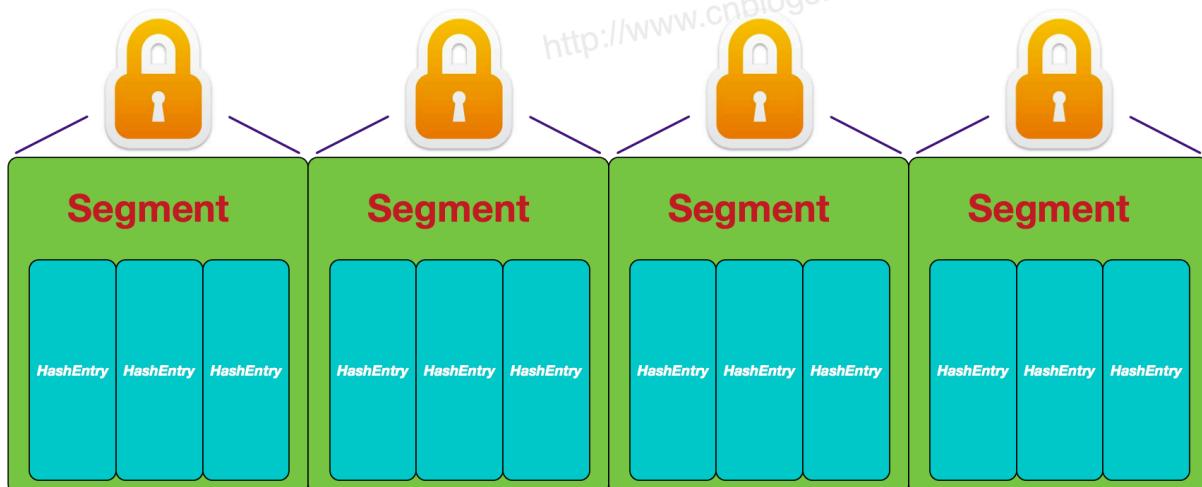
## HashTable 全表锁



[http://www.cnblogs.com/chengxiao/p/6842045.html>](http://www.cnblogs.com/chengxiao/p/6842045.html)

## JDK1.7 的 ConcurrentHashMap:

### ConcurrentHashMap 分段锁



[http://www.cnblogs.com/chengxiao/p/6842045.html>](http://www.cnblogs.com/chengxiao/p/6842045.html)

## JDK1.8 的 ConcurrentHashMap：

JDK1.8 的 ConcurrentHashMap 不再是 Segment 数组 + HashEntry 数组 + 链表，而是 Node 数组 + 链表 / 红黑树。不过，Node 只能用于链表的情况，红黑树的情况需要使用 TreeNode。当冲突链表达到一定长度时，链表会转换成红黑树。

### 2.2.12. ConcurrentHashMap 线程安全的具体实现方式/底层具体实现

#### 2.2.12.1. JDK1.7 (上面有示意图)

首先将数据分为一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据时，其他段的数据也能被其他线程访问。

ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。

Segment 实现了 ReentrantLock，所以 Segment 是一种可重入锁，扮演锁的角色。HashEntry 用于存储键值对数据。

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

一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和 HashMap 类似，是一种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，每个 Segment 守护着一个 HashEntry 数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 的锁。

#### 2.2.12.2. JDK1.8 (上面有示意图)

ConcurrentHashMap 取消了 Segment 分段锁，采用 CAS 和 synchronized 来保证并发安全。数据结构跟 HashMap1.8 的结构类似，数组+链表/红黑二叉树。Java 8 在链表长度超过一定阈值(8)时将链表(寻址时间复杂度为 O(N))转换为红黑树(寻址时间复杂度为 O(log(N)))

synchronized 只锁定当前链表或红黑二叉树的首节点，这样只要 hash 不冲突，就不会产生并发，效率又提升 N 倍。

## 2.2.13. 比较 HashSet、LinkedHashSet 和 TreeSet 三者的异同

HashSet 是 Set 接口的主要实现类， HashSet 的底层是 HashMap，线程不安全的，可以存储 null 值；

LinkedHashSet 是 HashSet 的子类，能够按照添加的顺序遍历；

TreeSet 底层使用红黑树，能够按照添加元素的顺序进行遍历，排序的方式有自然排序和定制排序。

## 2.2.14. 集合框架底层数据结构总结

先来看一下 Collection 接口下面的集合。

### 2.2.14.1. List

- ArrayList : Object[] 数组
- Vector : Object[] 数组
- LinkedList : 双向链表(JDK1.6 之前为循环链表，JDK1.7 取消了循环)

### 2.2.14.2. Set

- HashSet (无序, 唯一) : 基于 HashMap 实现的，底层采用 HashMap 来保存元素
- LinkedHashSet : LinkedHashSet 是 HashSet 的子类，并且其内部是通过 LinkedHashMap 来实现的。有点类似于我们之前说的 LinkedHashMap 其内部是基于 HashMap 实现一样，不过还是有一点点区别的
- TreeSet (有序, 唯一) : 红黑树(自平衡的排序二叉树)

再来看看 Map 接口下面的集合。

### 2.2.14.3. Map

- HashMap : JDK1.8 之前 HashMap 由数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）。JDK1.8 以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间
- LinkedHashMap : LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外， LinkedHashMap 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。详细可以查看：[《LinkedHashMap 源码详细分析 \(JDK1.8\)》](#)
- Hashtable : 数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲

突而存在的

- TreeMap : 红黑树 (自平衡的排序二叉树)

## 2.2.15. 如何选用集合?

主要根据集合的特点来选用，比如我们需要根据键值获取到元素值时就选用 Map 接口下的集合，需要排序时选择 TreeMap，不需要排序时就选择 HashMap，需要保证线程安全就选用 ConcurrentHashMap。

当我们只需要存放元素值时，就选择实现 Collection 接口的集合，需要保证元素唯一时选择实现 Set 接口的集合比如 TreeSet 或 HashSet，不需要就选择实现 List 接口的比如 ArrayList 或 LinkedList，然后再根据实现这些接口的集合的特点来选用。

如果大家想要实时关注我更新的文章以及分享的干货的话，可以关注我的公众号。

《JavaGuide 面试突击版》：由本文档衍生的专为面试而生的《JavaGuide 面试突击版》版本  
公众号后台回复 "Java 面试突击" 即可免费领取！



## 2.3. 多线程

作者：Guide 哥。

介绍：Github 90k Star 项目 [JavaGuide](#)（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取 Java 工程师必备学习资料+面试突击 pdf。

## 2.3.1. 什么是线程和进程？

### 2.3.1.1. 何为进程？

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。

在 Java 中，当我们启动 main 函数时其实就是启动了一个 JVM 的进程，而 main 函数所在的线程就是这个进程中的一个线程，也称主线程。

如下图所示，在 windows 中通过查看任务管理器的方式，我们就可以清楚看到 window 当前运行的进程（.exe 文件的运行）。

The screenshot shows the Windows Task Manager with the 'Processes' tab selected. It displays a list of running applications and their resource usage. The columns are: Name, Status, CPU, Memory, Disk, and Network. The 'CPU' column shows usage percentages, while the other three columns show current values.

| 名称                                 | 状态 | 11%<br>CPU | 57%<br>内存 | 1%<br>磁盘 | 0%<br>网络 |
|------------------------------------|----|------------|-----------|----------|----------|
| <b>应用 (6)</b>                      |    |            |           |          |          |
| > Google Chrome (20)               |    | 2.4%       | 1,177.3   | 0.1 MB/秒 | 0 Mbps   |
| > Microsoft OneNote                |    | 0%         | 13.5 MB   | 0 MB/秒   | 0 Mbps   |
| > TIM (32 位)                       |    | 0.2%       | 85.5 MB   | 0 MB/秒   | 0 Mbps   |
| > WeChat (32 位) (3)                |    | 4.7%       | 109.4 MB  | 0.1 MB/秒 | 0 Mbps   |
| > 金山PDF (32 位)                     |    | 0%         | 192.0 MB  | 0 MB/秒   | 0 Mbps   |
| > 任务管理器                            |    | 0.4%       | 24.1 MB   | 0 MB/秒   | 0 Mbps   |
| <b>后台进程 (71)</b>                   |    |            |           |          |          |
| > 64-bit Synaptics Pointing Enh... |    | 0%         | 0.1 MB    | 0 MB/秒   | 0 Mbps   |
| Application Frame Host             |    | 0%         | 0.1 MB    | 0 MB/秒   | 0 Mbps   |
| BaiduNetdisk (32 位)                |    | 0%         | 17.3 MB   | 0 MB/秒   | 0 Mbps   |
| BaiduNetdiskHost (32 位)            |    | 0%         | 0.1 MB    | 0 MB/秒   | 0 Mbps   |
| COM Surrogate                      |    | 0%         | 1.0 MB    | 0 MB/秒   | 0 Mbps   |

简略信息(D)      结束任务(E)

### 2.3.1.2. 何为线程？

线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享进程的堆和方法区资源，但每个线程有自己的程序计数器、虚拟机栈和本地方法栈，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

Java 程序天生就是多线程程序，我们可以通过 JMX 来看一下一个普通的 Java 程序有哪些线程，代码如下。

```
public class MultiThread {  
    public static void main(String[] args) {  
        // 获取 Java 线程管理 MXBean  
        ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();  
        // 不需要获取同步的 monitor 和 synchronizer 信息，仅获取线程和线程堆栈信息  
        ThreadInfo[] threadInfos = threadMXBean.dumpAllThreads(false, false);  
        // 遍历线程信息，仅打印线程 ID 和线程名称信息  
        for (ThreadInfo threadInfo : threadInfos) {  
            System.out.println("[" + threadInfo.getThreadId() + "] " +  
                threadInfo.getThreadName());  
        }  
    }  
}
```

上述程序输出如下（输出内容可能不同，不用太纠结下面每个线程的作用，只用知道 main 线程执行 main 方法即可）：

```
[5] Attach Listener //添加事件  
[4] Signal Dispatcher // 分发处理给 JVM 信号的线程  
[3] Finalizer //调用对象 finalize 方法的线程  
[2] Reference Handler //清除 reference 线程  
[1] main //main 线程,程序入口
```

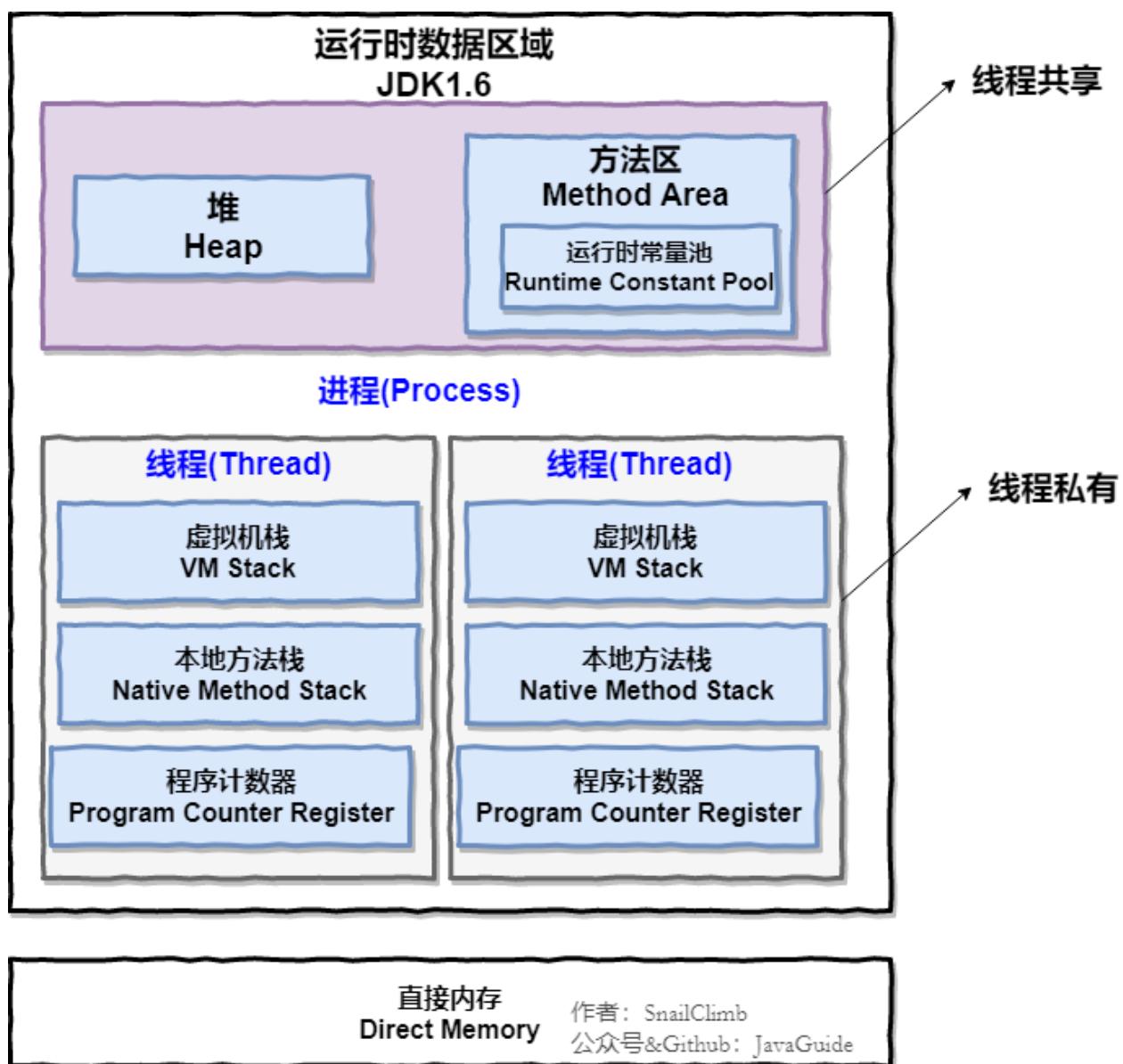
从上面的输出内容可以看出：一个 Java 程序的运行是 main 线程和多个其他线程同时运行。

## 2.3.2. 请简要描述线程与进程的关系,区别及优缺点?

从 JVM 角度说进程和线程之间的关系

### 2.3.2.1. 图解进程和线程的关系

下图是 Java 内存区域, 通过下图我们从 JVM 的角度来说一下线程和进程之间的关系。如果你对 Java 内存区域 (运行时数据区) 这部分知识不太了解的话可以阅读一下这篇文章: [《可能是把 Java 内存区域讲的最清楚的一篇文章》](#)



从上图可以看出: 一个进程中可以有多个线程, 多个线程共享进程的堆和方法区 (JDK1.8 之后的元空间) 资源, 但是每个线程有自己的程序计数器、虚拟机栈 和 本地方法栈。

总结: 线程 是 进程 划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的, 而各线程则不一定, 因为同一进程中的线程极有可能会相互影响。线程执行开销小, 但不利于资源的管理和保护; 而进程正相反

下面是该知识点的扩展内容！

下面来思考这样一个问题：为什么程序计数器、虚拟机栈和本地方法栈是线程私有的呢？为什么堆和方法区是线程共享的呢？

### 2.3.2.2. 程序计数器为什么是私有的？

程序计数器主要有下面两个作用：

1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

需要注意的是，如果执行的是 native 方法，那么程序计数器记录的是 undefined 地址，只有执行的是 Java 代码时程序计数器记录的才是下一条指令的地址。

所以，程序计数器私有主要是为了线程切换后能恢复到正确的执行位置。

### 2.3.2.3. 虚拟机栈和本地方法栈为什么是私有的？

- **虚拟机栈：**每个 Java 方法在执行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程，就对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程。
- **本地方法栈：**和虚拟机栈所发挥的作用非常相似，区别是：**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。**在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

所以，为了保证线程中的局部变量不被别的线程访问到，虚拟机栈和本地方法栈是线程私有的。

### 2.3.2.4. 一句话简单了解堆和方法区

堆和方法区是所有线程共享的资源，其中堆是进程中最大的一块内存，主要用于存放新创建的对象（所有对象都在这里分配内存），方法区主要用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

### 2.3.3. 说说并发与并行的区别？

- **并发：**同一时间段，多个任务都在执行（单位时间内不一定同时执行）；
- **并行：**单位时间内，多个任务同时执行。

## 2.3.4. 为什么要使用多线程呢？

先从总体上来说：

- **从计算机底层来说：** 线程可以比作是轻量级的进程，是程序执行的最小单位，线程间的切换和调度的成本远远小于进程。另外，多核 CPU 时代意味着多个线程可以同时运行，这减少了线程上下文切换的开销。
- **从当代互联网发展趋势来说：** 现在的系统动不动就要求百万级甚至千万级的并发量，而多线程并发编程正是开发高并发系统的基础，利用好多线程机制可以大大提高系统整体的并发能力以及性能。

再深入到计算机底层来探讨：

- **单核时代：** 在单核时代多线程主要是为了提高 CPU 和 IO 设备的综合利用率。举个例子：当只有一个线程的时候会导致 CPU 计算时，IO 设备空闲；进行 IO 操作时，CPU 空闲。我们可以简单地说这两者的利用率目前都是 50% 左右。但是当有两个线程的时候就不一样了，当一个线程执行 CPU 计算时，另外一个线程可以进行 IO 操作，这样两个的利用率就可以在理想情况下达到 100% 了。
- **多核时代：** 多核时代多线程主要是为了提高 CPU 利用率。举个例子：假如我们要计算一个复杂的任务，我们只用一个线程的话，CPU 只会一个 CPU 核心被利用到，而创建多个线程就可以让多个 CPU 核心被利用到，这样就提高了 CPU 的利用率。

## 2.3.5. 使用多线程可能带来什么问题？

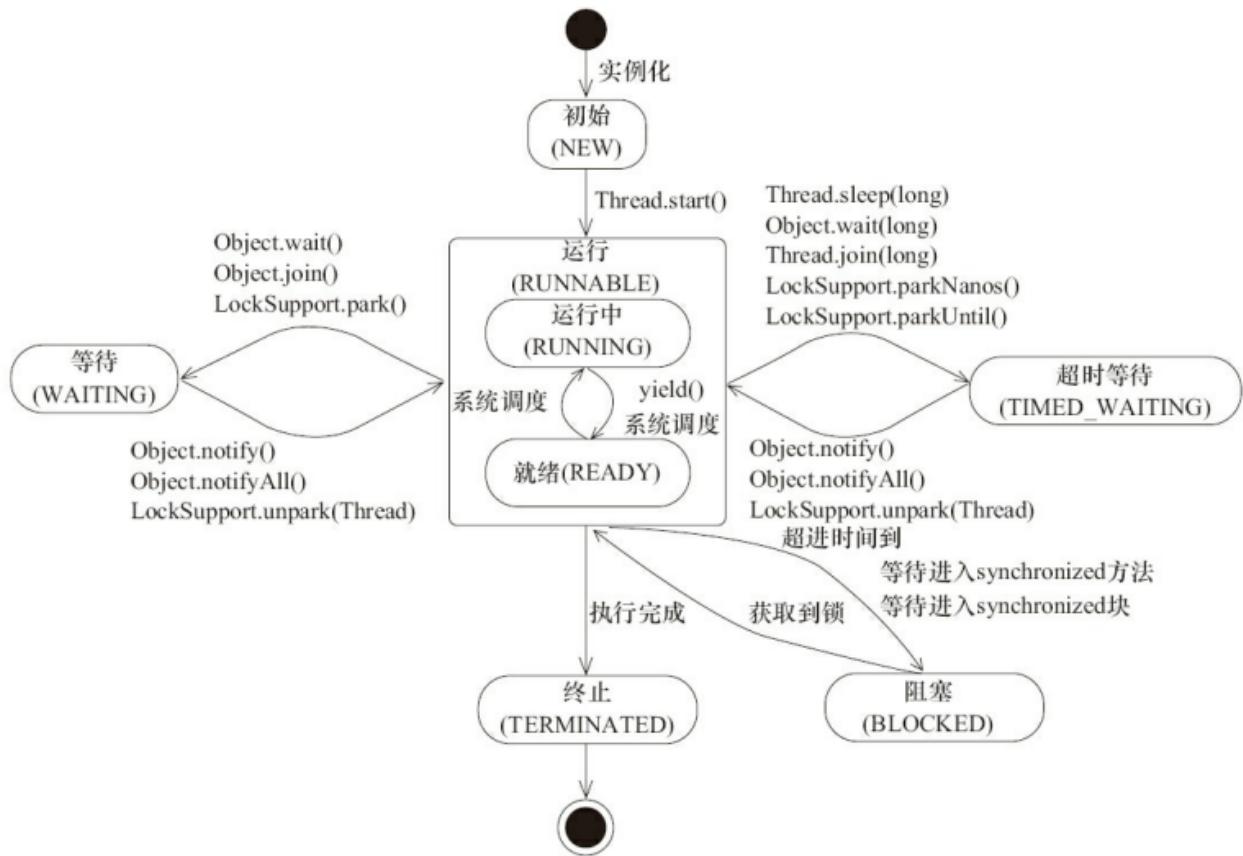
并发编程的目的就是为了能提高程序的执行效率提高程序运行速度，但是并发编程并不总是能提高程序运行速度的，而且并发编程可能会遇到很多问题，比如：**内存泄漏、上下文切换、死锁**。

## 2.3.6. 说说线程的生命周期和状态？

Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态（图源《Java 并发编程艺术》4.1.4 节）。

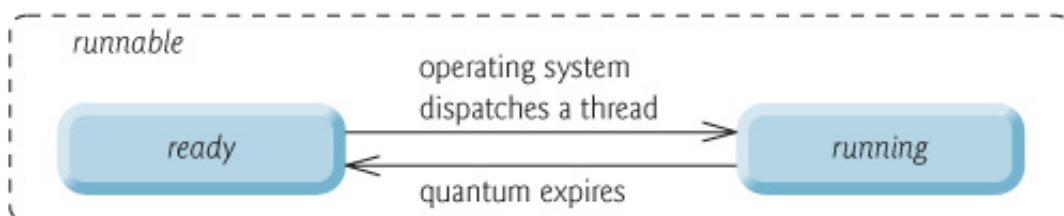
| 状态名称         | 说 明  |
|--------------|--|
| NEW          | 初始状态，线程被构建，但是还没有调用 start() 方法                      |
| RUNNABLE     | 运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”             |
| BLOCKED      | 阻塞状态，表示线程阻塞于锁                                      |
| WAITING      | 等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断） |
| TIME_WAITING | 超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的              |
| TERMINATED   | 终止状态，表示当前线程已经执行完毕                                  |

线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。Java 线程状态变迁如下图所示（图源《Java 并发编程艺术》4.1.4 节）：



由上图可以看出：线程创建之后它将处于 **NEW**（新建）状态，调用 `start()` 方法后开始运行，线程这时候处于 **RUNNABLE**（可运行）状态。可运行状态的线程获得了 CPU 时间片（timeslice）后就处于 **RUNNING**（运行）状态。

操作系统隐藏 Java 虚拟机（JVM）中的 **RUNNABLE** 和 **RUNNING** 状态，它只能看到 **RUNNABLE** 状态（图源：[HowToDoInJava: Java Thread Life Cycle and Thread States](#)），所以 Java 系统一般将这两个状态统称为 **RUNNABLE**（运行中）状态。



当线程执行 `wait()` 方法之后，线程进入 **WAITING**（等待）状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 **TIMED\_WAITING**（超时等待）状态相当于在等待状态的基础上增加了超时限制，比如通过 `sleep (long millis)` 方法或 `wait (long millis)` 方法可以将 Java 线程置于 **TIMED\_WAITING** 状态。当超时时间到达后 Java 线程将会返回到 **RUNNABLE** 状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到 **BLOCKED**（阻塞）状态。线程在执行 `Runnable` 的 `run()` 方法之后将会进入到 **TERMINATED**（终止）状态。

## 2.3.7. 什么是上下文切换?

多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。

概括来说就是：当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换回这个任务时，可以再加载这个任务的状态。**任务从保存到再加载的过程就是一次上下文切换。**

上下文切换通常是计算密集型的。也就是说，它需要相当可观的处理器时间，在每秒几十上百次的切换中，每次切换都需要纳秒量级的时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。

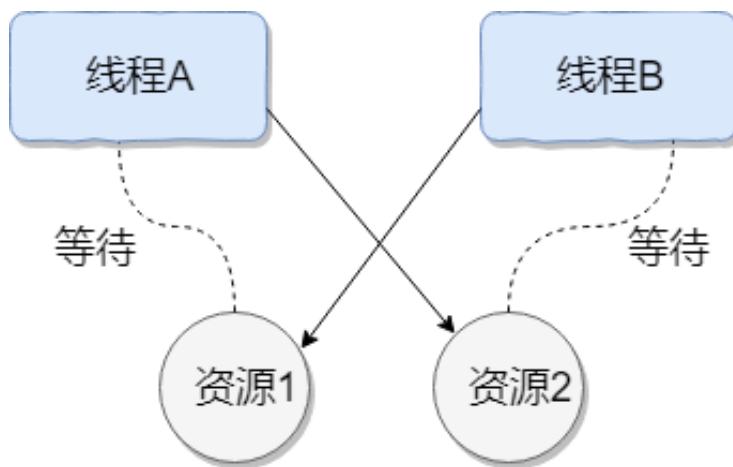
Linux 相比与其他操作系统（包括其他类 Unix 系统）有很多的优点，其中有一项就是，其上下文切换和模式切换的时间消耗非常少。

## 2.3.8. 什么是线程死锁?如何避免死锁?

### 2.3.8.1. 认识线程死锁

线程死锁描述的是这样一种情况：多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。



下面通过一个例子来说明线程死锁，代码模拟了上图的死锁的情况 (代码来源于《并发编程之美》)：

```
public class DeadLockDemo {  
    private static Object resource1 = new Object(); // 资源 1  
    private static Object resource2 = new Object(); // 资源 2  
  
    public static void main(String[] args) {  
        new Thread(() -> {  
            synchronized (resource1) {  
                System.out.println(Thread.currentThread() + " get resource1");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                System.out.println(Thread.currentThread() + " waiting get  
resource2");  
            }  
            synchronized (resource2) {  
                System.out.println(Thread.currentThread() + " get  
resource2");  
            }  
        }, "线程 1").start();  
  
        new Thread(() -> {  
            synchronized (resource2) {  
                System.out.println(Thread.currentThread() + " get resource2");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                System.out.println(Thread.currentThread() + " waiting get  
resource1");  
            }  
            synchronized (resource1) {  
                System.out.println(Thread.currentThread() + " get  
resource1");  
            }  
        }, "线程 2").start();  
    }  
}
```

## Output

```
Thread[线程 1,5,main]get resource1
Thread[线程 2,5,main]get resource2
Thread[线程 1,5,main]waiting get resource2
Thread[线程 2,5,main]waiting get resource1
```

线程 A 通过 synchronized (resource1) 获得 resource1 的监视器锁，然后通过 Thread.sleep(1000); 让线程 A 休眠 1s 为的是让线程 B 得到执行然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。上面的例子符合产生死锁的四个必要条件。

学过操作系统的朋友都知道产生死锁必须具备以下四个条件：

1. **互斥条件**: 该资源任意一个时刻只由一个线程占用。
2. **请求与保持条件**: 一个进程因请求资源而阻塞时，对已获得的资源保持不放。
3. **不剥夺条件**: 线程已获得的资源在未使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。
4. **循环等待条件**: 若干进程之间形成一种头尾相接的循环等待资源关系。

### 2.3.8.2. 如何避免线程死锁？

我上面说了产生死锁的四个必要条件，为了避免死锁，我们只要破坏产生死锁的四个条件中的其中一个就可以了。现在我们来挨个分析一下：

1. **破坏互斥条件**: 这个条件我们没有办法破坏，因为我们用锁本来就是想让他们互斥的（临界资源需要互斥访问）。
2. **破坏请求与保持条件**: 一次性申请所有的资源。
3. **破坏不剥夺条件**: 占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。
4. **破坏循环等待条件**: 靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。

我们对线程 2 的代码修改成下面这样就不会产生死锁了。

```
new Thread(() -> {
    synchronized (resource1) {
        System.out.println(Thread.currentThread() + "get resource1");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
})
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread() + "waiting get
resource2");
        synchronized (resource2) {
            System.out.println(Thread.currentThread() + "get
resource2");
        }
    }
}, "线程 2").start();
```

## Output

```
Thread[线程 1,5,main]get resource1
Thread[线程 1,5,main]waiting get resource2
Thread[线程 1,5,main]get resource2
Thread[线程 2,5,main]get resource1
Thread[线程 2,5,main]waiting get resource2
Thread[线程 2,5,main]get resource2

Process finished with exit code 0
```

我们分析一下上面的代码为什么避免了死锁的发生？

线程 1 首先获得到 resource1 的监视器锁,这时候线程 2 就获取不到了。然后线程 1 再去获取 resource2 的监视器锁，可以获取到。然后线程 1 释放了对 resource1、resource2 的监视器锁的占用，线程 2 获取到就可以执行了。这样就破坏了破坏循环等待条件，因此避免了死锁。

## 2.3.9. 说说 sleep() 方法和 wait() 方法区别和共同点？

- 两者最主要的区别在于： sleep() 方法没有释放锁，而 wait() 方法释放了锁。
- 两者都可以暂停线程的执行。
- wait() 通常被用于线程间交互/通信， sleep() 通常被用于暂停执行。
- wait() 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 notify() 或者 notifyAll() 方法。 sleep() 方法执行完成后，线程会自动苏醒。或者可以使用 wait(long timeout) 超时后线程会自动苏醒。

## 2.3.10. 为什么我们调用 `start()` 方法时会执行 `run()` 方法，为什么我们不能直接调用 `run()` 方法？

这是另一个非常经典的 java 多线程面试问题，而且在面试中会经常被问到。很简单，但是很多人都会答不上来！

`new` 一个 `Thread`，线程进入了新建状态。调用 `start()` 方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。`start()` 会执行线程的相应准备工作，然后自动执行 `run()` 方法的内容，这是真正的多线程工作。但是，直接执行 `run()` 方法，会把 `run()` 方法当成一个 `main` 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。

总结：调用 `start()` 方法方可启动线程并使线程进入就绪状态，直接执行 `run()` 方法的话不会以多线程的方式执行。



## 2.3.11. 说一说自己对于 `synchronized` 关键字的了解

`synchronized` 关键字解决的是多个线程之间访问资源的同步性，`synchronized` 关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

另外，在 Java 早期版本中，`synchronized` 属于 **重量级锁**，效率低下。

为什么呢？

因为监视器锁（monitor）是依赖于底层的操作系统的 `Mutex Lock` 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高。

庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对 `synchronized` 较大优化，所以现在的 `synchronized` 锁效率也优化得很不错了。JDK1.6 对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

所以，你会发现目前的话，不论是各种开源框架还是 JDK 源码都大量使用了 `synchronized` 关键字。

## 2.3.12. 说自己是怎么使用 synchronized 关键字

synchronized 关键字最主要的三种使用方式：

1.修饰实例方法：作用于当前对象实例加锁，进入同步代码前要获得 **当前对象实例的锁**

```
synchronized void method() {  
    //业务代码  
}
```

2.修饰静态方法：也就是给当前类加锁，会作用于类的所有对象实例，进入同步代码前要获得 **当前 class 的锁**。因为静态成员不属于任何一个实例对象，是类成员（*static* 表明这是该类的一个静态资源，不管 *new* 了多少个对象，只有一份）。所以，如果一个线程 A 调用一个实例对象的非静态 *synchronized* 方法，而线程 B 需要调用这个实例对象所属类的静态 *synchronized* 方法，是允许的，不会发生互斥现象，因为访问静态 *synchronized* 方法占用的锁是当前类的锁，而访问非静态 *synchronized* 方法占用的锁是当前实例对象锁。

```
synchronized void static method() {  
    //业务代码  
}
```

3.修饰代码块：指定加锁对象，对给定对象/类加锁。*synchronized(thisobject)* 表示进入同步代码库前要获得**给定对象的锁**。*synchronized(类.class)* 表示进入同步代码前要获得 **当前 class 的锁**

```
synchronized(this) {  
    //业务代码  
}
```

总结：

- *synchronized* 关键字加到 *static* 静态方法和 *synchronized(class)* 代码块上都是给 Class 上锁。
- *synchronized* 关键字加到实例方法上是给对象实例上锁。
- 尽量不要使用 *synchronized(String a)* 因为 JVM 中，字符串常量池具有缓存功能！

下面我以一个常见的面试题为例讲解一下 *synchronized* 关键字的具体使用。

面试中面试官经常会说：“单例模式了解吗？来给我手写一下！给我解释一下双重检验锁方式实现单例模式的原理呗！”

## 双重校验锁实现对象单例（线程安全）

```
public class Singleton {  
  
    private volatile static Singleton uniqueInstance;  
  
    private Singleton() {  
    }  
  
    public static Singleton getInstance() {  
        //先判断对象是否已经实例过，没有实例化过才进入加锁代码  
        if (uniqueInstance == null) {  
            //类对象加锁  
            synchronized (Singleton.class) {  
                if (uniqueInstance == null) {  
                    uniqueInstance = new Singleton();  
                }  
            }  
        }  
        return uniqueInstance;  
    }  
}
```

另外，需要注意 `uniqueInstance` 采用 `volatile` 关键字修饰也是很有必要。

`uniqueInstance` 采用 `volatile` 关键字修饰也是很有必要的，`uniqueInstance = new Singleton();` 这段代码其实是分为三步执行：

1. 为 `uniqueInstance` 分配内存空间
2. 初始化 `uniqueInstance`
3. 将 `uniqueInstance` 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1->3->2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 `getInstance()` 后发现 `uniqueInstance` 不为空，因此返回 `uniqueInstance`，但此时 `uniqueInstance` 还未被初始化。

使用 `volatile` 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。

## 2.3.13. 构造方法可以使用 `synchronized` 关键字修饰么？

先说结论：构造方法不能使用 `synchronized` 关键字修饰。

构造方法本身就属于线程安全的，不存在同步的构造方法一说。

## 2.3.14. 讲一下 `synchronized` 关键字的底层原理

`synchronized` 关键字底层原理属于 JVM 层面。

### 2.3.14.1. `synchronized` 同步语句块的情况

```
public class SynchronizedDemo {  
    public void method() {  
        synchronized (this) {  
            System.out.println("synchronized 代码块");  
        }  
    }  
}
```

通过 JDK 自带的 `javap` 命令查看 `SynchronizedDemo` 类的相关字节码信息：首先切换到类的对应目录执行 `javac SynchronizedDemo.java` 命令生成编译后的 `.class` 文件，然后执行 `javap -c -s -v -l SynchronizedDemo.class`。

```

public void method(){
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
        stack=2, locals=3, args_size=1
            0: aload_0
            1: dup
            2: astore_1
            3: monitorenter           // Field java/lang/System.out:Ljava/io/PrintStream;
            4: getstatic   #2          // String Method 1 start
            7: ldc         #3          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
            9: invokevirtual #4
            12: aload_1
            13: monitorexit
            14: goto       22
            17: astore_2
            18: aload_1
            19: monitorexit
            20: aload_2
            21: athrow
            22: return
    Exception table:
        from   to target type
        4      14    17  any
        17     20    17  any
    LineNumberTable:
        line 5: 0
        line 6: 4
        line 7: 12
        line 8: 22
    StackMapTable: number_of_entries = 2
        frame_type = 255 /* full_frame */
        offset_delta = 17
        locals = [ class test/SynchronizedDemo, class java/lang/Object ]
        stack = [ class java/lang/Throwable ]
        frame_type = 250 /* chop */
        offset_delta = 4
}
SourceFile: "SynchronizedDemo.java"

```

从上面我们可以看出：

**synchronized** 同步语句块的实现使用的是 `monitorenter` 和 `monitorexit` 指令，其中 `monitorenter` 指令指向同步代码块的开始位置，`monitorexit` 指令则指明同步代码块的结束位置。

当执行 `monitorenter` 指令时，线程试图获取锁也就是获取 **对象监视器** `monitor` 的持有权。

在 Java 虚拟机(HotSpot)中，Monitor 是基于 C++实现的，由[ObjectMonitor](#)实现的。每个对象中都内置了一个 ObjectMonitor 对象。

另外，`wait/notify` 等方法也依赖于 `monitor` 对象，这就是为什么只有在同步的块或者方法中才能调用 `wait/notify` 等方法，否则会抛出 `java.lang.IllegalMonitorStateException` 的异常的原因。

在执行 `monitorenter` 时，会尝试获取对象的锁，如果锁的计数器为 0 则表示锁可以被获取，获取后将锁计数器设为 1 也就是加 1。

在执行 `monitorexit` 指令后，将锁计数器设为 0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

### 2.3.14.2. synchronized 修饰方法的情况

```
public class SynchronizedDemo2 {  
    public synchronized void method() {  
        System.out.println("synchronized 方法");  
    }  
}
```

```
{  
    public test.SynchronizedDemo2();  
    descriptor: ()V  
    flags: ACC_PUBLIC  
    Code:  
        stack=1, locals=1, args_size=1  
        0: aload_0  
        1: invokespecial #1                  // Method java/lang/Object."<init>":()V  
        4: return  
    LineNumberTable:  
        line 3: 0  
  
    public synchronized void method();  
    descriptor: ()V  
    flags: ACC_PUBLIC, ACC_SYNCHRONIZED  
    Code:  
        stack=2, locals=1, args_size=1  
        0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;  
        3: ldc          #3                  // String synchronized 鐵規碼  
        5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V  
        8: return  
    LineNumberTable:  
        line 5: 0  
        line 6: 8  
}  
SourceFile: "SynchronizedDemo2.java"
```

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是 ACC\_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法。JVM 通过该 ACC\_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

### 2.3.14.3. 总结

synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置， monitorexit 指令则指明同步代码块的结束位置。

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是 ACC\_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法。

不过两者的本质都是对对象监视器 monitor 的获取。

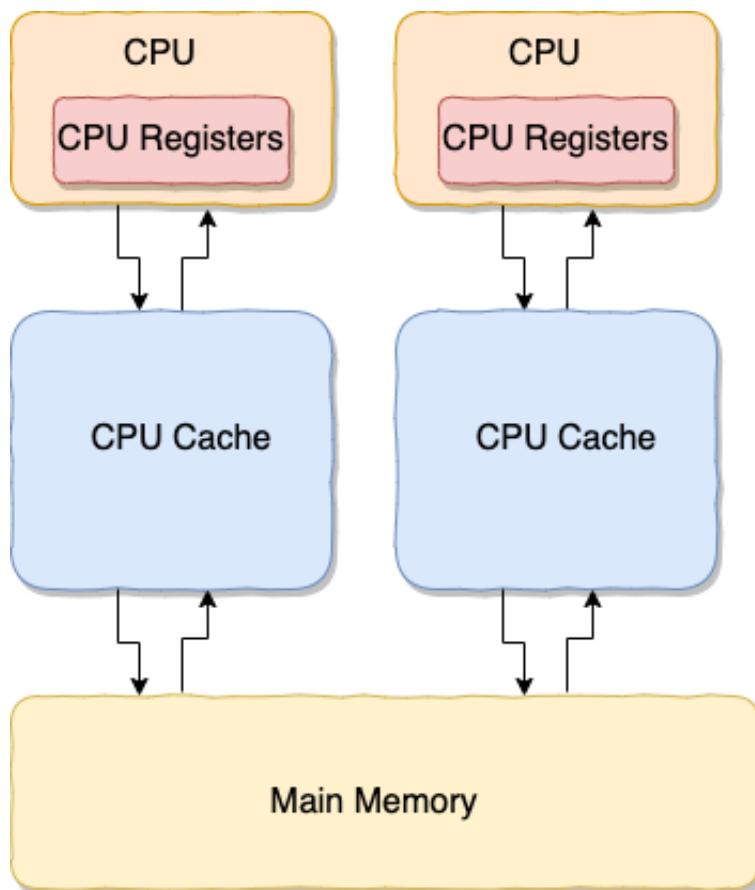
## 2.3.15. 为什么要弄一个 CPU 高速缓存呢？

类比我们开发网站后台系统使用的缓存（比如 Redis）是为了解决程序处理速度和访问常规关系型数据库速度不对等的问题。CPU 缓存则是为了解决 CPU 处理速度和内存处理速度不对等的问题。

我们甚至可以把 内存可以看作外存的高速缓存，程序运行的时候我们把外存的数据复制到内存，由于内存的处理速度远远高于外存，这样提高了处理速度。

总结：CPU Cache 缓存的是内存数据用于解决 CPU 处理速度和内存不匹配的问题，内存缓存的是硬盘数据用于解决硬盘访问速度过慢的问题。

为了更好地理解，我画了一个简单的 CPU Cache 示意图如下（实际上，现代的 CPU Cache 通常分为三层，分别叫 L1,L2,L3 Cache）：



作者：Guide哥  
公众号&Github：JavaGuide

CPU Cache 的工作方式：

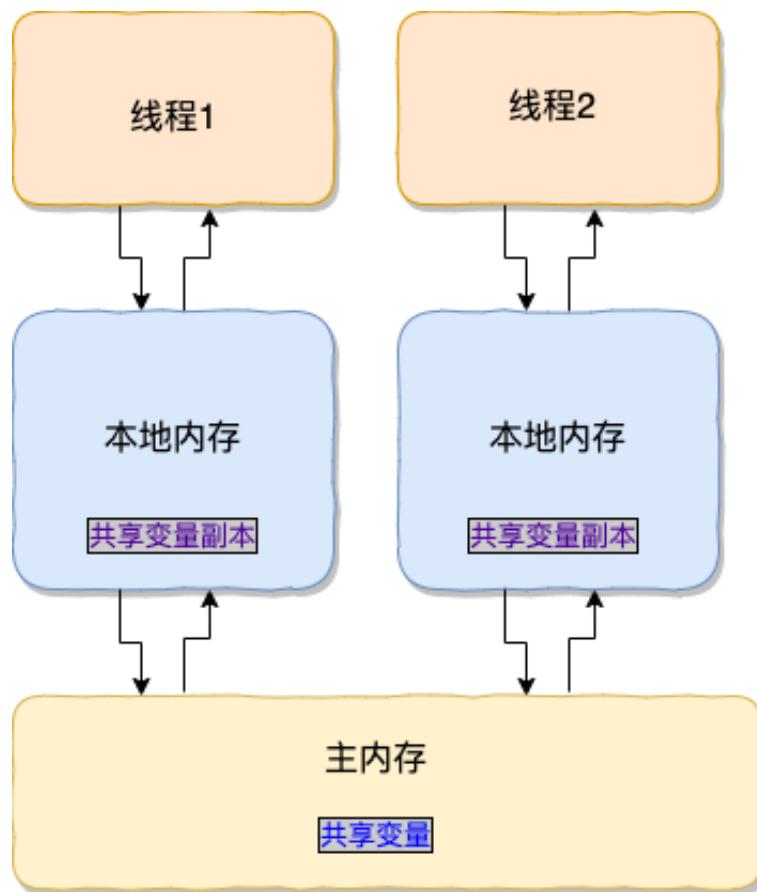
先复制一份数据到 CPU Cache 中，当 CPU 需要用到的时候就可以直接从 CPU Cache 中读取数据，当运算完成后，再将运算得到的数据写回 Main Memory 中。但是，这样存在 内存缓存不一致性的问题！比如我执行一个 `i++` 操作的话，如果两个线程同时执行的话，假设两个线程从 CPU Cache 中读取的 `i=1`，两个线程做了 `i++` 运算完之后再写回 Main Memory 之后 `i=2`，而正确结果

应该是  $i=3$ 。

CPU 为了解决内存缓存不一致性问题可以通过制定缓存一致协议或者其他手段来解决。

### 2.3.16. 讲一下 JMM(Java 内存模型)

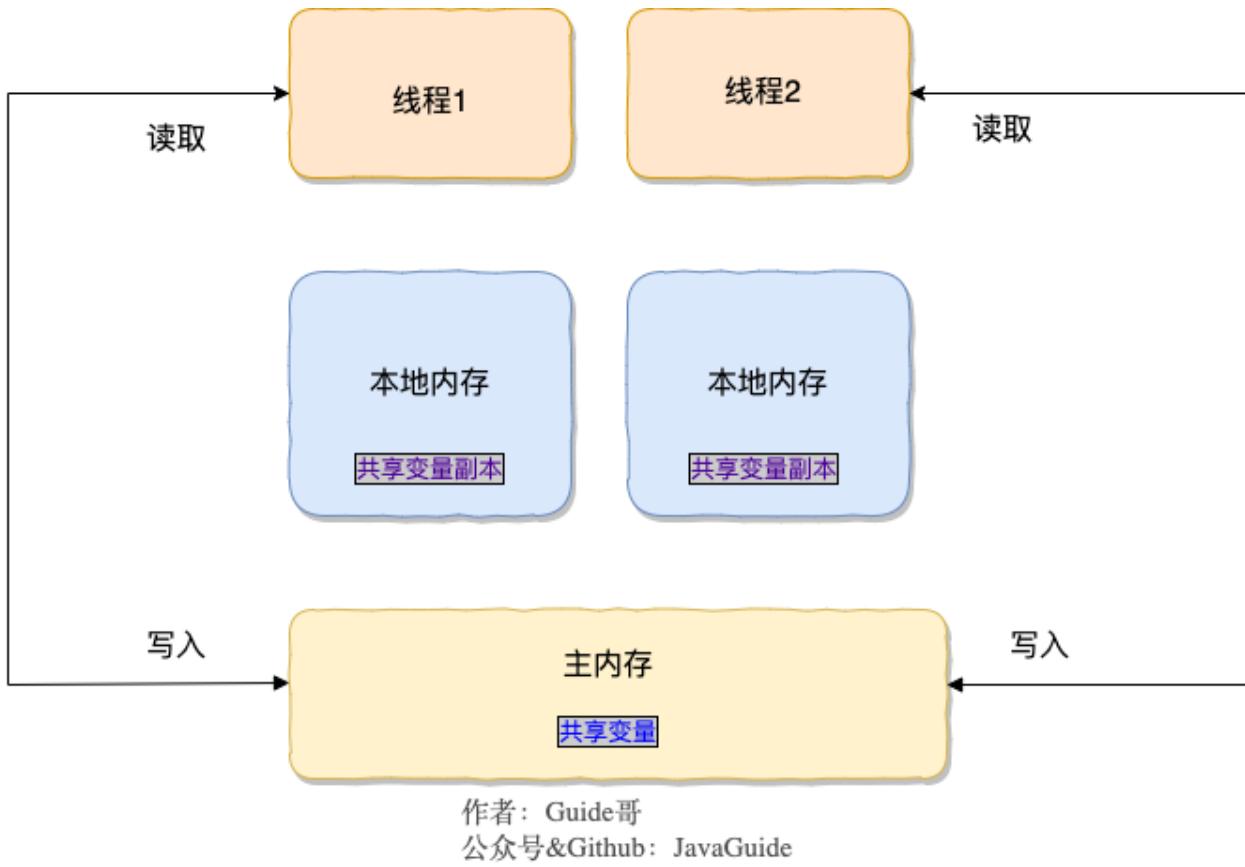
在 JDK1.2 之前, Java 的内存模型实现总是从主存 (即共享内存) 读取变量, 是不需要进行特别的注意的。而在当前的 Java 内存模型下, 线程可以把变量保存本地内存 (比如机器的寄存器) 中, 而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值, 而另外一个线程还继续使用它在寄存器中的变量值的拷贝, 造成数据的不一致。



作者: Guide哥  
公众号&Github: JavaGuide

要解决这个问题, 就需要把变量声明为 `volatile`, 这就指示 JVM, 这个变量是共享且不稳定的, 每次使用它都到主存中进行读取。

所以, `volatile` 关键字除了防止 JVM 的指令重排, 还有一个重要的作用就是保证变量的可见性。



### 2.3.17. 说说 `synchronized` 关键字和 `volatile` 关键字的区别

`synchronized` 关键字和 `volatile` 关键字是两个互补的存在，而不是对立的存在！

- `volatile` 关键字是线程同步的轻量级实现，所以 `volatile` 性能肯定比 `synchronized` 关键字要好。但是 `volatile` 关键字只能用于变量而 `synchronized` 关键字可以修饰方法以及代码块。
- `volatile` 关键字能保证数据的可见性，但不能保证数据的原子性。`synchronized` 关键字两者都能保证。
- `volatile` 关键字主要用于解决变量在多个线程之间的可见性，而 `synchronized` 关键字解决的是多个线程之间访问资源的同步性。

### 2.3.18. `ThreadLocal` 了解么？

通常情况下，我们创建的变量是可以被任何一个线程访问并修改的。如果想实现每一个线程都有自己的专属本地变量该如何解决呢？JDK 中提供的 `ThreadLocal` 类正是为了解决这样的问题。

`ThreadLocal` 类主要解决的就是让每个线程绑定自己的值，可以将 `ThreadLocal` 类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。

如果你创建了一个 `ThreadLocal` 变量，那么访问这个变量的每个线程都会有这个变量的本地副本，这也是 `ThreadLocal` 变量名的由来。他们可以使用 `get()` 和 `set()` 方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。

再举个简单的例子：

比如有两个人去宝屋收集宝物，这两个共用一个袋子的话肯定会产生争执，但是给他们两个人每个人分配一个袋子的话就不会出现这样的问题。如果把这两个人比作线程的话，那么 ThreadLocal 就是用来避免这两个线程竞争的。

### 2.3.19. ThreadLocal 原理讲一下

从 Thread 类源代码入手。

```
public class Thread implements Runnable {  
    ....  
    //与此线程有关的ThreadLocal值。由ThreadLocal类维护  
    ThreadLocal.ThreadLocalMap threadLocals = null;  
  
    //与此线程有关的InheritableThreadLocal值。由InheritableThreadLocal类维护  
    ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;  
    ....  
}
```

从上面 Thread 类源代码可以看出 Thread 类中有一个 threadLocals 和一个 inheritableThreadLocals 变量，它们都是 ThreadLocalMap 类型的变量，我们可以把 ThreadLocalMap 理解为 ThreadLocal 类实现的定制化的 HashMap。默认情况下这两个变量都是 null，只有当前线程调用 ThreadLocal 类的 set 或 get 方法时才创建它们，实际上调用这两个方法的时候，我们调用的是 ThreadLocalMap 类对应的 get()、set() 方法。

ThreadLocal 类的 set() 方法

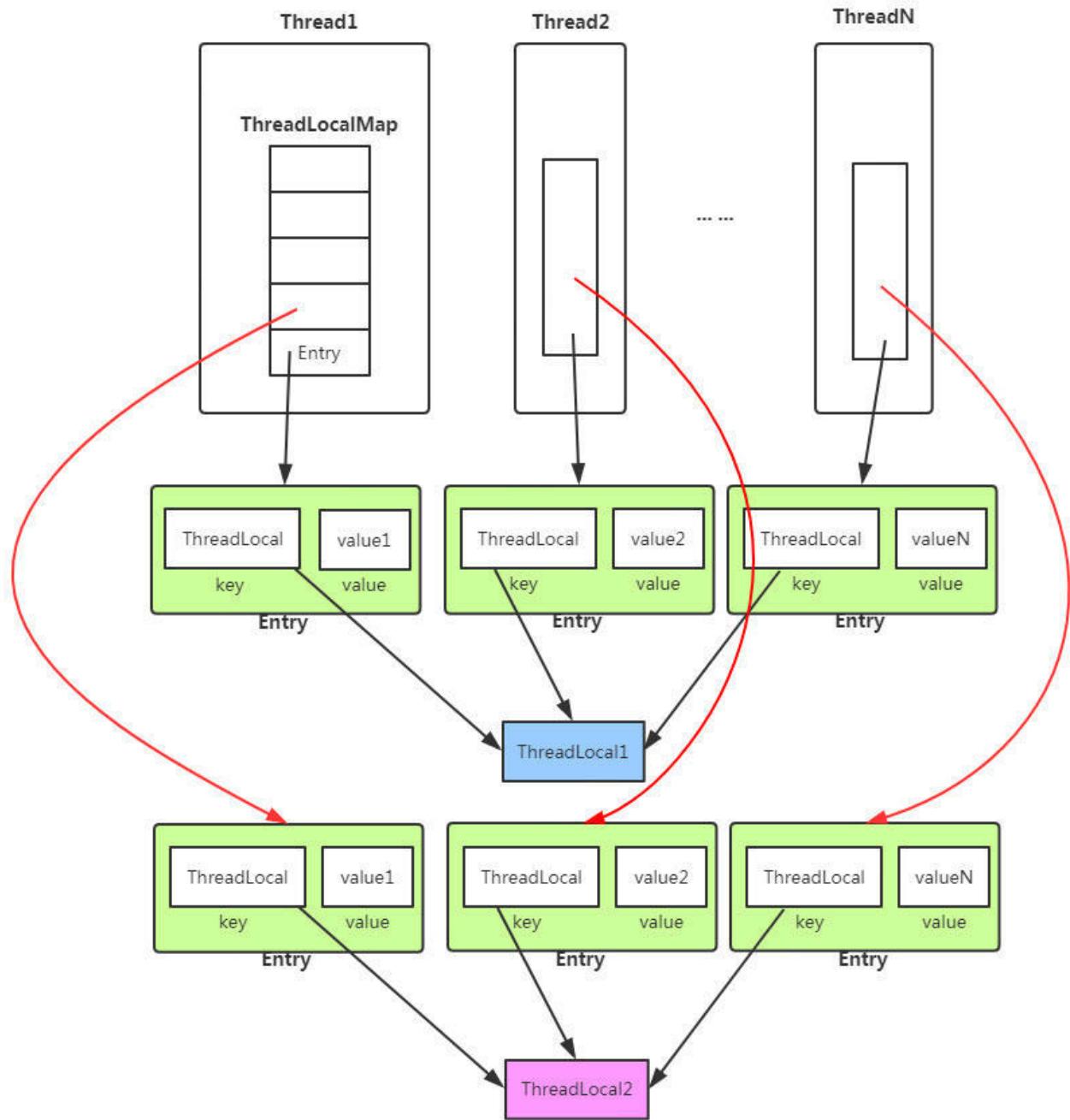
```
public void set(T value) {  
    Thread t = Thread.currentThread();  
    ThreadLocalMap map = getMap(t);  
    if (map != null)  
        map.set(this, value);  
    else  
        createMap(t, value);  
}  
ThreadLocalMap getMap(Thread t) {  
    return t.threadLocals;  
}
```

通过上面这些内容，我们足以通过猜测得出结论：最终的变量是放在了当前线程的 `ThreadLocalMap` 中，并不是存在 `ThreadLocal` 上，`ThreadLocal` 可以理解为只是 `ThreadLocalMap` 的封装，传递了变量值。 `ThrealLocal` 类中可以通过 `Thread.currentThread()` 获取到当前线程对象后，直接通过 `getMap(Thread t)` 可以访问到该线程的 `ThreadLocalMap` 对象。

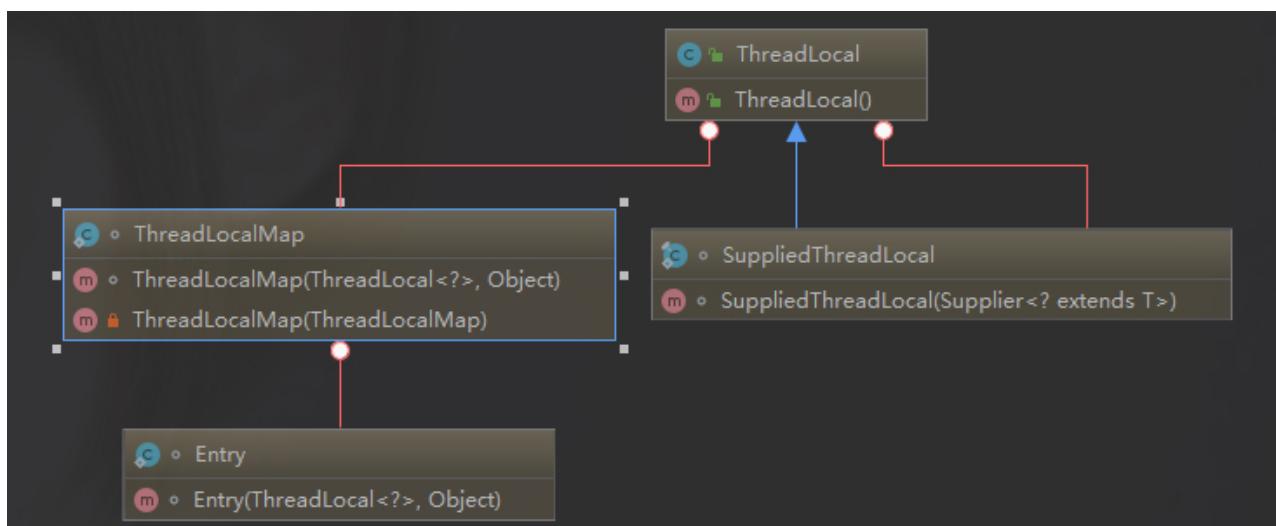
`ThreadLocal` 内部维护的是一个类似 `Map` 的 `ThreadLocalMap` 数据结构，`key` 为当前对象的 `Thread` 对象，值为 `Object` 对象。

```
ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {  
    .....  
}
```

比如我们在同一个线程中声明了两个 `ThreadLocal` 对象的话，会使用 `Thread` 内部都是使用仅有那个 `ThreadLocalMap` 存放数据的，`ThreadLocalMap` 的 `key` 就是 `ThreadLocal` 对象，`value` 就是 `ThreadLocal` 对象调用 `set` 方法设置的值。



ThreadLocalMap 是 ThreadLocal 的静态内部类。



## 2.3.20. ThreadLocal 内存泄露问题了解不？

ThreadLocalMap 中使用的 key 为 ThreadLocal 的弱引用,而 value 是强引用。所以, 如果 ThreadLocal 没有被外部强引用的情况下, 在垃圾回收的时候, key 会被清理掉, 而 value 不会。这样一来, ThreadLocalMap 中就会出现 key 为 null 的 Entry。假如我们不做任何措施的话, value 永远无法被 GC 回收, 这个时候就可能会产生内存泄露。ThreadLocalMap 实现中已经考虑了这种情况, 在调用 set()、get()、remove() 方法的时候, 会清理掉 key 为 null 的记录。使用完 ThreadLocal 方法后 最好手动调用 remove() 方法

```
static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
        super(k);
        value = v;
    }
}
```

弱引用介绍：

如果一个对象只具有弱引用, 那就类似于可有可无的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中, 一旦发现了只具有弱引用的对象, 不管当前内存空间足够与否, 都会回收它的内存。不过, 由于垃圾回收器是一个优先级很低的线程, 因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列 (ReferenceQueue) 联合使用, 如果弱引用所引用的对象被垃圾回收, Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

## 2.3.21. 线程池

### 2.3.21.1. 为什么要用线程池？

池化技术相比大家已经屡见不鲜了, 线程池、数据库连接池、Http 连接池等等都是对这个思想的应用。池化技术的思想主要是为了减少每次获取资源的消耗, 提高对资源的利用率。

线程池提供了一种限制和管理资源（包括执行一个任务）。每个线程池还维护一些基本统计信息，例如已完成任务的数量。

这里借用《Java 并发编程的艺术》提到的来说一下使用线程池的好处：

- **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。

### 2.3.21.2. 实现 Runnable 接口和 Callable 接口的区别

Runnable 自 Java 1.0 以来一直存在，但 Callable 仅在 Java 1.5 中引入，目的就是为了来处理 Runnable 不支持的用例。**Runnable** 接口不会返回结果或抛出检查异常，但是 **Callable** 接口可以。所以，如果任务不需要返回结果或抛出异常推荐使用 **Runnable** 接口，这样代码看起来会更加简洁。

工具类 `Executors` 可以实现 `Runnable` 对象和 `Callable` 对象之间的相互转换。

(`Executors.callable (Runnable task)` 或 `Executors.callable (Runnable task, Object result)`)。

Runnable.java

```
@FunctionalInterface  
public interface Runnable {  
    /**  
     * 被线程执行，没有返回值也无法抛出异常  
     */  
    public abstract void run();  
}
```

Callable.java

```
@FunctionalInterface
public interface Callable<V> {
    /**
     * 计算结果，或在无法这样做时抛出异常。
     * @return 计算得出的结果
     * @throws 如果无法计算结果，则抛出异常
     */
    V call() throws Exception;
}
```

### 2.3.21.3. 执行 execute()方法和 submit()方法的区别是什么呢？

1. execute() 方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；
2. submit() 方法用于提交需要返回值的任务。线程池会返回一个 Future 类型的对象，通过这个 Future 对象可以判断任务是否执行成功，并且可以通过 Future 的 get() 方法来获取返回值，get() 方法会阻塞当前线程直到任务完成，而使用 get (long timeout, TimeUnit unit) 方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

我们以 AbstractExecutorService 接口中的一个 submit 方法为例子来看看源代码：

```
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<Void> ftask = newTaskFor(task, null);
    execute(ftask);
    return ftask;
}
```

上面方法调用的 newTaskFor 方法返回了一个 FutureTask 对象。

```
protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {
    return new FutureTask<T>(runnable, value);
}
```

我们再来看看 execute() 方法：

```
public void execute(Runnable command) {  
    ...  
}
```

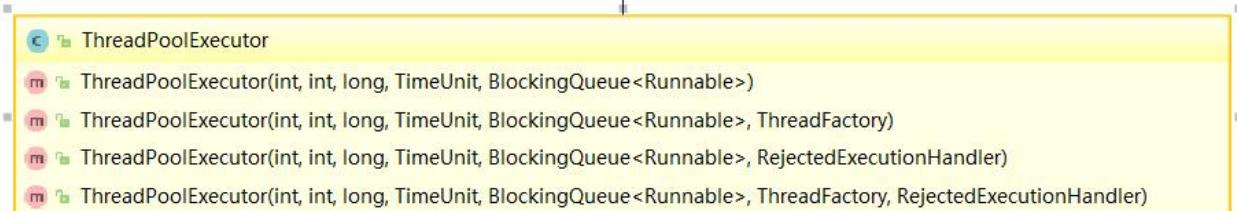
#### 2.3.21.4. 如何创建线程池

《阿里巴巴 Java 开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

Executors 返回线程池对象的弊端如下：

- **FixedThreadPool 和 SingleThreadExecutor**：允许请求的队列长度为 Integer.MAX\_VALUE，可能堆积大量的请求，从而导致 OOM。
- **CachedThreadPool 和 ScheduledThreadPool**：允许创建的线程数量为 Integer.MAX\_VALUE，可能会创建大量线程，从而导致 OOM。

#### 方式一：通过构造方法实现



#### 方式二：通过 Executor 框架的工具类 Executors 来实现

我们可以创建三种类型的 ThreadPoolExecutor：

- **FixedThreadPool**：该方法返回一个固定线程数量的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。
- **SingleThreadExecutor**：方法返回一个只有一个线程的线程池。若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。
- **CachedThreadPool**：该方法返回一个可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。

对应 Executors 工具类中的方法如图所示：

| Executors                              |                 |
|--|-----------------|
| newFixedThreadPool(int)                | ExecutorService |
| newWorkStealingPool(int)               | ExecutorService |
| newWorkStealingPool()                  | ExecutorService |
| newFixedThreadPool(int, ThreadFactory) | ExecutorService |
| newSingleThreadExecutor()              | ExecutorService |
| newSingleThreadExecutor(ThreadFactory) | ExecutorService |
| newCachedThreadPool()                  | ExecutorService |
| newCachedThreadPool(ThreadFactory)     | ExecutorService |

### 2.3.21.5. ThreadPoolExecutor 类分析

ThreadPoolExecutor 类中提供的四个构造方法。我们来看最长的那个，其余三个都是在这个构造方法的基础上产生（其他几个构造方法说白点都是给定某些默认参数的构造方法比如默认制定拒绝策略是什么），这里就不贴代码讲了，比较简单。

```
/*
 * 用给定的初始参数创建一个新的ThreadPoolExecutor。
 */
public ThreadPoolExecutor(int corePoolSize,
                           int maximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue<Runnable> workQueue,
                           ThreadFactory threadFactory,
                           RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

---

下面这些对创建 非常重要，在后面使用线程池的过程中你一定会用到！所以，务必拿着小本本记清楚。

### 2.3.21.5.1. ThreadPoolExecutor 构造函数重要参数分析

ThreadPoolExecutor 3 个最重要的参数：

- `corePoolSize` : 核心线程数线程数定义了最小可以同时运行的线程数量。
- `maximumPoolSize` : 当队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数。
- `workQueue` : 当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，新任务就会被存放在队列中。

ThreadPoolExecutor 其他常见参数：

1. `keepAliveTime` : 当线程池中的线程数量大于 `corePoolSize` 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 `keepAliveTime` 才会被回收销毁；
2. `unit` : `keepAliveTime` 参数的时间单位。
3. `threadFactory` : `executor` 创建新线程的时候会用到。
4. `handler` : 饱和策略。关于饱和策略下面单独介绍一下。

### 2.3.21.5.2. ThreadPoolExecutor 饱和策略

ThreadPoolExecutor 饱和策略定义：

如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任时， ThreadPoolTaskExecutor 定义一些策略：

- `ThreadPoolExecutor.AbortPolicy` : 抛出 `RejectedExecutionException` 来拒绝新任务的处理。
- `ThreadPoolExecutor.CallerRunsPolicy` : 调用执行自己的线程运行任务。您不会任务请求。但是这种策略会降低对于新任务提交速度，影响程序的整体性能。另外，这个策略喜欢增加队列容量。如果您的应用程序可以承受此延迟并且你不能任务丢弃任何一个任务请求的话，你可以选择这个策略。
- `ThreadPoolExecutor.DiscardPolicy` : 不处理新任务，直接丢弃掉。
- `ThreadPoolExecutor.DiscardOldestPolicy` : 此策略将丢弃最早的未处理的任务请求。

举个例子： Spring 通过 `ThreadPoolTaskExecutor` 或者我们直接通过 `ThreadPoolExecutor` 的构造函数创建线程池的时候，当我们不指定 `RejectedExecutionHandler` 饱和策略的话来配置线程池的时候默认使用的是 `ThreadPoolExecutor.AbortPolicy` 。在默认情况下， `ThreadPoolExecutor` 将抛出 `RejectedExecutionException` 来拒绝新来的任务，这代表你将丢失对这个任务的处理。对于可伸缩的应用程序，建议使用 `ThreadPoolExecutor.CallerRunsPolicy` 。当最大池被填满时，此策略为我

们提供可伸缩队列。（这个直接查看 `ThreadPoolExecutor` 的构造函数源码就可以看出，比较简单的原因，这里就不贴代码了）

### 2.3.21.6. 线程池原理分析

承接 4.6 节，我们通过代码输出结果可以看出：线程池每次会同时执行 5 个任务，这 5 个任务执行完之后，剩余的 5 个任务才会被执行。大家可以先通过上面讲解的内容，分析一下到底是咋回事？（自己独立思考一会儿）

现在，我们就分析上面的输出内容来简单分析一下线程池原理。

为了搞懂线程池的原理，我们需要首先分析一下 `execute` 方法。在 4.6 节中的 Demo 中我们使用 `executor.execute(worker)` 来提交一个任务到线程池中去，这个方法非常重要，下面我们来看看它的源码：

```
// 存放线程池的运行状态 (runState) 和线程池内有效线程的数量 (workerCount)
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

private static int workerCountOf(int c) {
    return c & CAPACITY;
}

private final BlockingQueue<Runnable> workQueue;

public void execute(Runnable command) {
    // 如果任务为null，则抛出异常。
    if (command == null)
        throw new NullPointerException();
    // ctl 中保存的线程池当前的一些状态信息
    int c = ctl.get();

    // 下面会涉及到 3 步 操作
    // 1.首先判断当前线程池中之行的任务数量是否小于 corePoolSize
    // 如果小于的话，通过addWorker(command, true)新建一个线程，并将任务(command)
添加到该线程中；然后，启动该线程从而执行任务。
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    // 2.如果当前之行的任务数量大于等于 corePoolSize 的时候就会走到这里
```

```

// 通过 isRunning 方法判断线程池状态，线程池处于 RUNNING 状态才会被并且队列可以
加入任务，该任务才会被加入进去

if (isRunning(c) && workQueue.offer(command)) {

    int recheck = ctl.get();

    // 再次获取线程池状态，如果线程池状态不是 RUNNING 状态就需要从任务队列中移除
    任务，并尝试判断线程是否全部执行完毕。同时执行拒绝策略。

    if (!isRunning(recheck) && remove(command))
        reject(command);

    // 如果当前线程池为空就新创建一个线程并执行。
    else if (workerCountOf(recheck) == 0)
        addWorker(null, false);

}

// 3. 通过addWorker(command, false)新建一个线程，并将任务(command)添加到该线
程中；然后，启动该线程从而执行任务。

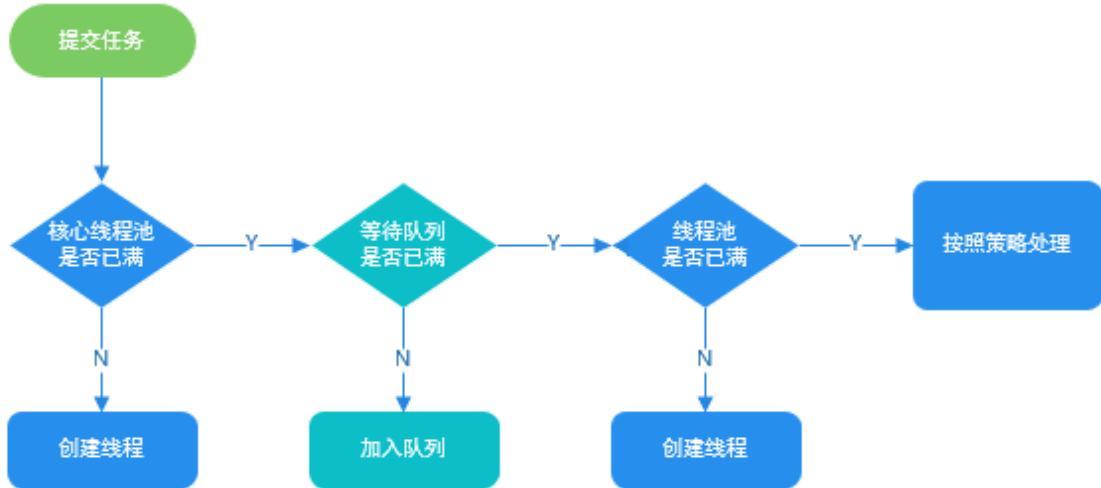
// 如果addWorker(command, false)执行失败，则通过reject()执行相应的拒绝策略的内
容。

else if (!addWorker(command, false))
    reject(command);

}

```

通过下图可以更好的对上面这 3 步做一个展示，下图是我为了省事直接从网上找到，原地址不明。



现在，让我们回到 4.6 节我们写的 Demo，现在应该是很容易就可以搞懂它的原理了呢？

没搞懂的话，也没关系，可以看看我的分析：

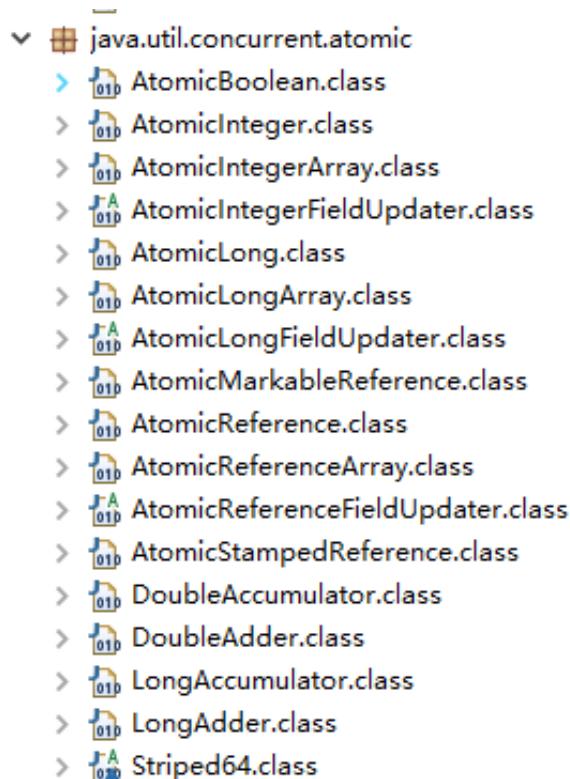
我们在代码中模拟了 10 个任务，我们配置的核心线程数为 5、等待队列容量为 100，所以每次只可能存在 5 个任务同时执行，剩下的 5 个任务会被放到等待队列中去。当前的 5 个任务之行完成后，才会之行剩下的 5 个任务。

### 2.3.22. 介绍一下 Atomic 原子类

Atomic 翻译成中文是原子的意思。在化学上，我们知道原子是构成一般物质的最小单位，在化学反应中是不可分割的。在我们这里 Atomic 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。

所以，所谓原子类说简单点就是具有原子/原子操作特征的类。

并发包 `java.util.concurrent` 的原子类都存放在 `java.util.concurrent.atomic` 下,如下图所示。



### 2.3.23. JUC 包中的原子类是哪 4 类?

基本类型

使用原子的方式更新基本类型

- `AtomicInteger` : 整形原子类
- `AtomicLong` : 长整形原子类
- `AtomicBoolean` : 布尔型原子类

## 数组类型

使用原子的方式更新数组里的某个元素

- AtomicIntegerArray : 整形数组原子类
- AtomicLongArray : 长整形数组原子类
- AtomicReferenceArray : 引用类型数组原子类

## 引用类型

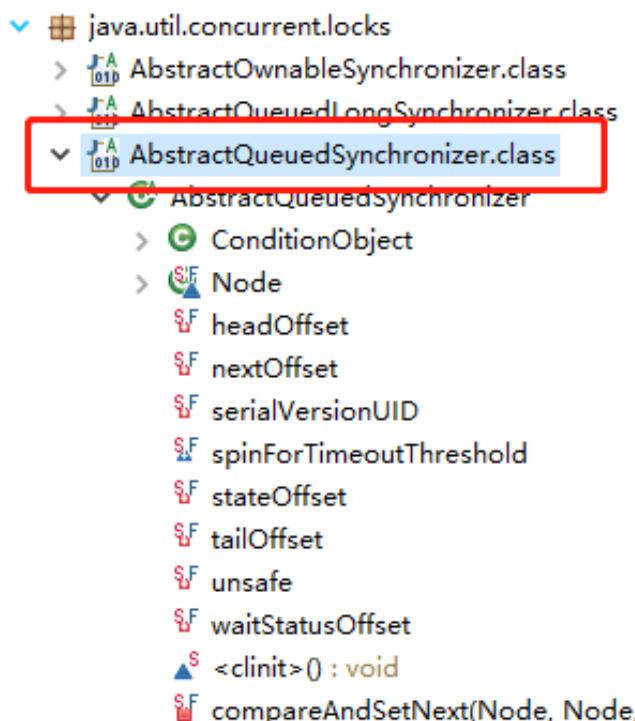
- AtomicReference : 引用类型原子类
- AtomicStampedReference : 原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。
- AtomicMarkableReference : 原子更新带有标记位的引用类型

## 对象的属性修改类型

- AtomicIntegerFieldUpdater : 原子更新整形字段的更新器
- AtomicLongFieldUpdater : 原子更新长整形字段的更新器
- AtomicReferenceFieldUpdater : 原子更新引用类型字段的更新器

## 2.3.24. AQS 了解么？

AQS 的全称为（ AbstractQueuedSynchronizer ），这个类在 java.util.concurrent.locks 包下面。



AQS 是一个用来构建锁和同步器的框架，使用 AQS 能简单且高效地构造出应用广泛的大量的同步器，比如我们提到的 `ReentrantLock`，`Semaphore`，其他的诸如 `ReentrantReadWriteLock`，`SynchronousQueue`，`FutureTask` 等等皆是基于 AQS 的。当然，我们自己也能利用 AQS 非常轻松容易地构造出符合我们自己需求的同步器。

### 2.3.25. AQS 原理了解么？

AQS 原理这部分参考了部分博客，在 5.2 节末尾放了链接。

在面试中被问到并发知识的时候，大多都会被问到“请你说一下自己对于 AQS 原理的理解”。下面给大家一个示例供大家参加，面试不是背题，大家一定要加入自己的思想，即使加入不了自己的思想也要保证自己能够通俗的讲出来而不是背出来。

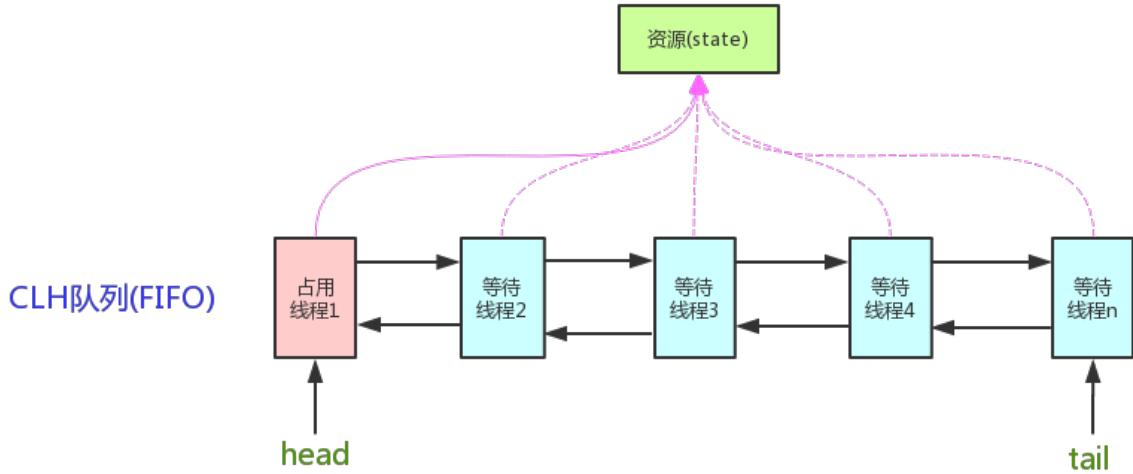
下面大部分内容其实在 AQS 类注释上已经给出了，不过是英语看着比较吃力一点，感兴趣的话可以看看源码。

#### 2.3.25.1. AQS 原理概览

AQS 核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制 AQS 是用 CLH 队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

CLH(Craig,Landin,and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS 是将每条请求共享资源的线程封装成一个 CLH 锁队列的一个结点（Node）来实现锁的分配。

看个 AQS(AbstractQueuedSynchronizer)原理图：



AQS 使用一个 int 成员变量来表示同步状态，通过内置的 FIFO 队列来完成获取资源线程的排队工作。AQS 使用 CAS 对该同步状态进行原子操作实现对其值的修改。

```
private volatile int state; //共享变量，使用volatile修饰保证线程可见性
```

状态信息通过 protected 类型的 getState, setState, compareAndSetState 进行操作

```
//返回同步状态的当前值
protected final int getState() {
    return state;
}

// 设置同步状态的值
protected final void setState(int newState) {
    state = newState;
}

//原子地（CAS操作）将同步状态值设置为给定值update如果当前同步状态的值等于expect（期望值）
protected final boolean compareAndSetState(int expect, int update) {
    return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
}
```

## 2.3.25.2. AQS 对资源的共享方式

### AQS 定义两种资源共享方式

- **Exclusive** (独占) : 只有一个线程能执行, 如 `ReentrantLock`。又可分为公平锁和非公平锁:
  - 公平锁: 按照线程在队列中的排队顺序, 先到者先拿到锁
  - 非公平锁: 当线程要获取锁时, 无视队列顺序直接去抢锁, 谁抢到就是谁的
- **Share** (共享) : 多个线程可同时执行, 如 `CountDownLatch`、`Semaphore`、`CountDownLatch`、`CyclicBarrier`、`ReadWriteLock` 我们都会在后面讲到。

`ReentrantReadWriteLock` 可以看成是组合式, 因为 `ReentrantReadWriteLock` 也就是读写锁允许多个线程同时对某一资源进行读。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 `state` 的获取与释放方式即可, 至于具体线程等待队列的维护 (如获取资源失败入队/唤醒出队等), AQS 已经在顶层实现好了。

## 2.3.25.3. AQS 底层使用了模板方法模式

同步器的设计是基于模板方法模式的, 如果需要自定义同步器一般的方式是这样 (模板方法模式很经典的一个应用) :

1. 使用者继承 `AbstractQueuedSynchronizer` 并重写指定的方法。(这些重写方法很简单, 无非是对于共享资源 `state` 的获取和释放)
2. 将 AQS 组合在自定义同步组件的实现中, 并调用其模板方法, 而这些模板方法会调用使用者重写的方法。

这和我们以往通过实现接口的方式有很大区别, 这是模板方法模式很经典的一个运用。

**AQS 使用了模板方法模式, 自定义同步器时需要重写下面几个 AQS 提供的模板方法:**

```
isHeldExclusively() //该线程是否正在独占资源。只有用到condition才需要去实现它。  
tryAcquire(int) //独占方式。尝试获取资源, 成功则返回true, 失败则返回false。  
tryRelease(int) //独占方式。尝试释放资源, 成功则返回true, 失败则返回false。  
tryAcquireShared(int) //共享方式。尝试获取资源。负数表示失败; 0表示成功, 但没有剩余可用资源; 正数表示成功, 且有剩余资源。  
tryReleaseShared(int) //共享方式。尝试释放资源, 成功则返回true, 失败则返回false。
```

默认情况下，每个方法都抛出 `UnsupportedOperationException`。这些方法的实现必须是内部线程安全的，并且通常应该简短而不是阻塞。AQS 类中的其他方法都是 `final`，所以无法被其他类使用，只有这几个方法可以被其他类使用。

以 `ReentrantLock` 为例，`state` 初始化为 0，表示未锁定状态。A 线程 `lock()` 时，会调用 `tryAcquire()` 独占该锁并将 `state+1`。此后，其他线程再 `tryAcquire()` 时就会失败，直到 A 线程 `unlock()` 到 `state=0`（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A 线程自己是可以重复获取此锁的（`state` 会累加），这就是可重入的概念。但要注意，获取多少次就要释放多少次，这样才能保证 `state` 是能回到零态的。

再以 `CountDownLatch` 以例，任务分为 N 个子线程去执行，`state` 也初始化为 N（注意 N 要与线程个数一致）。这 N 个子线程是并行执行的，每个子线程执行完后 `countDown()` 一次，`state` 会 CAS(Compare and Swap) 减 1。等到所有子线程都执行完后（即 `state=0`），会 `unpark()` 主调用线程，然后主调用线程就会从 `await()` 函数返回，继续后余动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现 `tryAcquire-tryRelease`、`tryAcquireShared-tryReleaseShared` 中的一种即可。但 AQS 也支持自定义同步器同时实现独占和共享两种方式，如 `ReentrantReadWriteLock`。

推荐两篇 AQS 原理和相关源码分析的文章：

- <http://www.cnblogs.com/waterystone/p/4920797.html>
- <https://www.cnblogs.com/chengxiao/archive/2017/07/24/7141160.html>

## 2.3.26. AQS 组件总结

- **Semaphore (信号量)-允许多个线程同时访问：** `synchronized` 和 `ReentrantLock` 都是一次只允许一个线程访问某个资源，`Semaphore (信号量)` 可以指定多个线程同时访问某个资源。
- **CountDownLatch (倒计时器)：** `CountDownLatch` 是一个同步工具类，用来协调多个线程之间的同步。这个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。
- **CyclicBarrier (循环栅栏)：** `CyclicBarrier` 和 `CountDownLatch` 非常类似，它也可以实现线程间的技术等待，但是它的功能比 `CountDownLatch` 更加复杂和强大。主要应用场景和 `CountDownLatch` 类似。`CyclicBarrier` 的字面意思是可循环使用（`Cyclic`）的屏障（`Barrier`）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。`CyclicBarrier` 默认的构造方法是 `CyclicBarrier(int parties)`，其参数表示屏障拦截的线程数量，每个线程调用 `await()` 方法告诉 `CyclicBarrier` 我已经到达了屏障，然后当前线程被阻塞。

## 2.3.27. 用过 CountDownLatch 么？什么场景下用的？

CountDownLatch 的作用就是 允许 count 个线程阻塞在一个地方，直至所有线程的任务都执行完毕。之前在项目中，有一个使用多线程读取多个文件处理的场景，我用到了 CountDownLatch 。具体场景是下面这样的：

我们要读取处理 6 个文件，这 6 个任务都是没有执行顺序依赖的任务，但是我们需要返回给用户的时候将这几个文件的处理的结果进行统计整理。

为此我们定义了一个线程池和 count 为 6 的 CountDownLatch 对象。使用线程池处理读取任务，每一个线程处理完之后就将 count-1，调用 CountDownLatch 对象的 await() 方法，直到所有文件读取完之后，才会接着执行后面的逻辑。

伪代码是下面这样的：

```
public class CountDownLatchExample1 {  
    // 处理文件的数量  
    private static final int threadCount = 6;  
  
    public static void main(String[] args) throws InterruptedException {  
        // 创建一个具有固定线程数量的线程池对象（推荐使用构造方法创建）  
        ExecutorService threadPool = Executors.newFixedThreadPool(10);  
        final CountDownLatch countDownLatch = new CountDownLatch(threadCount);  
        for (int i = 0; i < threadCount; i++) {  
            final int threadnum = i;  
            threadPool.execute(() -> {  
                try {  
                    // 处理文件的业务操作  
                    .....  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                } finally {  
                    // 表示一个文件已经被完成  
                    countDownLatch.countDown();  
                }  
            });  
        }  
        countDownLatch.await();  
        threadPool.shutdown();  
        System.out.println("finish");  
    }  
}
```

```
}
```

有没有可以改进的地方呢？

可以使用 `CompletableFuture` 类来改进！Java8 的 `CompletableFuture` 提供了很多对多线程友好的方法，使用它可以很方便地为我们编写多线程程序，什么异步、串行、并行或者等待所有线程执行完任务什么的都非常方便。

```
CompletableFuture<Void> task1 =
    CompletableFuture.supplyAsync(() ->{
        //自定义业务操作
    });
    .....
CompletableFuture<Void> task6 =
    CompletableFuture.supplyAsync(() ->{
        //自定义业务操作
    });
    .....
CompletableFuture<Void>
headerFuture=CompletableFuture.allOf(task1,.....,task6);

try {
    headerFuture.join();
} catch (Exception ex) {
    .....
}
System.out.println("all done. ");
```

上面的代码还可以接续优化，当任务过多的时候，把每一个 task 都列出来不太现实，可以考虑通过循环来添加任务。

```
//文件夹位置  
List<String> filePaths = Arrays.asList(...)  
// 异步处理所有文件  
List<CompletableFuture<String>> fileFutures = filePaths.stream()  
    .map(filePath -> doSomeThing(filePath))  
    .collect(Collectors.toList());  
// 将他们合并起来  
CompletableFuture<Void> allFutures = CompletableFuture.allOf(  
    fileFutures.toArray(new CompletableFuture[fileFutures.size()]))  
;
```

## 2.4.1. Reference

- 《深入理解 Java 虚拟机》
- 《实战 Java 高并发程序设计》
- 《Java 并发编程的艺术》
- <http://www.cnblogs.com/waterystone/p/4920797.html>
- <https://www.cnblogs.com/chengxiao/archive/2017/07/24/7141160.html>
- <https://www.journaldev.com/1076/java-threadlocal-example>

## 2.4. JVM

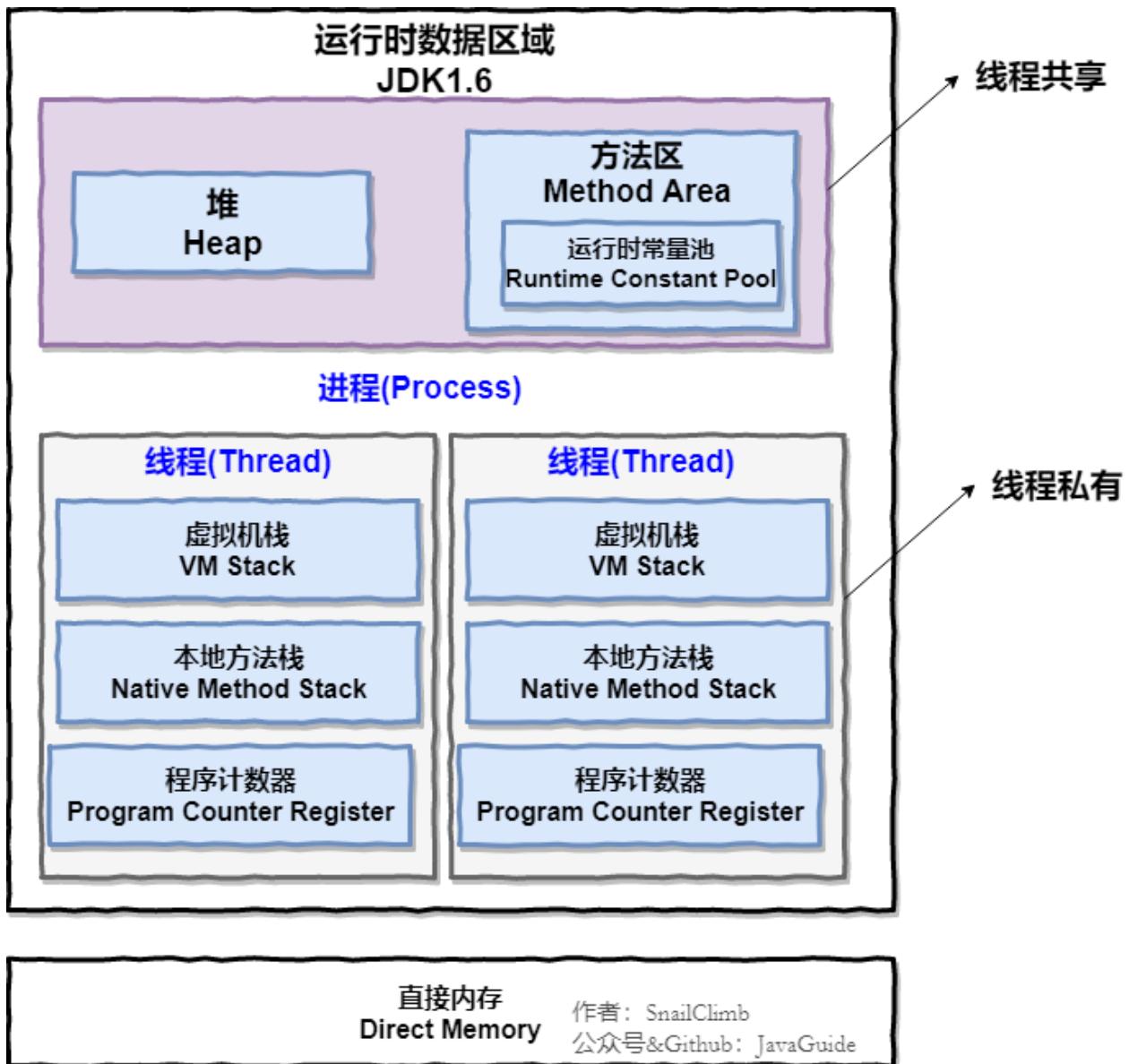
作者：Guide哥。

介绍：Github 70k Star 项目 [JavaGuide](#)（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

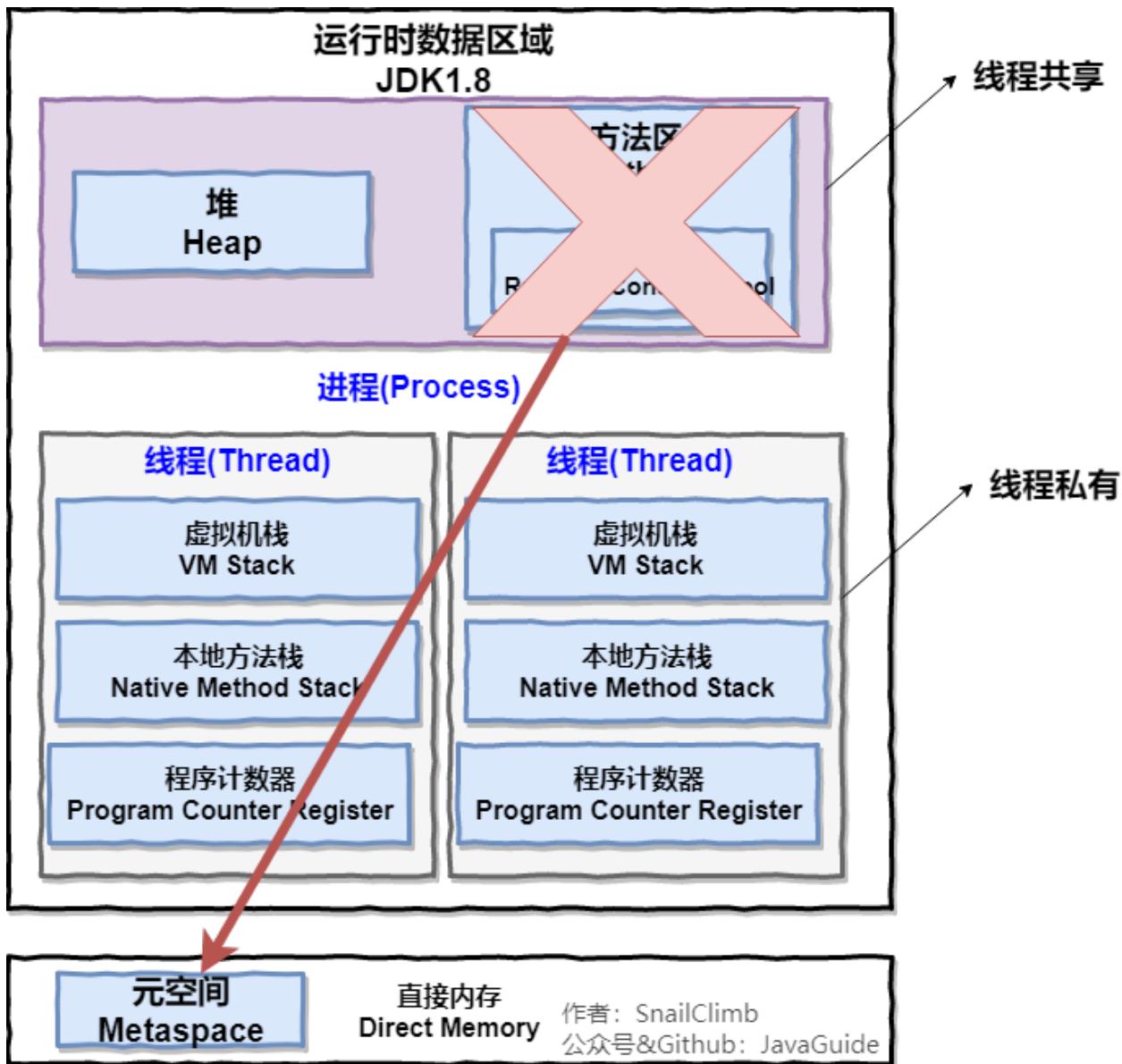
### 2.4.1. 介绍下 Java 内存区域(运行时数据区)

Java 虚拟机在执行 Java 程序的过程中会把它管理的内存划分成若干个不同的数据区域。JDK.1.8 和之前的版本略有不同，下面会介绍到。

JDK 1.8 之前：



JDK 1.8 :



线程私有的:

- 程序计数器
- 虚拟机栈
- 本地方法栈

线程共享的:

- 堆
- 方法区
- 直接内存 (非运行时数据区的一部分)

### 2.4.1.1. 程序计数器

程序计数器是一块较小的内存空间，可以看作是当前线程所执行的字节码的行号指示器。字节码解释器工作时通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等功能都需要依赖这个计数器来完成。

另外，为了线程切换后能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，各线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

从上面的介绍中我们知道程序计数器主要有两个作用：

1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

注意：程序计数器是唯一一个不会出现 `OutOfMemoryError` 的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。

### 2.4.1.2. Java 虚拟机栈

与程序计数器一样，Java 虚拟机栈也是线程私有的，它的生命周期和线程相同，描述的是 Java 方法执行的内存模型，每次方法调用的数据都是通过栈传递的。

Java 内存可以粗略的区分为堆内存（Heap）和栈内存（Stack），其中栈就是现在说的虚拟机栈，或者说是虚拟机栈中局部变量表部分。（实际上，Java 虚拟机栈是由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法出口信息。）

局部变量表主要存放了编译期可知的各种数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference 类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）。

Java 虚拟机栈会出现两种错误：`StackOverFlowError` 和 `OutOfMemoryError`。

- `StackOverFlowError`：若 Java 虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前 Java 虚拟机栈的最大深度的时候，就抛出 `StackOverFlowError` 错误。
- `OutOfMemoryError`：若 Java 虚拟机堆中没有空闲内存，并且垃圾回收器也无法提供更多内存的话。就会抛出 `OutOfMemoryError` 错误。

Java 虚拟机栈也是线程私有的，每个线程都有各自的 Java 虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。

扩展：那么方法/函数如何调用？

Java 栈可用类比数据结构中栈，Java 栈中保存的主要内容是栈帧，每一次函数调用都会有一个对应的栈帧被压入 Java 栈，每一个函数调用结束后，都会有一个栈帧被弹出。

Java 方法有两种返回方式：

1. return 语句。
2. 抛出异常。

不管哪种返回方式都会导致栈帧被弹出。

#### 2.4.1.3. 本地方法栈

和虚拟机栈所发挥的作用非常相似，区别是：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。在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。

方法执行完毕后相应的栈帧也会出栈并释放内存空间，也会出现 `StackOverFlowError` 和 `OutOfMemoryError` 两种错误。

#### 2.4.1.4. 堆

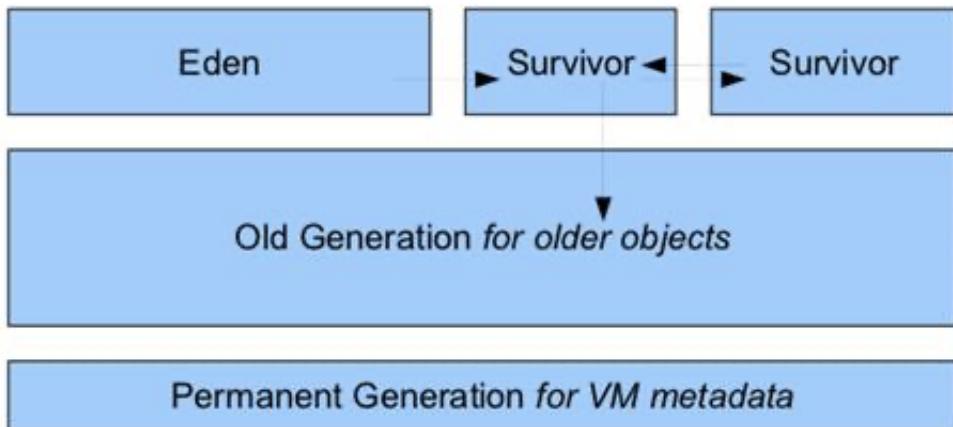
Java 虚拟机所管理的内存中最大的一块，Java 堆是所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。

Java 世界中“几乎”所有的对象都在堆中分配，但是，随着 JIT 编译期的发展与逃逸分析技术逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化，所有的对象都分配到堆上也渐渐变得不那么“绝对”了。从 jdk 1.7 开始已经默认开启逃逸分析，如果某些方法中的对象引用没有被返回或者未被外面使用（也就是未逃逸出去），那么对象可以直接在栈上分配内存。

Java 堆是垃圾收集器管理的主要区域，因此也被称作 **GC 堆 (Garbage Collected Heap)**。从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代；再细致一点有：Eden 空间、From Survivor、To Survivor 空间等。进一步划分的目的是更好地回收内存，或者更快地分配内存。

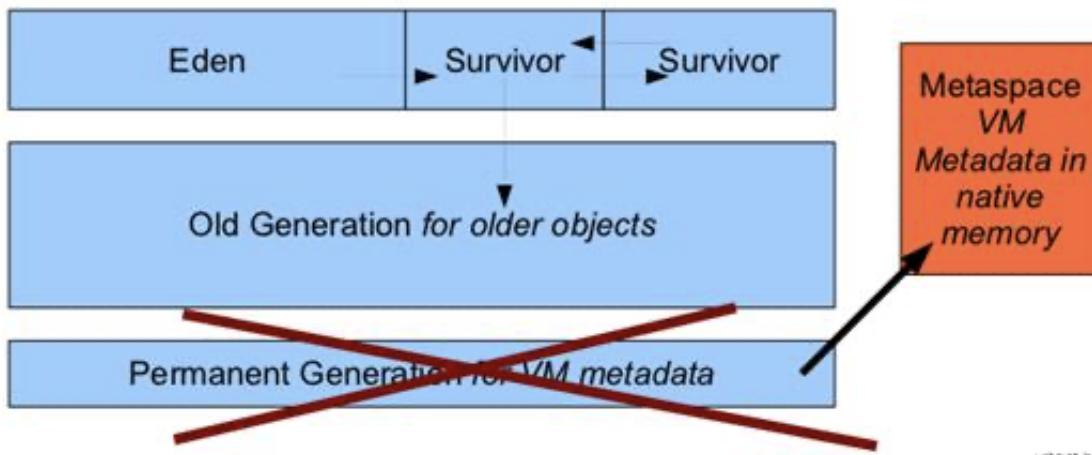
在 JDK 7 版本及 JDK 7 版本之前，堆内存被通常被分为下面三部分：

1. 新生代内存(Young Generation)
2. 老年代(Old Generation)
3. 永生代(Permanent Generation)



© 腾讯技术社区

JDK 8 版本之后方法区（HotSpot 的永久代）被彻底移除了（JDK1.7 就已经开始了），取而代之是元空间，元空间使用的是直接内存。



© 腾讯技术社区

上图所示的 **Eden** 区、两个 **Survivor** 区都属于新生代（为了区分，这两个 **Survivor** 区域按照顺序被命名为 **from** 和 **to**），中间一层属于老年带。

大部分情况，对象都会首先在 **Eden** 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 **s0** 或者 **s1**，并且对象的年龄还会加 1(**Eden** 区->**Survivor** 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年带中。对象晋升到老年带的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

修正 ([issue552](#))：“Hotspot遍历所有对象时，按照年龄从小到大对其所占用的大小进行累积，当累积的某个年龄大小超过了survivor区的一半时，取这个年龄和 MaxTenuringThreshold中更小的一个值，作为新的晋升年龄阈值”。

动态年龄计算的代码如下

```
uint ageTable::compute_tenuring_threshold(size_t survivor_capacity) {
    //survivor_capacity是survivor空间的大小
    size_t desired_survivor_size = (size_t)((double)
        survivor_capacity)*TargetSurvivorRatio)/100);
    size_t total = 0;
    uint age = 1;
    while (age < table_size) {
        total += sizes[age];//sizes数组是每个年龄段对象大小
        if (total > desired_survivor_size) break;
        age++;
    }
    uint result = age < MaxTenuringThreshold ? age : MaxTenuringThreshold;
    ...
}
```

堆这里最容易出现的就是 `OutOfMemoryError` 错误，并且出现这种错误之后的表现形式还会有几种，比如：

1. `OutOfMemoryError: GC Overhead Limit Exceeded`：当JVM花太多时间执行垃圾回收并且只能回收很少的堆空间时，就会发生此错误。
2. `java.lang.OutOfMemoryError: Java heap space`：假如在创建新的对象时，堆内存中的空间不足以存放新创建的对象，就会引发 `java.lang.OutOfMemoryError: Java heap space` 错误。(和本机物理内存无关，和你配置的内存大小有关！)
3. .....

#### 2.4.1.5. 方法区

方法区与 Java 堆一样，是各个线程共享的内存区域，它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。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 **Non-Heap**（非堆），目的应该是与 Java 堆区分开来。

方法区也被称为永久代。很多人都会分不清方法区和永久代的关系，为此我也查阅了文献。

#### 2.4.1.5.1. 方法区和永久代的关系

《Java 虚拟机规范》只是规定了有方法区这么个概念和它的作用，并没有规定如何去实现它。那么，在不同的 JVM 上方法区的实现肯定是不同的了。**方法区和永久代的关系很像 Java 中接口和类的关系，类实现了接口，而永久代就是 HotSpot 虚拟机对虚拟机规范中方法区的一种实现方式。**也就是说，永久代是 HotSpot 的概念，方法区是 Java 虚拟机规范中的定义，是一种规范，而永久代是一种实现，一个是标准一个是实现，其他的虚拟机实现并没有永久代这一说法。

#### 2.4.1.5.2. 常用参数

JDK 1.8 之前永久代还没被彻底移除的时候通常通过下面这些参数来调节方法区大小

```
-XX:PermSize=N //方法区（永久代）初始大小  
-XX:MaxPermSize=N //方法区（永久代）最大大小，超过这个值将会抛出 OutOfMemoryError 异常：java.lang.OutOfMemoryError: PermGen
```

相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入方法区后就“永远存在”了。

JDK 1.8 的时候，方法区（HotSpot 的永久代）被彻底移除了（JDK1.7 就已经开始了），取而代之是元空间，元空间使用的是直接内存。

下面是一些常用参数：

```
-XX:MetaspaceSize=N //设置 Metaspace 的初始（和最小大小）  
-XX:MaxMetaspaceSize=N //设置 Metaspace 的最大大小
```

与永久代很大的不同就是，如果不指定大小的话，随着更多类的创建，虚拟机会耗尽所有可用的系统内存。

#### 2.4.1.5.3. 为什么要将永久代 (PermGen) 替换为元空间 (MetaSpace) 呢？

- 整个永久代有一个 JVM 本身设置固定大小上限，无法进行调整，而元空间使用的是直接内存，受本机可用内存的限制，虽然元空间仍旧可能溢出，但是比原来出现的几率会更小。

当你元空间溢出时会得到如下错误： java.lang.OutOfMemoryError: MetaSpace

你可以使用 `-XX: MaxMetaspaceSize` 标志设置最大元空间大小，默认值为 `unlimited`，这意味着它只受系统内存的限制。`-XX: MetaspaceSize` 调整标志定义元空间的初始大小如果未指定此标志，则 Metaspace 将根据运行时的应用程序需求动态地重新调整大小。

2. 元空间里面存放的是类的元数据，这样加载多少类的元数据就不由 `MaxPermSize` 控制了，而由系统的实际可用空间来控制，这样能加载的类就更多了。
3. 在 JDK8，合并 HotSpot 和 JRockit 的代码时，JRockit 从来没有一个叫永久代的东西，合并之后就没有必要额外的设置这么一个永久代的地方了。

#### 2.4.1.6. 运行时常量池

运行时常量池是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有常量池表（用于存放编译期生成的各种字面量和符号引用）

既然运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出 `OutOfMemoryError` 错误。

~~JDK1.7 及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆（Heap）中开辟了一块区域存放运行时常量池。~~

修正([issue747](#), [reference](#)):

1. **JDK1.7之前**运行时常量池逻辑包含字符串常量池存放在方法区, 此时hotspot虚拟机对方法区的实现为永久代
2. **JDK1.7**字符串常量池被从方法区拿到了堆中, 这里没有提到运行时常量池,也就是说字符串常量池被单独拿到堆,运行时常量池剩下的东西还在方法区, 也就是hotspot中的永久代。
3. **JDK1.8** hotspot移除了永久代用元空间(Metaspace)取而代之, 这时候字符串常量池还在堆, 运行时常量池还在方法区, 只不过方法区的实现从永久代变成了元空间(Metaspace)

相关问题：JVM 常量池中存储的是对象还是引用呢？：<https://www.zhihu.com/question/57109429/answer/151717241> by RednaxelaFX

#### 2.4.1.7. 直接内存

直接内存并不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用。而且也可能导致 `OutOfMemoryError` 错误出现。

JDK1.4 中新加入的 **NIO(New Input/Output)** 类，引入了一种基于通道（**Channel**）与缓存区（**Buffer**）的 I/O 方式，它可以直接使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 **DirectByteBuffer** 对象作为这块内存的引用进行操作。这样就能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆之间来回复制数据。

本机直接内存的分配不会受到 Java 堆的限制，但是，既然是内存就会受到本机总内存大小以及处理器寻址空间的限制。

## 2.4.2. 说一下Java对象的创建过程

下图便是 Java 对象的创建过程，我建议最好是能默写出来，并且要掌握每一步在做什么。

### 2.4.2.1. Step1:类加载检查

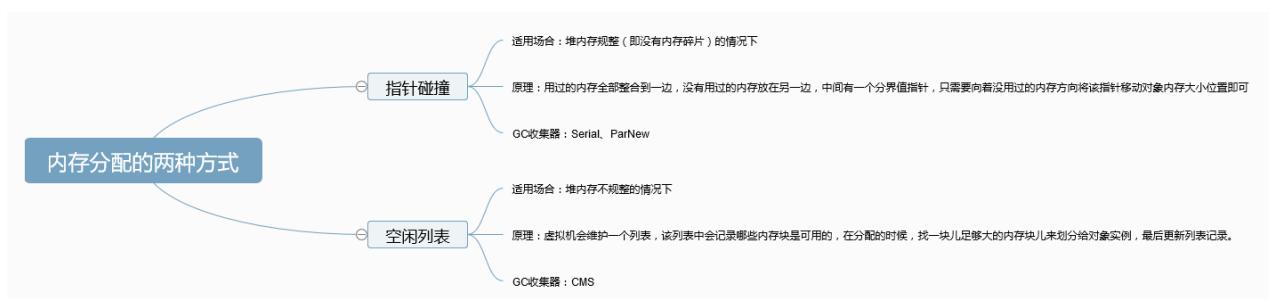
虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

### 2.4.2.2. Step2:分配内存

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。分配方式有“指针碰撞”和“空闲列表”两种，选择哪种分配方式由 Java 堆是否规整决定，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。

内存分配的两种方式：（补充内容，需要掌握）

选择以上两种方式中的哪一种，取决于 Java 堆内存是否规整。而 Java 堆内存是否规整，取决于 GC 收集器的算法是“标记-清除”，还是“标记-整理”（也称作“标记-压缩”），值得注意的是，复制算法内存也是规整的



内存分配并发问题（补充内容，需要掌握）

在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很频繁的事情，作为虚拟机来说，必须要保证线程是安全的，通常来讲，虚拟机采用两种方式来保证线程安全：

- **CAS+失败重试**： CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。**虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性。**
- **TLAB**：为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配

#### 2.4.2.3. Step3: 初始化零值

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

#### 2.4.2.4. Step4: 设置对象头

初始化零值完成之后，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。**这些信息存放在对象头中**。另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

#### 2.4.2.5. Step5: 执行 init 方法

在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚刚开始，`<init>` 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令之后会接着执行 `<init>` 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全产生出来。

### 2.4.3. 对象的访问定位有哪两种方式？

建立对象就是为了使用对象，我们的Java程序通过栈上的 reference 数据来操作堆上的具体对象。对象的访问方式有虚拟机实现而定，目前主流的访问方式有①使用句柄和②直接指针两种：

1. **句柄**：如果使用句柄的话，那么Java堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息；

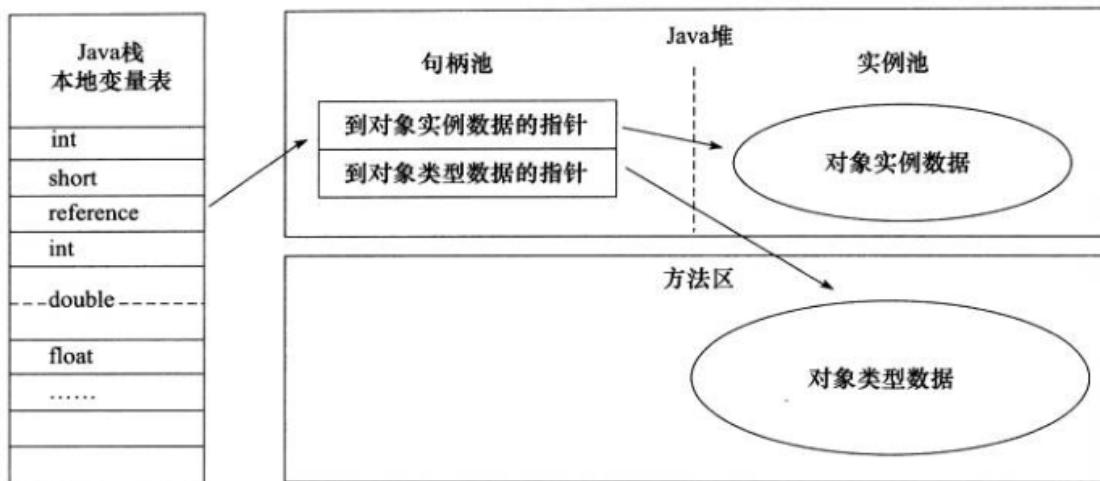


图 2-2 通过句柄访问对象

2. 直接指针：如果使用直接指针访问，那么 Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而reference 中存储的直接就是对象的地址。

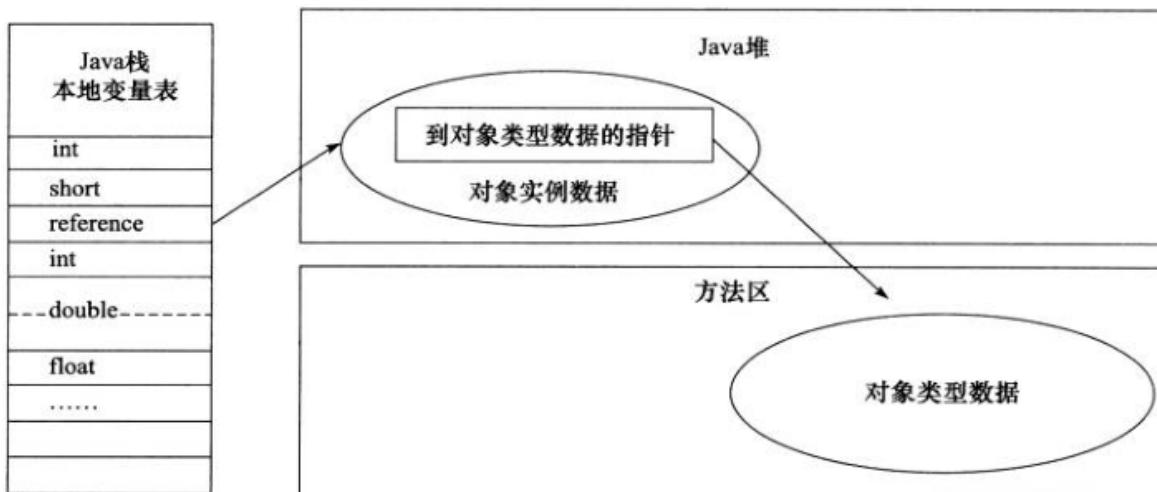


图 2-3 通过直接指针访问对象

这两种对象访问方式各有优势。使用句柄来访问的最大好处是 reference 中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而 reference 本身不需要修改。使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销。

#### 2.4.4. 简单聊聊 JVM 内存分配与回收

Java 的自动内存管理主要是针对对象内存的回收和对象内存的分配。同时，Java 自动内存管理最核心的功能是 堆 内存中对象的分配与回收。

Java 堆是垃圾收集器管理的主要区域，因此也被称作**GC 堆（Garbage Collected Heap）**。从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代；再细致一点有：Eden 空间、From Survivor、To Survivor 空间等。进一步划分的目的是更好地回收内存，或者更快地分配内存。

## 堆空间的基本结构：

上图所示的 Eden 区、From Survivor0("From") 区、To Survivor1("To") 区都属于新生代， Old Memory 区属于老年带。

大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s0 或者 s1，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年带中。对象晋升到老年带的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

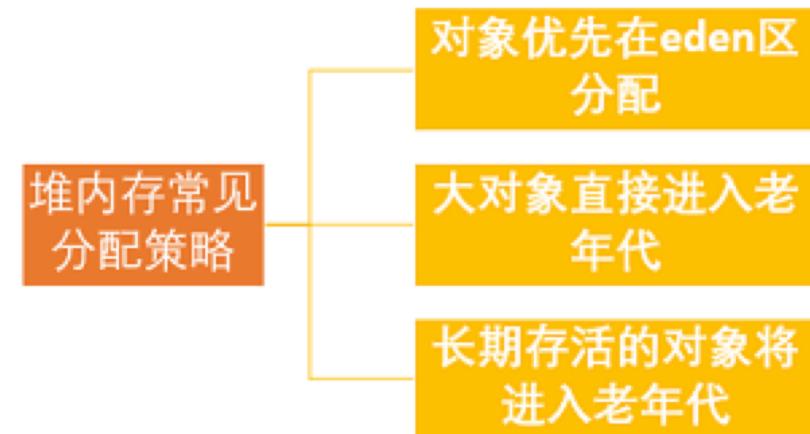
修正 ([issue552](#))：“Hotspot 遍历所有对象时，按照年龄从小到大对所占用的大小进行累积，当累积的某个年龄大小超过了 survivor 区的一半时，取这个年龄和 MaxTenuringThreshold 中更小的一个值，作为新的晋升年龄阈值”。

动态年龄计算的代码如下

```
uint ageTable::compute_tenuring_threshold(size_t survivor_capacity) {
    //survivor_capacity是survivor空间的大小
    size_t desired_survivor_size = (size_t)((((double)
        survivor_capacity)*TargetSurvivorRatio)/100);
    size_t total = 0;
    uint age = 1;
    while (age < table_size) {
        total += sizes[age];//sizes数组是每个年龄段对象大小
        if (total > desired_survivor_size) break;
        age++;
    }
    uint result = age < MaxTenuringThreshold ? age : MaxTenuringThreshold;
    ...
}
```

经过这次 GC 后，Eden 区和"From"区已经被清空。这个时候，"From"和"To"会交换他们的角色，也就是新的"To"就是上次 GC 前的"From"，新的"From"就是上次 GC 前的"To"。不管怎样，都会保证名为 To 的 Survivor 区域是空的。Minor GC 会一直重复这样的过程，直到"To"区被填满，"To"区被填满之后，会将所有对象移动到老年带中。

## 2.4.5. 说一下堆内存中对象的分配的基本策略



公众号：JavaGuide

### 2.4.5.1. 对象优先在 eden 区分配

目前主流的垃圾收集器都会采用分代回收算法，因此需要将堆内存分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

大多数情况下，对象在新生代中 eden 区分配。当 eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC.下面我们来进行实际测试以下。

测试：

```
public class GCTest {  
  
    public static void main(String[] args) {  
        byte[] allocation1, allocation2;  
        allocation1 = new byte[30900*1024];  
        //allocation2 = new byte[900*1024];  
    }  
}
```

通过以下方式运行：

添加的参数： -XX:+PrintGCDetails

运行结果 (红色字体描述有误，应该是对于 JDK1.7 的永久代)：

```

Heap
PSYoungGen      新生代
total 38400K, used 33280K [0x00000000d5d00000, 0x00000000d8780000, 0x0000000010000000)
eden space 33280K, 100% used [0x00000000d5d00000, 0x00000000d7d80000, 0x00000000d7d80000)
from space 5120K, 0% used [0x00000000d8280000, 0x00000000d8280000, 0x00000000d8780000)
to space 5120K, 0% used [0x00000000d7d80000, 0x00000000d7d80000, 0x00000000d8280000)
ParOldGen        老年代
total 87552K, used 0K [0x0000000081600000, 0x0000000081600000, 0x00000000d5d00000)
object space 87552K, 0% used [0x0000000081600000, 0x0000000081600000, 0x00000000d5d00000)
Metaspace
used 2621K, capacity 4486K, committed 4864K, reserved 1056768K
class space     元空间对应于JDK1.8的永久代

```

从上图我们可以看出 eden 区内存几乎已经被分配完全（即使程序什么也不做，新生代也会使用 2000 多 k 内存）。假如我们再为 allocation2 分配内存会出现什么情况呢？

```
allocation2 = new byte[900*1024];
```

简单解释一下为什么会出现这种情况：因为给 allocation2 分配内存的时候 eden 区内存几乎已经被分配完了，我们刚刚讲了当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC。GC 期间虚拟机又发现 allocation1 无法存入 Survivor 空间，所以只好通过 分配担保机制 把新生代的对象提前转移到老年代中去，老年代上的空间足够存放 allocation1，所以不会出现 Full GC。执行 Minor GC 后，后面分配的对象如果能够存在 eden 区的话，还是会在 eden 区分配内存。可以执行如下代码验证：

```

public class GCTest {

    public static void main(String[] args) {
        byte[] allocation1, allocation2, allocation3, allocation4, allocation5;
        allocation1 = new byte[32000*1024];
        allocation2 = new byte[1000*1024];
        allocation3 = new byte[1000*1024];
        allocation4 = new byte[1000*1024];
        allocation5 = new byte[1000*1024];
    }
}

```

## 2.4.5.2. 大对象直接进入老年代

大对象就是需要大量连续内存空间的对象（比如：字符串、数组）。

为什么要这样呢？

为了避免为大对象分配内存时由于分配担保机制带来的复制而降低效率。

## 2.4.5.3. 长期存活的对象将进入老年代

既然虚拟机采用了分代收集的思想来管理内存，那么内存回收时就必须能识别哪些对象应放在新生代，哪些对象应放在老年代中。为了做到这一点，虚拟机给每个对象一个对象年龄（Age）计数器。

如果对象在 Eden 出生并经过第一次 Minor GC 后仍然能够存活，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 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），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

## 2.4.5.4. 动态对象年龄判定

大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s0 或者 s1，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

修正 ([issue552](#))：“Hotspot 遍历所有对象时，按照年龄从小到大对所占用的大小进行累积，当累积的某个年龄大小超过了 survivor 区的一半时，取这个年龄和 MaxTenuringThreshold 中更小的一个值，作为新的晋升年龄阈值”。

动态年龄计算的代码如下

```
uint ageTable::compute_tenuring_threshold(size_t survivor_capacity) {
    //survivor_capacity是survivor空间的大小
    size_t desired_survivor_size = (size_t)((((double)
        survivor_capacity)*TargetSurvivorRatio)/100);
    size_t total = 0;
    uint age = 1;
    while (age < table_size) {
        total += sizes[age];//sizes数组是每个年龄段对象大小
        if (total > desired_survivor_size) break;
        age++;
    }
}
```

```
        }
        uint result = age < MaxTenuringThreshold ? age : MaxTenuringThreshold;
        ...
    }
```

额外补充说明([issue672](#))：关于默认的晋升年龄是 15，这个说法的来源大部分都是《深入理解 Java 虚拟机》这本书。

如果你去 Oracle 的官网阅读[相关的虚拟机参数](#)，你会发现 -

XX:MaxTenuringThreshold=threshold 这里有个说明

**Sets the maximum tenuring threshold for use in adaptive GC sizing. The largest value is 15. The default value is 15 for the parallel (throughput) collector, and 6 for the CMS collector.** 默认晋升年龄并不都是 15，这个是要区分垃圾收集器的，CMS 就是 6.

#### 2.4.5.5. 主要进行 gc 的区域

周志明先生在《深入理解 Java 虚拟机》第二版中 P92 如是写道：

“老年代 GC (Major GC/Full GC)，指发生在老年代的 GC.....”

上面的说法已经在《深入理解 Java 虚拟机》第三版中被改正过来了。感谢 R 大的回答：



RednaxelaFX

计算机科学等 7 个话题下的优秀回答者

Asterisk、柳树等 614 人赞同了该回答

针对HotSpot VM的实现，它里面的GC其实准确分类只有两大种：

- Partial GC：并不收集整个GC堆的模式
  - Young GC：只收集young gen的GC
  - Old GC：只收集old gen的GC。只有CMS的concurrent collection是这个模式
  - Mixed GC：收集整个young gen以及部分old gen的GC。只有G1有这个模式
- Full GC：收集整个堆，包括young gen、old gen、perm gen（如果存在的话）等所有部分的模式。

Major GC通常是跟full GC是等价的，收集整个GC堆。但因为HotSpot VM发展了这么多年，外界对各种名词的解读已经完全混乱了，当有人说“major GC”的时候一定要问清楚他想要指的是上面的full GC还是old GC。

最简单的分代式GC策略，按HotSpot VM的serial GC的实现来看，触发条件是：

- young GC：当young gen中的eden区分配满的时候触发。注意young GC中有部分存活对象会晋升到old gen，所以young GC后old gen的占用量通常会有所升高。
- full GC：当准备要触发一次young GC时，如果发现统计数据说之前young GC的平均晋升大小比目前old gen剩余的空间大，则不会触发young GC而是转为触发full GC（因为HotSpot VM的GC里，除了CMS的concurrent collection之外，其它能收集old gen的GC都会同时收集整个GC堆，包括young gen，所以不需要事先触发一次单独的young GC）；或者，如果有perm gen的话，要在perm gen分配空间但已经没有足够空间时，也要触发一次full GC；或者System.gc()、heap dump带GC，默认也是触发full GC。

HotSpot VM里其它非并发GC的触发条件复杂一些，不过大致的原理与上面说的其实一样。

当然也总有例外。Parallel Scavenge (-XX:+UseParallelGC) 框架下，默认是在要触发full GC前先执行一次young GC，并且两次GC之间能让应用程序稍微运行一小下，以期降低full GC的暂停时间（因为young GC会尽量清理了young gen的死对象，减少了full GC的工作量）。控制这个行为的VM参数是-XX:+ScavengeBeforeFullGC。这是HotSpot VM里的奇葩嗯。可跳传送门围观：[JVM full GC的奇怪现象，求解惑？ - RednaxelaFX 的回答](#)

并发GC的触发条件就不太一样。以CMS GC为例，它主要是定时去检查old gen的使用量，当使用量超过了触发比例就会启动一次CMS GC，对old gen做并发收集。

编辑于 2017-02-06

总结：

针对 HotSpot VM 的实现，它里面的 GC 其实准确分类只有两大种：

部分收集 (Partial GC)：

- 新生代收集 (Minor GC / Young GC)：只对新生代进行垃圾收集；
- 老年代收集 (Major GC / Old GC)：只对老年代进行垃圾收集。需要注意的是 Major GC 在

有的语境中也用于指代整堆收集；

- 混合收集（Mixed GC）：对整个新生代和部分老年代进行垃圾收集。

整堆收集(Full GC)：收集整个Java堆和方法区。

## 2.4.6. 如何判断对象是否死亡？(两种方法)

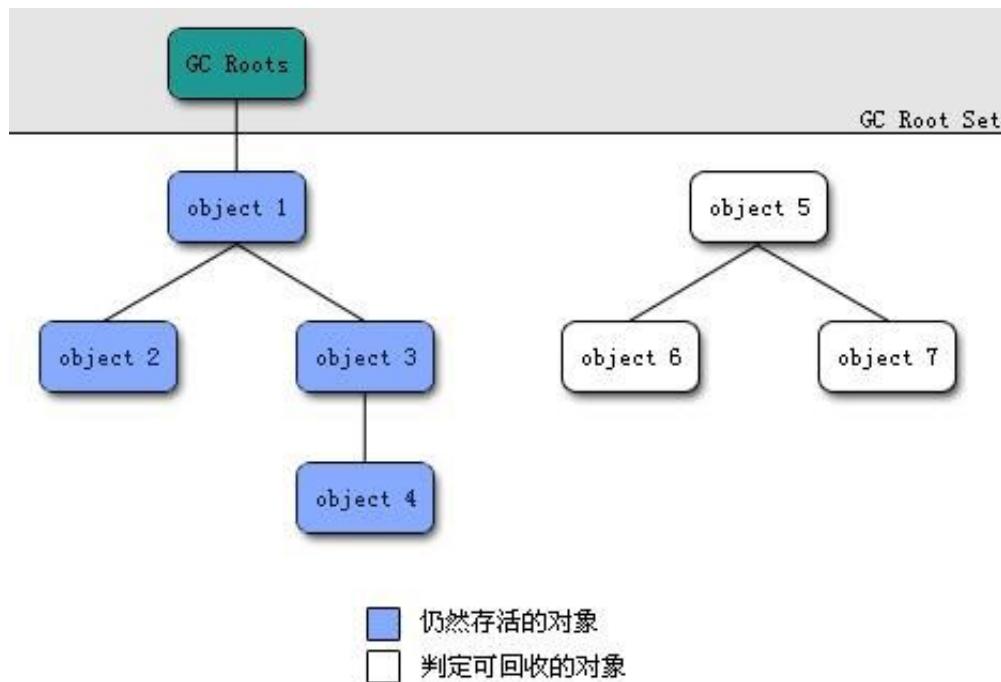
堆中几乎放着所有的对象实例，对堆垃圾回收前的第一步就是要判断哪些对象已经死亡（即不能再被任何途径使用的对象）。

### 2.4.6.1. 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，计数器就加1；当引用失效，计数器就减1；任何时候计数器为0的对象就是不可能再被使用的。

### 2.4.6.2. 可达性分析算法

这个算法的基本思想就是通过一系列的称为“**GC Roots**”的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连的话，则证明此对象是不可用的。



## 2.4.7. 简单的介绍一下强引用,软引用,弱引用,虚引用

无论是通过引用计数法判断对象引用数量，还是通过可达性分析法判断对象的引用链是否可达，判定对象的存活都与“引用”有关。

JDK1.2之前，Java中引用的定义很传统：如果reference类型的数据存储的数值代表的是另一块内存的起始地址，就称这块内存代表一个引用。

JDK1.2以后，Java对引用的概念进行了扩充，将引用分为强引用、软引用、弱引用、虚引用四种（引用强度逐渐减弱）

#### 2.4.7.1. 强引用(StrongReference)

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空间不足，Java虚拟机宁愿抛出`OutOfMemoryError`错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

#### 2.4.7.2. 软引用(SoftReference)

如果一个对象只具有软引用，那就类似于可有可无的生活用品。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（`ReferenceQueue`）联合使用，如果软引用所引用的对象被垃圾回收，JAVA虚拟机就会把这个软引用加入到与之关联的引用队列中。

#### 2.4.7.3. 弱引用(WeakReference)

如果一个对象只具有弱引用，那就类似于可有可无的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（`ReferenceQueue`）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

### 4. 虚引用 (PhantomReference)

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

虚引用主要用来跟踪对象被垃圾回收的活动。

虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（`ReferenceQueue`）联合使用。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，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

特别注意，在程序设计中一般很少使用弱引用与虚引用，使用软引用的情况较多，这是因为软引用可以加速JVM对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生。

## 2.4.8. 如何判断一个常量是废弃常量？

运行时常量池主要回收的是废弃的常量。那么，我们如何判断一个常量是废弃常量呢？

假如在常量池中存在字符串 "abc"，如果当前没有任何String对象引用该字符串常量的话，就说明常量 "abc" 就是废弃常量，如果这时发生内存回收的话而且有必要的话，"abc" 就会被系统清理出常量池。

## 2.4.9. 如何判断一个类是无用的类？

方法区主要回收的是无用的类，那么如何判断一个类是无用的类的呢？

判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面 3 个条件才能算是“无用的类”：

- 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 已经被回收。
- 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样不使用了就会必然被回收。

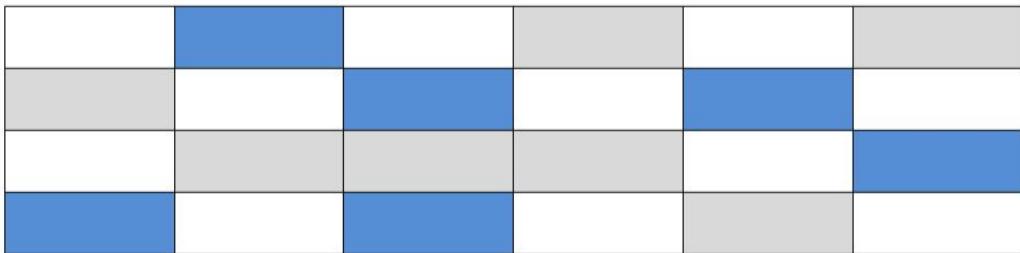
## 2.4.10. 垃圾收集有哪些算法，各自的特点？

### 2.4.10.1. 标记-清除算法

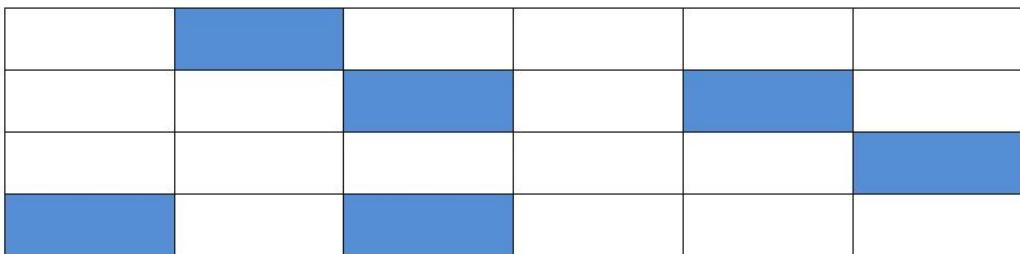
该算法分为“标记”和“清除”阶段：首先标记出所有不需要回收的对象，在标记完成后统一回收掉所有没有被标记的对象。它是最基础的收集算法，后续的算法都是对其不足进行改进得到。这种垃圾收集算法会带来两个明显的问题：

1. 效率问题
2. 空间问题（标记清除后会产生大量不连续的碎片）

## 内存整理前



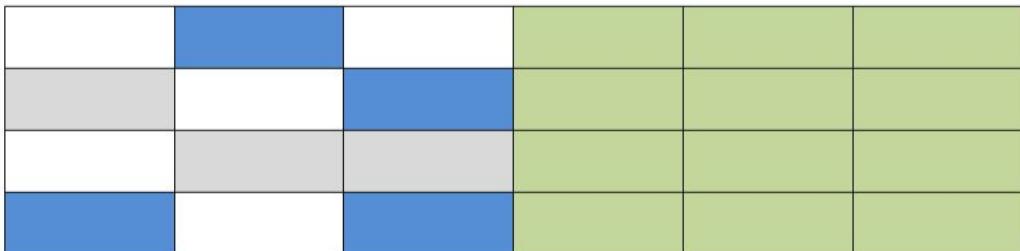
## 内存整理后



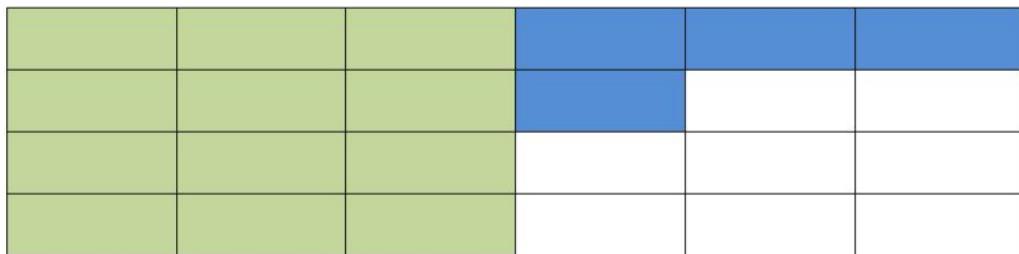
### 2.4.10.2. 复制算法

为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

## 内存整理前



## 内存整理后



### 2.4.10.3. 标记-整理算法

根据老年代的特点提出的一种标记算法，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回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。

### 2.4.10.4. 分代收集算法

当前虚拟机的垃圾收集都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。

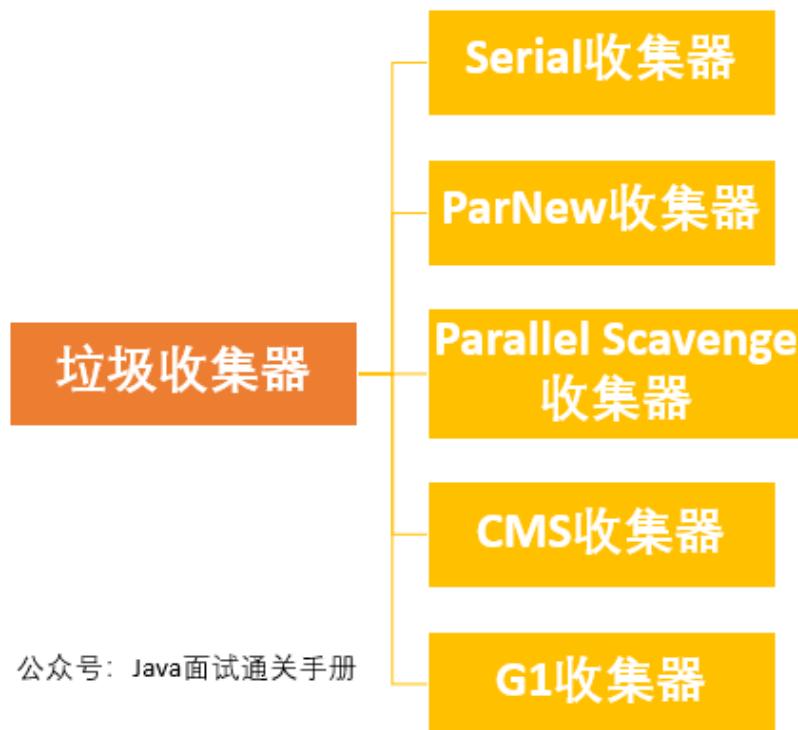
延伸面试问题： HotSpot 为什么要分为新生代和老年代？

根据上面的对分代收集算法的介绍回答。

## 2.4.11. HotSpot 为什么要分为新生代和老年代？

主要是为了提升 GC 效率。上面提到的分代收集算法已经很好的解释了这个问题。

## 2.4.12. 常见的垃圾回收器有那些？



如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

虽然我们对各个收集器进行比较，但并非要挑选出一个最好的收集器。因为知道现在为止还没有最好的垃圾收集器出现，更加没有万能的垃圾收集器，**我们能做的就是根据具体应用场景选择适合自己的垃圾收集器**。试想一下：如果有一种四海之内、任何场景下都适用的完美收集器存在，那么我们的 HotSpot 虚拟机就不会实现那么多不同的垃圾收集器了。

### 2.4.12.1. Serial 收集器

Serial（串行）收集器是最基本、历史最悠久的垃圾收集器了。大家看名字就知道这个收集器是一个单线程收集器了。它的“**单线程**”的意义不仅仅意味着它只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集工作的时候必须暂停其他所有的工作线程（"**Stop The World**"），直到它收集结束。

新生代采用复制算法，老年代采用标记-整理算法。

虚拟机的设计者们当然知道 Stop The World 带来的不良用户体验，所以在后续的垃圾收集器设计中停顿时间在不断缩短（仍然还有停顿，寻找最优秀的垃圾收集器的过程仍然在继续）。

但是 Serial 收集器有没有优于其他垃圾收集器的地方呢？当然有，它简单而高效（与其他收集器的单线程相比）。Serial 收集器由于没有线程交互的开销，自然可以获得很高的单线程收集效率。Serial 收集器对于运行在 Client 模式下的虚拟机来说是个不错的选择。

#### 2.4.12.2. ParNew 收集器

ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和 Serial 收集器完全一样。

新生代采用复制算法，老年代采用标记-整理算法。

它是许多运行在 Server 模式下的虚拟机的首要选择，除了 Serial 收集器外，只有它能与 CMS 收集器（真正意义上的并发收集器，后面会介绍到）配合工作。

并行和并发概念补充：

- **并行 (Parallel)**：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。
- **并发 (Concurrent)**：指用户线程与垃圾收集线程同时执行（但不一定是并行，可能会交替执行），用户程序在继续运行，而垃圾收集器运行在另一个 CPU 上。

#### 2.4.12.3. Parallel Scavenge 收集器

Parallel Scavenge 收集器也是使用复制算法的多线程收集器，它看上去几乎和 ParNew 都一样。那么它有什么特别之处呢？

-XX:+UseParallelGC

使用 Parallel 收集器+ 老年代串行

-XX:+UseParallelOldGC

使用 Parallel 收集器+ 老年代并行

Parallel Scavenge 收集器关注点是吞吐量（高效率的利用 CPU）。CMS 等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）。所谓吞吐量就是 CPU 中用于运行用户代码的时间与 CPU 总消耗时间的比值。Parallel Scavenge 收集器提供了很多参数供用户找到最合适

的停顿时间或最大吞吐量，如果对于收集器运作不太了解，手工优化存在困难的时候，使用 Parallel Scavenge 收集器配合自适应调节策略，把内存管理优化交给虚拟机去完成也是一个不错的选择。

新生代采用复制算法，老年代采用标记-整理算法。

这是 JDK1.8 默认收集器

使用 `java -XX:+PrintCommandLineFlags -version` 命令查看

```
-XX:InitialHeapSize=262921408 -XX:MaxHeapSize=4206742528 -
XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers -
XX:+UseCompressedOops -XX:+UseParallelGC
java version "1.8.0_211"
Java(TM) SE Runtime Environment (build 1.8.0_211-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.211-b12, mixed mode)
```

JDK1.8 默认使用的是 Parallel Scavenge + Parallel Old，如果指定了`-XX:+UseParallelGC` 参数，则默认指定了`-XX:+UseParallelOldGC`，可以使用`-XX:-UseParallelOldGC` 来禁用该功能

#### 2.4.12.4. Serial Old 收集器

Serial 收集器的老年代版本，它同样是一个单线程收集器。它主要有两大用途：一种用途是在 JDK1.5 以及以前的版本中与 Parallel Scavenge 收集器搭配使用，另一种用途是作为 CMS 收集器的后备方案。

#### 2.4.12.5. Parallel Old 收集器

Parallel Scavenge 收集器的老年代版本。使用多线程和“标记-整理”算法。在注重吞吐量以及 CPU 资源的场合，都可以优先考虑 Parallel Scavenge 收集器和 Parallel Old 收集器。

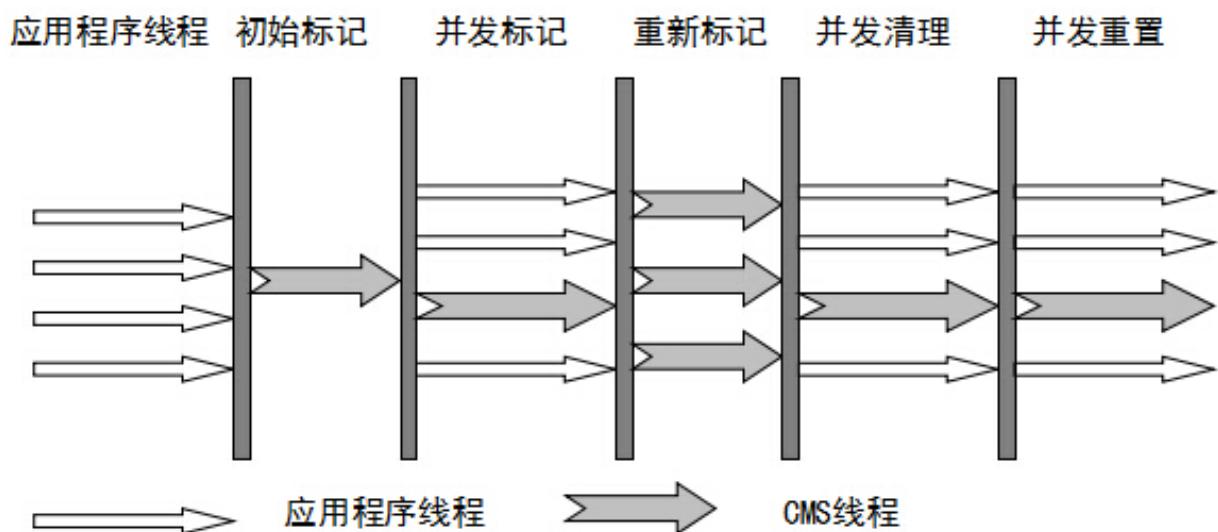
#### 2.4.12.6. CMS 收集器

CMS (Concurrent Mark Sweep) 收集器是一种以获取最短回收停顿时间为 目标的收集器。它非常符合在注重用户体验的应用上使用。

CMS (Concurrent Mark Sweep) 收集器是 HotSpot 虚拟机第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作。

从名字中的**Mark Sweep**这两个词可以看出，CMS 收集器是一种“标记-清除”算法实现的，它的运作过程相比于前面几种垃圾收集器来说更加复杂一些。整个过程分为四个步骤：

- **初始标记**：暂停所有的其他线程，并记录下直接与 root 相连的对象，速度很快；
- **并发标记**：同时开启 GC 和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以 GC 线程无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方。
- **重新标记**：重新标记阶段就是为了修正并发标记期间因为用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短
- **并发清理**：开启用户线程，同时 GC 线程开始对未标记的区域做清扫。



从它的名字就可以看出它是一款优秀的垃圾收集器，主要优点：**并发收集、低停顿**。但是它有下面三个明显的缺点：

- 对 CPU 资源敏感；
- 无法处理浮动垃圾；
- 它使用的回收算法-“标记-清除”算法会导致收集结束时会有大量空间碎片产生。

#### 2.4.12.7. G1 收集器

**G1 (Garbage-First)** 是一款面向服务器的垃圾收集器，主要针对配备多颗处理器及大容量内存的机器。以极高概率满足 GC 停顿时间要求的同时，还具备高吞吐量性能特征。

被视为 JDK1.7 中 HotSpot 虚拟机的一个重要进化特征。它具备一下特点：

- **并行与并发**：G1 能充分利用 CPU、多核环境下的硬件优势，使用多个 CPU (CPU 或者 CPU 核心) 来缩短 Stop-The-World 停顿时间。部分其他收集器原本需要停顿 Java 线程执

行的 GC 动作，G1 收集器仍然可以通过并发的方式让 java 程序继续执行。

- **分代收集**: 虽然 G1 可以不需要其他收集器配合就能独立管理整个 GC 堆，但是还是保留了分代的概念。
- **空间整合**: 与 CMS 的“标记--清理”算法不同，G1 从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。
- **可预测的停顿**: 这是 G1 相对于 CMS 的另一个大优势，降低停顿时间是 G1 和 CMS 共同的关注点，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 M 毫秒的时间片段内。

G1 收集器的运作大致分为以下几个步骤：

- 初始标记
- 并发标记
- 最终标记
- 筛选回收

G1 收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的 Region(这也就是它的名字 Garbage-First 的由来)。这种使用 Region 划分内存空间以及有优先级的区域回收方式，保证了 G1 收集器在有限时间内可以尽可能高的收集效率（把内存化整为零）。

#### 2.4.12.8. ZGC 收集器

与 CMS 中的 ParNew 和 G1 类似，ZGC 也采用标记-复制算法，不过 ZGC 对该算法做了重大改进。

在 ZGC 中出现 Stop The World 的情况会更少！

详情可以看：[《新一代垃圾回收器 ZGC 的探索与实践》](#) -----

## 三 计算机基础

### 3.1 计算机网络

作者：Guide哥。

介绍：Github 70k Star 项目 [JavaGuide](#) (公众号同名) 作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。