

# 《Java 突击面试真题手册》

## 2.1. Java 基础

这部分内容摘自 [JavaGuide](#) 下面几篇文章：

- [Java基础常见面试题总结\(上\)](#)
- [Java基础常见面试题总结\(中\)](#)
- [Java基础常见面试题总结\(下\)](#)

## JVM vs JDK vs JRE

### 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。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。

**JVM 并不是只有一种！只要满足 JVM 规范，每个公司、组织或者个人都可以开发自己的专属 JVM。** 也就是说我们平时接触到的 HotSpot VM 仅仅是是 JVM 规范的一种实现而已。

除了我们平时最常用的 HotSpot VM 外，还有 J9 VM、Zing VM、JRockit VM 等 JVM。维基百科上就有常见 JVM 的对比：[Comparison of Java virtual machines](#)，感兴趣的可以去看看。并且，你可以在 [Java SE Specifications](#) 上找到各个版本的 JDK 对应的 JVM 规范。

## Java Language and Virtual Machine Specifications

### Java SE 18

Released March 2022 as [JSR 393](#)



#### The Java Language Specification, Java SE 18 Edition

- [HTML](#) | [PDF](#)
- Preview feature: [Pattern Matching for switch](#)



#### The Java Virtual Machine Specification, Java SE 18 Edition

- [HTML](#) | [PDF](#)

### Java SE 17

Released September 2021 as [JSR 392](#)



#### The Java Language Specification, Java SE 17 Edition

- [HTML](#) | [PDF](#)
- Preview feature: [Pattern Matching for switch](#)



#### The Java Virtual Machine Specification, Java SE 17 Edition

- [HTML](#) | [PDF](#)

## JDK 和 JRE

JDK 是 Java Development Kit 缩写，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。

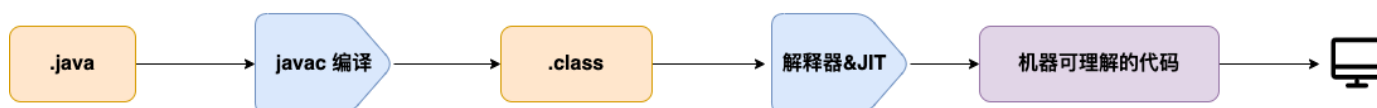
JRE 是 Java 运行时环境。它是运行已编译 Java 程序所需的所有内容的集合，包括 Java 虚拟机（JVM），Java 类库，java 命令和其他的一些基础构件。但是，它不能用于创建新程序。

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。

## 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 程序运行时相对来说还是高效的(不过, 和 C++, Rust, Go 等语言还是有一定差距的), 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的过程如下图所示:



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 (just-in-time compilation) 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), 而这也就是 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。

## 为什么不全部使用 AOT 呢?

AOT 可以提前编译节省启动时间, 那为什么不全部使用这种编译方式呢?

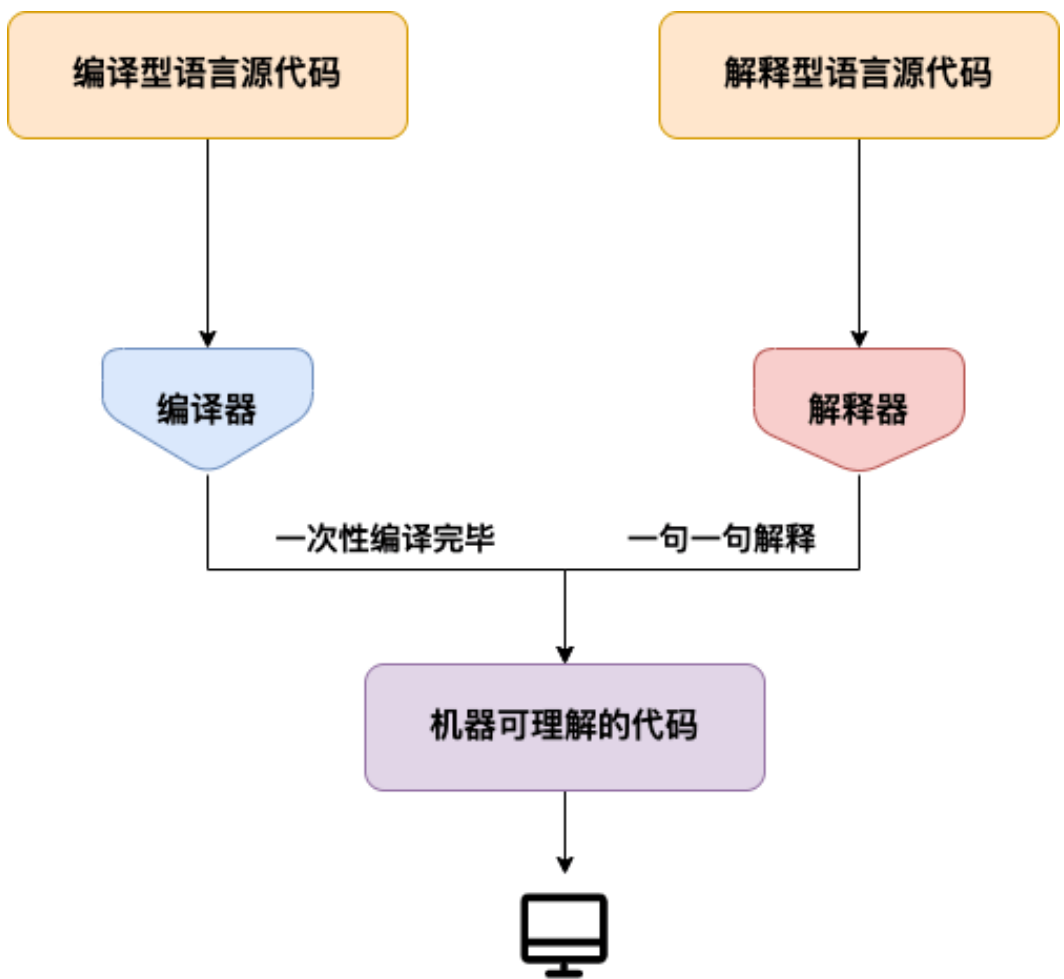
长话短说, 这和 Java 语言的动态特性有千丝万缕的联系了。举个例子, CGLIB 动态代理使用的是 ASM 技术, 而这种技术大致原理是运行时直接在内存中生成并加载修改后的字节码文件也就是 `.class` 文件, 如果全部使用 AOT 提前编译, 也就不能使用 ASM 技术了。为了支持类似的动态特性, 所以选择使用 JIT 即时编译器。

## 为什么说 Java 语言“编译与解释并存”？

其实这个问题我们讲字节码的时候已经提到过，因为比较重要，所以我们这里再提一下。

我们可以将高级编程语言按照程序的执行方式分为两种：

- **编译型**：**编译型语言** 会通过**编译器**将源代码一次性翻译成可被该平台执行的机器码。一般情况下，编译语言的执行速度比较快，开发效率比较低。常见的编译性语言有 C、C++、Go、Rust 等等。
- **解释型**：**解释型语言** 会通过**解释器**一句一句的将代码解释（interpret）为机器代码后再执行。解释型语言开发效率比较快，执行速度比较慢。常见的解释性语言有 Python、JavaScript、PHP 等等。



根据维基百科介绍：

为了改善编译语言的效率而发展出的**即时编译**技术，已经缩小了这两种语言间的差距。这种技术混合了编译语言与解释型语言的优点，它像编译语言一样，先把程序源代码编译成**字节码**。到执行期时，再将字节码直译，之后执行。**Java**与**LLVM**是这种技术的代表产物。

这是因为 Java 语言既具有编译型语言的特征，也具有解释型语言的特征。因为 Java 程序要经过先编译，后解释两个步骤，由 Java 编写的程序需要先经过编译步骤，生成字节码（`.class` 文件），这种字节码必须由 Java 解释器来解释执行。

## Oracle JDK vs OpenJDK

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 JDK 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，以及一些闭源的第三方组件，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：**（提示：下面括号内的内容是基于原文补充说明的，因为原文太过于晦涩难懂，用人话重新解释了下，如果你看得懂里面的术语，可以忽略括号解释的内容）

1. Oracle JDK 大概每 6 个月发一次主要版本（从 2014 年 3 月 JDK 8 LTS 发布到 2017 年 9 月 JDK 9 发布经历了长达 3 年多的时间，所以并不总是 6 个月），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；（个人观点：众所周知，JDK 原来是 SUN 公司开发的，后来 SUN 公司又卖给了 Oracle 公司，Oracle 公司以 Oracle 数据库而著名，而 Oracle 数据库又是闭源的，这个时候 Oracle 公司就不想完全开源了，但是原来的 SUN 公司又把 JDK 给开源了，如果这个时候 Oracle 收购回来之后就把他给闭源，必然会引其很多 Java 开发者的不满，导致大家对 Java 失去信心，那 Oracle 公司收购回来不就把 Java 烂在手里了吗！然后，Oracle 公司就想了个骚操作，这样吧，我把一部分核心代码开源出来给你们玩，并且我要和你们自己搞的 JDK 区分下，你们叫 OpenJDK，我叫 Oracle JDK，我发布我的，你们继续玩你们的，要是你们搞出来什么好玩的东西，我后续发布 Oracle JDK 也会拿来用一下，一举两得！）OpenJDK 开源项目：<https://github.com/openjdk/jdk>
3. Oracle JDK 比 OpenJDK 更稳定（肯定啦，Oracle JDK 由 Oracle 内部团队进行单独研发的，而且发布时间比 OpenJDK 更长，质量更有保障）。OpenJDK 和 Oracle JDK 的代码几乎相同（OpenJDK 的代码是从 Oracle JDK 代码派生出来的，可以理解为在 Oracle JDK 分支上拉了一条新的分支叫 OpenJDK，所以大部分代码相同），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（如果是 LTS 长期支持版本的话也会，比如 JDK 8，但并不是每个版本都是 LTS 版本），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 使用 BCL/OTN 协议获得许可，而 OpenJDK 根据 GPL v2 许可获得许可。

既然 Oracle JDK 这么好，那为什么还要有 OpenJDK？

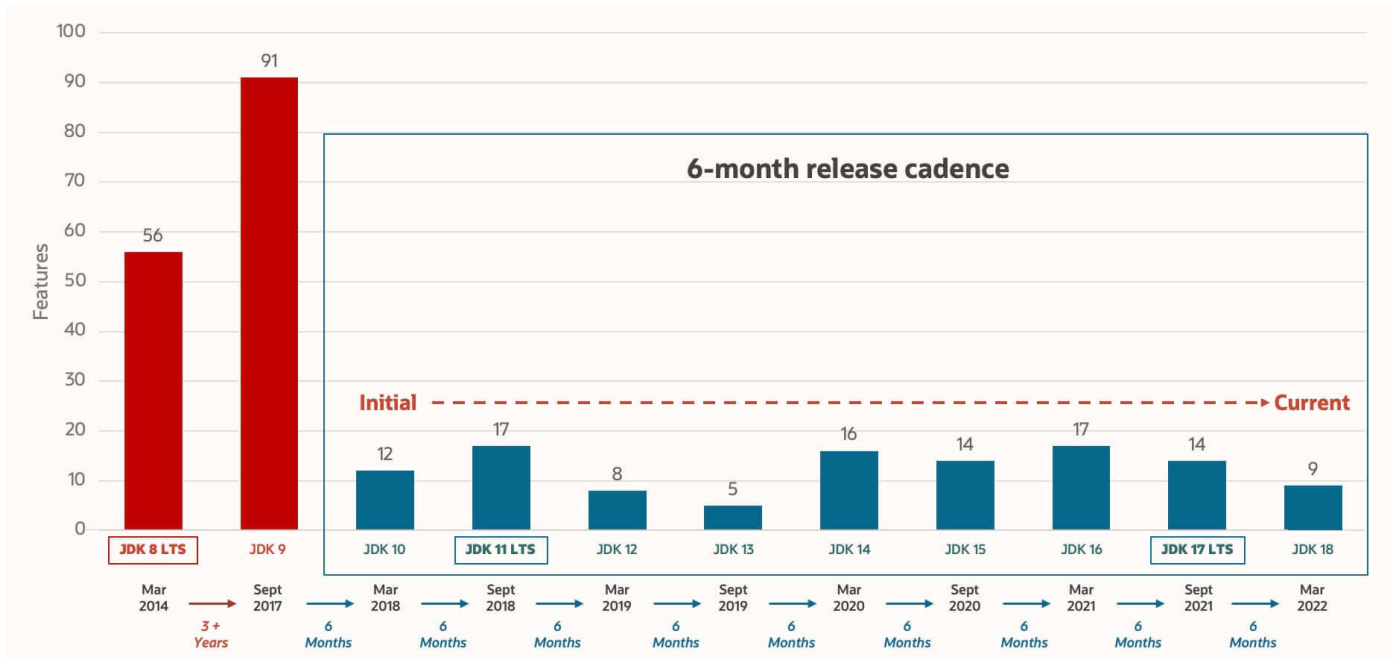
答：

1. OpenJDK 是开源的，开源意味着你可以对它根据你自己的需要进行修改、优化，比如 Alibaba 基于 OpenJDK 开发了 Dragonwell8：<https://github.com/alibaba/dragonwell8>
2. OpenJDK 是商业免费的（这也是为什么通过 yum 包管理器上默认安装的 JDK 是 OpenJDK 而不是 Oracle JDK）。虽然 Oracle JDK 也是商业免费（比如 JDK 8），但并不

是所有版本都是免费的。

3. OpenJDK 更新频率更快。Oracle JDK 一般是每 6 个月发布一个新版本，而 OpenJDK 一般是每 3 个月发布一个新版本。（现在你知道为啥 Oracle JDK 更稳定了吧，先在 OpenJDK 试试水，把大部分问题都解决掉了才在 Oracle JDK 上发布）

基于以上这些原因，OpenJDK 还是有存在的必要的！



拓展一下：

- BCL 协议（Oracle Binary Code License Agreement）：可以使用 JDK（支持商用），但是不能进行修改。
- OTN 协议（Oracle Technology Network License Agreement）：11 及之后新发布的 JDK 用的都是这个协议，可以自己私下用，但是商用需要付费。



Oracle JDK 各个版本所用的协议		
Oracle JDK 版本	BCL协议	OTN协议
6	最后一个公共更新6u45之前	
7	最后一个公共更新7u80之前	
8	8u201/8u202之前	8u211/8u212之后
9	✓	
10	✓	
11		✓
12		✓

相关阅读👍：[《Differences Between Oracle JDK and OpenJDK》](#)

## Java 语言关键字有哪些？

分类	关键字						
访问控制	private	protected	public				
类，方法和变量修饰符	abstract	class	extends	final	implements	interface	native
	new	static	strictfp	synchronized	transient	volatile	enum
程序控制	break	continue	return	do	while	if	else
	for	instanceof	switch	case	default	assert	
错误处理	try	catch	throw	throws	finally		
包相关	import	package					
基本类型	boolean	byte	char	double	float	int	long
	short						
变量引用	super	this	void				
保留字	goto	const					



Tips: 所有的关键字都是小写的, 在 IDE 中会以特殊颜色显示。

`default` 这个关键字很特殊, 既属于程序控制, 也属于类, 方法和变量修饰符, 还属于访问控制。

- 在程序控制中, 当在 `switch` 中匹配不到任何情况时, 可以使用 `default` 来编写默认匹配的情况。
- 在类, 方法和变量修饰符中, 从 JDK8 开始引入了默认方法, 可以使用 `default` 关键字来定义一个方法的默认实现。
- 在访问控制中, 如果一个方法前没有任何修饰符, 则默认会有一个修饰符 `default`, 但是这个修饰符加上了就会报错。

⚠ 注意: 虽然 `true`, `false`, 和 `null` 看起来像关键字但实际上他们是字面值, 同时你也不可以作为标识符来使用。

官方文档: [https://docs.oracle.com/javase/tutorial/java/nutsandbolts/\\_keywords.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html)

## 自增自减运算符

在写代码的过程中, 常见的一种情况是需要某个整数类型变量增加 1 或减少 1, Java 提供了一种特殊的运算符, 用于这种表达式, 叫做自增运算符 (`++`) 和自减运算符 (`--`)。

`++` 和 `--` 运算符可以放在变量之前, 也可以放在变量之后, 当运算符放在变量之前时(前缀), 先自增/减, 再赋值; 当运算符放在变量之后时(后缀), 先赋值, 再自增/减。例如, 当 `b = ++a` 时, 先自增 (自己增加 1), 再赋值 (赋值给 b); 当 `b = a++` 时, 先赋值 (赋值给 b), 再自增 (自己增加 1)。也就是, `++a` 输出的是 `a+1` 的值, `a++` 输出的是 `a` 值。用一句口诀就是: “符号在前就先加/减, 符号在后就后加/减”。

## 成员变量与局部变量的区别?

- **语法形式**: 从语法形式上看, 成员变量是属于类的, 而局部变量是在代码块或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。
- **存储方式**: 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。而对象存在于堆内存, 局部变量则存在于栈内存。
- **生存时间**: 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生成, 随着方法的调用结束而消亡。

- **默认值**：从变量是否有默认值来看，成员变量如果没有被赋初始值，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。

## 静态变量有什么作用？

静态变量可以被类的所有实例共享。无论一个类创建了多少个对象，它们都共享同一份静态变量。

通常情况下，静态变量会被 `final` 关键字修饰成为常量。

## 字符型常量和字符串常量的区别？

1. **形式**：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 0 个或若干个字符。
2. **含义**：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)

## 静态方法和实例方法有何不同？

### 1、调用方式

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.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**。

不过，需要注意的是一般不建议使用 `对象.方法名` 的方式来调用静态方法。这种方式非常容易造成混淆，静态方法不属于类的某个对象而是属于这个类。

因此，一般建议使用 `类名.方法名` 的方式来调用静态方法。

```
public class Person {  
    public void method() {  
        //.....  
    }  
  
    public static void staicMethod(){  
        //.....  
    }  
}
```

```
public static void main(String[] args) {  
    Person person = new Person();  
    // 调用实例方法  
    person.method();  
    // 调用静态方法  
    Person.staicMethod()  
}  
}
```

## 2、访问类成员是否存在限制

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），不允许访问实例成员（即实例成员变量和实例方法），而实例方法不存在这个限制。

## 重载和重写有什么区别？

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（或者父类和子类之间），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。

《Java 核心技术》这本书是这样介绍重载的：

如果多个方法(比如 `StringBuilder` 的构造方法)有相同的名字、不同的参数，便产生了重载。

```
StringBuilder sb = new StringBuilder();  
StringBuilder sb2 = new StringBuilder("HelloWorld");
```

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))。

Java 允许重载任何方法，而不只是构造器方法。

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. 方法名、参数列表必须相同，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，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。

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();  
    }  
}
```

## 什么是可变长参数？

从 Java5 开始，Java 支持定义可变长参数，所谓可变长参数就是允许在调用方法时传入不定长度的参数。就比如下面的这个 `printVariable` 方法就可以接受 0 个或者多个参数。

```
public static void method1(String... args) {  
    //.....  
}
```

另外，可变参数只能作为函数的最后一个参数，但其前面可以有也可以没有任何其他参数。

```
public static void method2(String arg1, String... args) {  
    //.....  
}
```

遇到方法重载的情况怎么办呢？会优先匹配固定参数还是可变参数的方法呢？

答案是会优先匹配固定参数的方法，因为固定参数的方法匹配度更高。

我们通过下面这个例子来证明一下。

```
/**  
 * 微信搜 JavaGuide 回复"面试突击"即可免费领取个人原创的 Java 面试手册  
 *  
 * @author Guide哥  
 * @date 2021/12/13 16:52  
 **/  
  
public class VariableLengthArgument {  
  
    public static void printVariable(String... args) {  
        for (String s : args) {  
            System.out.println(s);  
        }  
    }  
}
```

```

public static void printVariable(String arg1, String arg2) {
    System.out.println(arg1 + arg2);
}

public static void main(String[] args) {
    printVariable("a", "b");
    printVariable("a", "b", "c", "d");
}
}

```

输出：

```

ab
a
b
c
d

```

另外，Java 的可变参数编译后实际会被转换成一个数组，我们看编译后生成的 `class` 文件就可以看出来了。

```

public class VariableLengthArgument {

    public static void printVariable(String... args) {
        String[] var1 = args;
        int var2 = args.length;

        for(int var3 = 0; var3 < var2; ++var3) {
            String s = var1[var3];
            System.out.println(s);
        }

    }

    // .....
}

```



# Java 中的几种基本数据类型了解么？

Java 中有 8 种基本数据类型，分别为：

- 6 种数字类型：
  - 4 种整数型： byte 、 short 、 int 、 long
  - 2 种浮点型： float 、 double
- 1 种字符类型： char
- 1 种布尔型： boolean 。

这 8 种基本数据类型的默认值以及所占空间的大小如下：

基本类型	位数	字节	默认值	取值范围
byte	8	1	0	-128 ~ 127
short	16	2	0	-32768 ~ 32767
int	32	4	0	-2147483648 ~ 2147483647
long	64	8	0L	-9223372036854775808 ~ 9223372036854775807
char	16	2	'u0000'	0 ~ 65535
float	32	4	0f	1.4E-45 ~ 3.4028235E38
double	64	8	0d	4.9E-324 ~ 1.7976931348623157E308
boolean	1		false	true、false

对于 boolean ，官方文档未明确定义，它依赖于 JVM 厂商的具体实现。逻辑上理解是占用 1 位，但是实际中会考虑计算机高效存储因素。

另外，Java 的每种基本类型所占存储空间的大小不会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（《Java 编程思想》2.2 节有提到）。

注意：

1. Java 里使用 `long` 类型的数据一定要在数值后面加上 `L`，否则将作为整型解析。
2. `char a = 'h'` `char` :单引号， `String a = "hello"` :双引号。


这八种基本类型都有对应的包装类分别

为： `Byte` 、 `Short` 、 `Integer` 、 `Long` 、 `Float` 、 `Double` 、 `Character` 、 `Boolean` 。

## 基本类型和包装类型的区别？

- 成员变量包装类型不赋值就是 `null` ，而基本类型有默认值且不是 `null` 。
- 包装类型可用于泛型，而基本类型不可以。
- 基本数据类型的局部变量存放在 Java 虚拟机栈中的局部变量表中，基本数据类型的成员变量（未被 `static` 修饰）存放在 Java 虚拟机的堆中。包装类型属于对象类型，我们知道几乎所有对象实例都存在于堆中。
- 相比于对象类型，基本数据类型占用的空间非常小。

为什么说几乎是所有对象实例呢？这是因为 HotSpot 虚拟机引入了 JIT 优化之后，会对对象进行逃逸分析，如果发现某一个对象并没有逃逸到方法外部，那么就可能通过标量替换来实现栈上分配，而避免堆上分配内存

 注意：基本数据类型存放在栈中是一个常见的误区！基本数据类型的成员变量如果没有被 `static` 修饰的话（不建议这么使用，应该要使用基本数据类型对应的包装类型），就存放在堆中。

```
class BasicTypeVar{  
    private int x;  
}
```

## 包装类型的缓存机制了解么？

Java 基本数据类型的包装类型的大部分都用到了缓存机制来提升性能。

`Byte` ， `Short` ， `Integer` ， `Long` 这 4 种包装类默认创建了数值 `[-128, 127]` 的相应类型的缓存数据， `Character` 创建了数值在 `[0,127]` 范围的缓存数据， `Boolean` 直接返回 `True` or `False` 。

**Integer 缓存源码：**

```

public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}

private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static {
        // high value may be configured by property
        int h = 127;
    }
}

```

Character 缓存源码:

```

public static Character valueOf(char c) {
    if (c <= 127) { // must cache
        return CharacterCache.cache[(int)c];
    }
    return new Character(c);
}

private static class CharacterCache {
    private CharacterCache(){}
    static final Character cache[] = new Character[127 + 1];
    static {
        for (int i = 0; i < cache.length; i++)
            cache[i] = new Character((char)i);
    }
}

```

Boolean 缓存源码:

```
public static Boolean valueOf(boolean b) {  
    return (b ? TRUE : FALSE);  
}
```

如果超出对应范围仍然会去创建新的对象，缓存的范围区间的大小只是在性能和资源之间的权衡。

两种浮点数类型的包装类 `Float`，`Double` 并没有实现缓存机制。

```
Integer i1 = 33;  
Integer i2 = 33;  
System.out.println(i1 == i2); // 输出 true  
  
Float i11 = 333f;  
Float i22 = 333f;  
System.out.println(i11 == i22); // 输出 false  
  
Double i3 = 1.2;  
Double i4 = 1.2;  
System.out.println(i3 == i4); // 输出 false
```

下面我们来看一下问题。下面的代码的输出结果是 `true` 还是 `false` 呢？

```
Integer i1 = 40;  
Integer i2 = new Integer(40);  
System.out.println(i1==i2);
```

`Integer i1=40` 这一行代码会发生装箱，也就是说这行代码等价于 `Integer i1=Integer.valueOf(40)`。因此，`i1` 直接使用缓存中的对象。而 `Integer i2 = new Integer(40)` 会直接创建新的对象。

因此，答案是 `false`。你答对了吗？

记住：所有整型包装类对象之间值的比较，全部使用 `equals` 方法比较。

7. **【强制】**所有整型包装类对象之间值的比较，全部使用 equals 方法比较。

**说明：**对于 Integer var = ? 在-128 至 127 之间的赋值，Integer 对象是在 IntegerCache.cache 产生，会复用已有对象，这个区间内的 Integer 值可以直接使用 == 进行判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑，推荐使用 equals 方法进行判断。

## 自动装箱与拆箱了解吗？原理是什么？

什么是自动拆装箱？

- 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；

举例：

```
Integer i = 10;    //装箱
int n = i;         //拆箱
```

上面这两行代码对应的字节码为：

```
L1

LINENUMBER 8 L1

ALOAD 0

BIPUSH 10

INVOKESTATIC java/lang/Integer.valueOf (I)Ljava/lang/Integer;

PUTFIELD AutoBoxTest.i : Ljava/lang/Integer;

L2

LINENUMBER 9 L2
```

```

ALOAD 0

ALOAD 0

GETFIELD AutoBoxTest.i : Ljava/lang/Integer;

INVOKEVIRTUAL java/lang/Integer.intValue ()I

PUTFIELD AutoBoxTest.n : I

RETURN

```

从字节码中，我们发现装箱其实就是调用了包装类的 `valueOf()` 方法，拆箱其实就是调用了 `xxxValue()` 方法。

因此，

- `Integer i = 10` 等价于 `Integer i = Integer.valueOf(10)`
- `int n = i` 等价于 `int n = i.intValue()` ；

注意：如果频繁拆装箱的话，也会严重影响系统的性能。我们应该尽量避免不必要的拆装箱操作。

```

private static long sum() {
    // 应该使用 long 而不是 Long
    Long sum = 0L;
    for (long i = 0; i <= Integer.MAX_VALUE; i++)
        sum += i;
    return sum;
}

```

## 为什么浮点数运算的时候会有精度丢失的风险？

浮点数运算精度丢失代码演示：

```
float a = 2.0f - 1.9f;
float b = 1.8f - 1.7f;

System.out.println(a); // 0.100000024
System.out.println(b); // 0.0999999905
System.out.println(a == b); // false
```

为什么会出现这个问题呢？

这个和计算机保存浮点数的机制有很大关系。我们知道计算机是二进制的，而且计算机在表示一个数字时，宽度是有限的，无限循环的小数存储在计算机时，只能被截断，所以就会导致小数精度发生损失的情况。这也就是解释了为什么浮点数没有办法用二进制精确表示。

就比如说十进制下的 0.2 就没办法精确转换成二进制小数：

```
// 0.2 转换为二进制数的过程为，不断乘以 2，直到不存在小数为止，
// 在这个计算过程中，得到的整数部分从上到下排列就是二进制的结果。

0.2 * 2 = 0.4 -> 0
0.4 * 2 = 0.8 -> 0
0.8 * 2 = 1.6 -> 1
0.6 * 2 = 1.2 -> 1
0.2 * 2 = 0.4 -> 0 (发生循环)
...
```

关于浮点数的更多内容，建议看一下[计算机系统基础（四）浮点数](#)这篇文章。

## 如何解决浮点数运算的精度丢失问题？

`BigDecimal` 可以实现对浮点数的运算，不会造成精度丢失。通常情况下，大部分需要浮点数精确运算结果的业务场景（比如涉及到钱的场景）都是通过 `BigDecimal` 来做的。



```
BigDecimal a = new BigDecimal("1.0");
BigDecimal b = new BigDecimal("0.9");
BigDecimal c = new BigDecimal("0.8");

BigDecimal x = a.subtract(b);
BigDecimal y = b.subtract(c);

System.out.println(x); /* 0.1 */
System.out.println(y); /* 0.1 */
System.out.println(Objects.equals(x, y)); /* true */
```

关于 `BigDecimal` 的详细介绍，可以看看我写的这篇文章：[BigDecimal 详解](#)。

## 超过 long 整型的数据应该如何表示？

基本数值类型都有一个表达范围，如果超过这个范围就会有数值溢出的风险。

在 Java 中，64 位 long 整型是最大的整数类型。

```
long l = Long.MAX_VALUE;
System.out.println(l + 1); // -9223372036854775808
System.out.println(l + 1 == Long.MIN_VALUE); // true
```

`BigInteger` 内部使用 `int[]` 数组来存储任意大小的整形数据。

相对于常规整数类型的运算来说，`BigInteger` 运算的效率会相对较低。

## 如果一个类没有声明构造方法，该程序能正确执行吗？

如果一个类没有声明构造方法，也可以执行！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。如果我们自己添加了类的构造方法（无论是否有参），Java 就不会再添加默认的无参数的构造方法了，我们一直在不知不觉地使用构造方法，这也是为什么我们在创建对象的时候要加一个括号（因为要调用无参的构造方法）。如果我们重载了有参的构造方法，记得都要把无参的构造方法也写出来（无论是否用到），因为这可以帮助我们在创建对象的时候少踩坑。

## 构造方法有哪些特点？是否可被 override？

构造方法特点如下：

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

构造方法不能被 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。

## 面向对象三大特征

### 封装

封装是指把一个对象的状态信息（也就是属性）隐藏在对象内部，不允许外部对象直接访问对象的内部信息。但是可以提供一些可以被外界访问的方法来操作属性。就好像我们看不到挂在墙上的空调的内部的零件信息（也就是属性），但是可以通过遥控器（方法）来控制空调。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。就好像如果没有空调遥控器，那么我们就无法操控空调制冷，空调本身就没有意义了（当然现在还有很多其他方法，这里只是为了举例子）。

```
public class Student {  
    private int id;//id属性私有化  
    private String name;//name属性私有化  
  
    //获取id的方法  
    public int getId() {  
        return id;  
    }  
  
    //设置id的方法  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    //获取name的方法  
    public String getName() {  
        return name;  
    }  
}
```

```
    }

    //设置name的方法
    public void setName(String name) {
        this.name = name;
    }
}
```

## 继承

不同类型的对象，相互之间经常有一定数量的共同点。例如，小明同学、小红同学、小李同学，都共享学生的特性（班级、学号等）。同时，每一个对象还定义了额外的特性使得他们与众不同。例如小明的数学比较好，小红的性格惹人喜爱；小李的力气比较大。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，可以快速地创建新的类，可以提高代码的重用，程序的可维护性，节省大量创建新类的时间，提高我们的开发效率。

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）。

## 多态

多态，顾名思义，表示一个对象具有多种的状态，具体表现为父类的引用指向子类的实例。

多态的特点：

- 对象类型和引用类型之间具有继承（类）/实现（接口）的关系；
- 引用类型变量发出的方法调用的到底是哪个类中的方法，必须在程序运行期间才能确定；
- 多态不能调用“只在子类存在但在父类不存在”的方法；
- 如果子类重写了父类的方法，真正执行的是子类覆盖的方法，如果子类没有覆盖父类的方法，执行的是父类的方法。

## 接口和抽象类有什么共同点和区别？

共同点：

- 都不能被实例化。
- 都可以包含抽象方法。
- 都可以有默认实现的方法（Java 8 可以用 `default` 关键字在接口中定义默认方法）。

区别：

- 接口主要用于对类的行为进行约束，你实现了某个接口就具有了对应的行为。抽象类主要用于代码复用，强调的是所属关系。
- 一个类只能继承一个类，但是可以实现多个接口。
- 接口中的成员变量只能是 `public static final` 类型的，不能被修改且必须有初始值，而抽象类的成员变量默认 `default`，可在子类中被重新定义，也可被重新赋值。

## 深拷贝和浅拷贝区别了解吗？什么是引用拷贝？

关于深拷贝和浅拷贝区别，我这里先给结论：

- **浅拷贝**：浅拷贝会在堆上创建一个新的对象（区别于引用拷贝的一点），不过，如果原对象内部的属性是引用类型的话，浅拷贝会直接复制内部对象的引用地址，也就是说拷贝对象和原对象共用同一个内部对象。
- **深拷贝**：深拷贝会完全复制整个对象，包括这个对象所包含的内部对象。

上面的结论没有完全理解的话也没关系，我们来看一个具体的案例！

浅拷贝

浅拷贝的示例代码如下，我们这里实现了 `Cloneable` 接口，并重写了 `clone()` 方法。

`clone()` 方法的实现很简单，直接调用的是父类 `Object` 的 `clone()` 方法。

```
public class Address implements Cloneable{
    private String name;
    // 省略构造函数、Getter&Setter方法
    @Override
    public Address clone() {
        try {
```

```

        return (Address) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}

public class Person implements Cloneable {
    private Address address;
    // 省略构造函数、Getter&Setter方法
    @Override
    public Person clone() {
        try {
            Person person = (Person) super.clone();
            return person;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
}

```

测试：

```

Person person1 = new Person(new Address("武汉"));
Person person1Copy = person1.clone();
// true
System.out.println(person1.getAddress() == person1Copy.getAddress());

```

从输出结构就可以看出，`person1` 的克隆对象和 `person1` 使用的仍然是同一个 `Address` 对象。

## 深拷贝

这里我们简单对 `Person` 类的 `clone()` 方法进行修改，连带着要把 `Person` 对象内部的 `Address` 对象一起复制。

```

@Override
public Person clone() {
    try {
        Person person = (Person) super.clone();
        person.setAddress(person.getAddress().clone());
        return person;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}

```

测试：

```

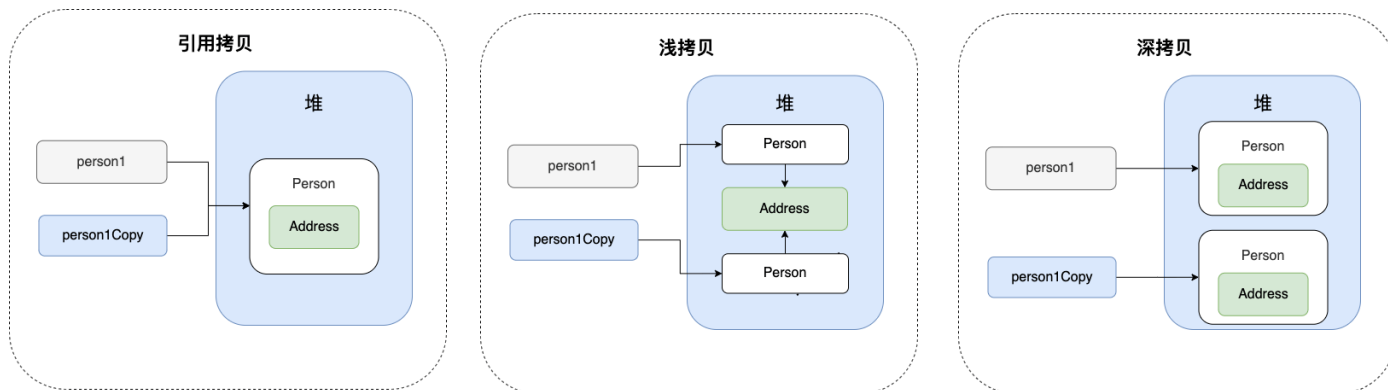
Person person1 = new Person(new Address("武汉"));
Person person1Copy = person1.clone();
// false
System.out.println(person1.getAddress() == person1Copy.getAddress());

```

从输出结构就可以看出，虽然 `person1` 的克隆对象和 `person1` 包含的 `Address` 对象已经是不同的了。

那什么是引用拷贝呢？简单来说，引用拷贝就是两个不同的引用指向同一个对象。

我专门画了一张图来描述浅拷贝、深拷贝、引用拷贝：



# Object

## Object 类的常见方法有哪些？

Object 类是一个特殊的类，是所有类的父类。它主要提供了以下 11 个方法：

```
/**
 * native 方法，用于返回当前运行时对象的 Class 对象，使用了 final 关键字修饰，故不允许子类
重写。
 */
public final native Class<?> getClass()

/**
 * native 方法，用于返回对象的哈希码，主要使用在哈希表中，比如 JDK 中的 HashMap。
 */
public native int hashCode()

/**
 * 用于比较 2 个对象的内存地址是否相等，String 类对该方法进行了重写以用于比较字符串的值是否相
等。
 */
public boolean equals(Object obj)

/**
 * native 方法，用于创建并返回当前对象的一份拷贝。
 */
protected native Object clone() throws CloneNotSupportedException

/**
 * 返回类的名字实例的哈希码的 16 进制的字符串。建议 Object 所有的子类都重写这个方法。
 */
public String toString()

/**
 * native 方法，并且不能重写。唤醒一个在此对象监视器上等待的线程(监视器相当于就是锁的概念)。
如果有多个线程在等待只会任意唤醒一个。
 */
public final native void notify()

/**
 * native 方法，并且不能重写。跟 notify 一样，唯一的区别就是会唤醒在此对象监视器上等待的所有
线程，而不是一个线程。
 */
public final native void notifyAll()

/**
```



```

    * native方法，并且不能重写。暂停线程的执行。注意：sleep 方法没有释放锁，而 wait 方法释放了
    锁，timeout 是等待时间。
    */

    public final native void wait(long timeout) throws InterruptedException
    /**
    * 多了 nanos 参数，这个参数表示额外时间（以毫微秒为单位，范围是 0-999999）。所以超时的时间
    还需要加上 nanos 毫秒。。
    */

    public final void wait(long timeout, int nanos) throws InterruptedException
    /**
    * 跟之前的2个wait方法一样，只不过该方法一直等待，没有超时时间这个概念
    */

    public final void wait() throws InterruptedException
    /**
    * 实例被垃圾回收器回收的时候触发的操作
    */

    protected void finalize() throws Throwable { }

```

## == 和 equals() 的区别

== 对于基本类型和引用类型的作用效果是不同的：

- 对于基本数据类型来说，== 比较的是值。
- 对于引用数据类型来说，== 比较的是对象的内存地址。

因为 Java 只有值传递，所以，对于 == 来说，不管是比较基本数据类型，还是引用数据类型的变量，其本质比较的都是值，只是引用类型变量存的值是对象的地址。

equals() 不能用于判断基本数据类型的变量，只能用来判断两个对象是否相等。equals() 方法存在于 Object 类中，而 Object 类是所有类的直接或间接父类，因此所有的类都有 equals() 方法。

Object 类 equals() 方法：

```

public boolean equals(Object obj) {
    return (this == obj);
}

```

`equals()` 方法存在两种使用情况：

- **类没有重写 `equals()` 方法**：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，使用的默认是 `Object` 类 `equals()` 方法。
- **类重写了 `equals()` 方法**：一般我们都重写 `equals()` 方法来比较两个对象中的属性是否相等；若它们的属性相等，则返回 `true`(即，认为这两个对象相等)。

举个例子（这里只是为了举例。实际上，你按照下面这种写法的话，像 IDEA 这种比较智能的 IDE 都会提示你将 `==` 换成 `equals()`）：

```
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
System.out.println(aa == bb); // true
System.out.println(a == b); // false
System.out.println(a.equals(b)); // true
System.out.println(42 == 42.0); // true
```

`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。

`String` 类 `equals()` 方法：

```
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
```

```

        char v2[] = anotherString.value;

        int i = 0;

        while (n-- != 0) {
            if (v1[i] != v2[i])
                return false;

            i++;
        }

        return true;
    }
}

return false;
}

```

## hashCode() 有什么用？

`hashCode()` 的作用是获取哈希码（`int` 整数），也称为散列码。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）

## 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` ？

下面这段内容摘自我的 Java 启蒙书《Head First Java》：

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。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。

其实，`hashCode()` 和 `equals()` 都是用于比较两个对象是否相等。

**那为什么 JDK 还要同时提供这两个方法呢？**

这是因为在一些容器（比如 `HashMap`、`HashSet`）中，有了 `hashCode()` 之后，判断元素是否在对应容器中的效率会更高（参考添加元素进 `HashSet` 的过程）！

我们在前面也提到了添加元素进 `HashSet` 的过程，如果 `HashSet` 在对比的时候，同样的 `hashCode` 有多个对象，它会继续使用 `equals()` 来判断是否真的相同。也就是说 `hashCode` 帮助我们大大缩小了查找成本。

**那为什么不只提供 `hashCode()` 方法呢？**

这是因为两个对象的 `hashCode` 值相等并不代表两个对象就相等。

**那为什么两个对象有相同的 `hashCode` 值，它们也不一定是相等的？**

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

总结下来就是：

- 如果两个对象的 `hashCode` 值相等，那这两个对象不一定相等（哈希碰撞）。
- 如果两个对象的 `hashCode` 值相等并且 `equals()` 方法也返回 `true`，我们才认为这两个对象相等。
- 如果两个对象的 `hashCode` 值不相等，我们就可以直接认为这两个对象不相等。

相信大家看了我前面对 `hashCode()` 和 `equals()` 的介绍之后，下面这个问题已经难不倒你们了。

**为什么重写 `equals()` 时必须重写 `hashCode()` 方法？**

因为两个相等的对象的 `hashCode` 值必须是相等。也就是说如果 `equals` 方法判断两个对象是相等的，那这两个对象的 `hashCode` 值也要相等。

如果重写 `equals()` 时没有重写 `hashCode()` 方法的话就可能会导致 `equals` 方法判断是相等的两个对象，`hashCode` 值却不相等。

思考：重写 `equals()` 时没有重写 `hashCode()` 方法的话，使用 `HashMap` 可能会出现什么问题。

总结：

- `equals` 方法判断两个对象是相等的，那这两个对象的 `hashCode` 值也要相等。
- 两个对象有相同的 `hashCode` 值，他们也不一定是相等的（哈希碰撞）。

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](#)

## String

### String、StringBuffer、StringBuilder 的区别？

可变性

`String` 是不可变的（后面会详细分析原因）。

`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，不过没有使用 `final` 和 `private` 关键字修饰，最关键的是这个 `AbstractStringBuilder` 类还提供了很多修改字符串的方法比如 `append` 方法。

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {
    char[] value;

    public AbstractStringBuilder append(String str) {
        if (str == null)
            return appendNull();

        int len = str.length();
        ensureCapacityInternal(count + len);
        str.getChars(0, len, value, count);
        count += len;
        return this;
    }

    //...
}
```

## 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`

## String 为什么是不可变的？

`String` 类中使用 `final` 关键字修饰字符数组来保存字符串，所以 `String` 对象是不可变的。

```
public final class String implements java.io.Serializable, Comparable<String>,
CharSequence {
    private final char value[];
    //...
}
```



修正：我们知道被 `final` 关键字修饰的类不能被继承，修饰的方法不能被重写，修饰的变量是基本数据类型则值不能改变，修饰的变量是引用类型则不能再指向其他对象。因此，`final` 关键字修饰的数组保存字符串并不是 `String` 不可变的根本原因，因为这个数组保存的字符串是可变的（`final` 修饰引用类型变量的情况）。

`String` 真正不可变有下面几点原因：

1. 保存字符串的数组被 `final` 修饰且为私有的，并且 `String` 类没有提供/暴露修改这个字符串的方法。
2. `String` 类被 `final` 修饰导致其不能被继承，进而避免了子类破坏 `String` 不可变。

相关阅读：[如何理解 String 类型值的不可变？ - 知乎提问](#)

补充（来自[issue 675](#)）：在 Java 9 之后，`String`、`StringBuilder` 与 `StringBuffer` 的实现改用 `byte` 数组存储字符串。

```
public final class String implements
    java.io.Serializable, Comparable<String>, CharSequence {
    // @Stable 注解表示变量最多被修改一次，称为“稳定的”。
    @Stable
    private final byte[] value;
}

abstract class AbstractStringBuilder implements Appendable, CharSequence {
    byte[] value;
}
```

**Java 9 为何要将 `String` 的底层实现由 `char[]` 改成了 `byte[]` ？**

新版的 `String` 其实支持两个编码方案：`Latin-1` 和 `UTF-16`。如果字符串中包含的汉字没有超过 `Latin-1` 可表示范围内的字符，那就会使用 `Latin-1` 作为编码方案。`Latin-1` 编码方案下，`byte` 占一个字节(8 位)，`char` 占用 2 个字节（16），`byte` 相较 `char` 节省一半的内存空间。

JDK 官方就说了绝大部分字符串对象只包含 `Latin-1` 可表示的字符。

### Motivation

The current implementation of the `String` class stores characters in a `char` array, using two bytes (sixteen bits) for each character. Data gathered from many different applications indicates that strings are a major component of heap usage and, moreover, that most `String` objects contain only `Latin-1` characters. Such characters require only one byte of storage, hence half of the space in the internal `char` arrays of such `String` objects is going unused.



如果字符串中包含的汉字超过 Latin-1 可表示范围内的字符，byte 和 char 所占用的空间是一样的。

这是官方的介绍：<https://openjdk.java.net/jeps/254>。

## 字符串拼接用“+” 还是 StringBuilder?

Java 语言本身并不支持运算符重载，“+”和“+=”是专门为 String 类重载过的运算符，也是 Java 中仅有的两个重载过的运算符。

```
String str1 = "he";
String str2 = "llo";
String str3 = "world";
String str4 = str1 + str2 + str3;
```

上面的代码对应的字节码如下：

```
1  0  ldc  #2  <he>
2  2  astore_1
3  3  ldc  #3  <llo>
4  5  astore_2
5  6  ldc  #4  <world>
6  8  astore_3
7  9  new  #5  <java/lang/StringBuilder>
8 12  dup
9 13  invokespecial  #6  <java/lang/StringBuilder.<init> : ()V>
10 16  aload_1
11 17  invokevirtual  #7  <java/lang/StringBuilder.append : (Ljava/lang/String;)Ljava/lang/StringBuilder;>
12 20  aload_2
13 21  invokevirtual  #7  <java/lang/StringBuilder.append : (Ljava/lang/String;)Ljava/lang/StringBuilder;>
14 24  aload_3
15 25  invokevirtual  #7  <java/lang/StringBuilder.append : (Ljava/lang/String;)Ljava/lang/StringBuilder;>
16 28  invokevirtual  #8  <java/lang/StringBuilder.toString : ()Ljava/lang/String;>
17 31  astore 4
18 33  return
```

创建 StringBuilder

调用 StringBuilder 的 append 方法

调用 StringBuilder 的 toString 方法

可以看出，字符串对象通过“+”的字符串拼接方式，实际上是通过 `StringBuilder` 调用 `append()` 方法实现的，拼接完成之后调用 `toString()` 得到一个 `String` 对象。

不过，在循环内使用“+”进行字符串的拼接的话，存在比较明显的缺陷：编译器不会创建单个 `StringBuilder` 以复用，会导致创建过多的 `StringBuilder` 对象。

```
String[] arr = {"he", "llo", "world"};

String s = "";

for (int i = 0; i < arr.length; i++) {
    s += arr[i];
}

System.out.println(s);
```

StringBuilder 对象是在循环内部被创建的，这意味着每循环一次就会创建一个 StringBuilder 对象。

```
20 23 astore_3
21 26 arraylength
22 27 istore 4
23 29 iconst_0
24 30 istore 5
25 32 iload 5
26 34 iload 4
27 36 if_icmpge 71 (+35)
28 39 aload_3
29 40 iload 5
30 42 aaload
31 43 astore 6
32 45 new #7 <java/lang/StringBuilder>
33 48 dup
34 49 invokespecial #8 <java/lang/StringBuilder.<init> : ()V>
35 52 aload_2
36 53 invokevirtual #9 <java/lang/StringBuilder.append : (Ljava/lang/String;)Ljava/lang/StringBuilder;>
37 56 aload 6
38 58 invokevirtual #9 <java/lang/StringBuilder.append : (Ljava/lang/String;)Ljava/lang/StringBuilder;>
39 61 invokevirtual #10 <java/lang/StringBuilder.toString : ()Ljava/lang/String;>
40 64 astore_2
41 65 iinc 5 by 1
42 68 goto 32 (-36)
43 71 getstatic #11 <java/lang/System.out : Ljava/io/PrintStream;>
44 74 aload_2
45 75 invokevirtual #12 <java/io/PrintStream.println : (Ljava/lang/String;)V>
46 78 return
```

循环内部创建 StringBuilder 对象

如果直接使用 StringBuilder 对象进行字符串拼接的话，就不会存在这个问题了。

```
String[] arr = {"he", "llo", "world"};

StringBuilder s = new StringBuilder();

for (String value : arr) {
    s.append(value);
}

System.out.println(s);
```

```

16 20 new #6 <java/lang/StringBuilder>
17 23 dup
18 24 invokespecial #7 <java/lang/StringBuilder.<init> : ()V>
19 27 astore_2
20 28 aload_1
21 29 astore_3
22 30 aload_3
23 31 arraylength
24 32 istore 4
25 34 iconst_0
26 35 istore 5
27 37 iload 5
28 39 iload 4
29 41 if_icmpge 63 (+22)
30 44 aload_3
31 45 iload 5
32 47 aaload
33 48 astore 6
34 50 aload_2
35 51 aload 6
36 53 invokevirtual #8 <java/lang/StringBuilder.append : (Ljava/lang/String;)Ljava/lang/StringBuilder;>
37 56 pop
38 57 iinc 5 by 1
39 60 goto 37 (-23)

```

如果你使用 IDEA 的话，IDEA 自带的代码检查机制也会提示你修改代码。

## String#equals() 和 Object#equals() 有何区别？

String 中的 equals 方法是被重写过的，比较的是 String 字符串的值是否相等。Object 的 equals 方法是比较的对象的内存地址。

## 字符串常量池的作用了解吗？

字符串常量池 是 JVM 为了提升性能和减少内存消耗针对字符串（String 类）专门开辟的一块区域，主要目的是为了减少字符串的重复创建。

```

// 在堆中创建字符串对象"ab"
// 将字符串对象"ab"的引用保存在字符串常量池中
String aa = "ab";
// 直接返回字符串常量池中字符串对象"ab"的引用
String bb = "ab";

System.out.println(aa==bb); // true

```

更多关于字符串常量池的介绍可以看一下 [Java 内存区域详解](#) 这篇文章。

## String s1 = new String("abc");这句话创建了几个字符串对象？

会创建 1 或 2 个字符串对象。

1、如果字符串常量池中不存在字符串对象“abc”的引用，那么会在堆中创建 2 个字符串对象“abc”。

示例代码（JDK 1.8）：

```
String s1 = new String("abc");
```

对应的字节码：

```
1 0 new #2 <java/lang/String>      → 堆中创建一个 String 对象，此时未被初始化
2 3 dup
3 4 ldc #3 <abc>                    → 堆中创建字符串对象“abc”并在字符串常量池中保存对应的引用
4 6 invokespecial #4 <java/lang/String.<init> : (Ljava/lang/String;)V
5 9 astore_1
6 10 return
```

↓  
调用构造方法对第 0 步创建的 String 对象赋值

ldc 命令用于判断字符串常量池中是否保存了对应的字符串对象的引用，如果保存了的话直接返回，如果没有保存的话，会在堆中创建对应的字符串对象并将该字符串对象的引用保存到字符串常量池中。

2、如果字符串常量池中已存在字符串对象“abc”的引用，则只会在堆中创建 1 个字符串对象“abc”。

示例代码（JDK 1.8）：

```
// 字符串常量池中已存在字符串对象“abc”的引用
String s1 = "abc";
// 下面这段代码只会在堆中创建 1 个字符串对象“abc”
String s2 = new String("abc");
```

对应的字节码：

```

1 0 ldc #2 <abc>
2 2 astore_1
3 3 new #3 <java/lang/String>
4 6 dup
5 7 ldc #2 <abc>
6 9 invokespecial #4 <java/lang/String.<init> : (Ljava/lang/String;)V>
7 12 astore_2
8 13 return

```

这里就不对上面的字节码进行详细注释了，7 这个位置的 `ldc` 命令不会在堆中创建新的字符串对象“abc”，这是因为 0 这个位置已经执行了一次 `ldc` 命令，已经在堆中创建过一次字符串对象“abc”了。7 这个位置执行 `ldc` 命令会直接返回字符串常量池中字符串对象“abc”对应的引用。

## intern 方法有什么作用？

`String.intern()` 是一个 native（本地）方法，其作用是将指定的字符串对象的引用保存在字符串常量池中，可以简单分为两种情况：

- 如果字符串常量池中保存了对应的字符串对象的引用，就直接返回该引用。
- 如果字符串常量池中没有保存了对应的字符串对象的引用，那就在常量池中创建一个指向该字符串对象的引用并返回。

示例代码（JDK 1.8）：

```

// 在堆中创建字符串对象"Java"
// 将字符串对象"Java"的引用保存在字符串常量池中
String s1 = "Java";
// 直接返回字符串常量池中字符串对象"Java"对应的引用
String s2 = s1.intern();
// 会在堆中在单独创建一个字符串对象
String s3 = new String("Java");
// 直接返回字符串常量池中字符串对象"Java"对应的引用
String s4 = s3.intern();
// s1 和 s2 指向的是堆中的同一个对象
System.out.println(s1 == s2); // true
// s3 和 s4 指向的是堆中不同的对象
System.out.println(s3 == s4); // false
// s1 和 s4 指向的是堆中的同一个对象
System.out.println(s1 == s4); //true

```

## String 类型的变量和常量做“+”运算时发生了什么？

先来看字符串不加 `final` 关键字拼接的情况（JDK1.8）：

```
String str1 = "str";
String str2 = "ing";
String str3 = "str" + "ing";
String str4 = str1 + str2;
String str5 = "string";
System.out.println(str3 == str4); // false
System.out.println(str3 == str5); // true
System.out.println(str4 == str5); // false
```

**注意：** 比较 String 字符串的值是否相等，可以使用 `equals()` 方法。String 中的 `equals` 方法是被重写过的。Object 的 `equals` 方法是比较的对象的内存地址，而 String 的 `equals` 方法比较的是字符串的值是否相等。如果你使用 `==` 比较两个字符串是否相等的话，IDEA 还是提示你使用 `equals()` 方法替换。



```
String str1 = "str";
String str2 = "ing";

String str3 = "str" + "ing"; // 常量池中的对象
String str4 = str1 + str2; // 在堆上创建的新的对象
String str5 = "string"; // 常量池中的对象
System.out.println(str3 == str4); // false
System.out.println(str3 == str5); // true
System.out.println(str4 == str5); // false
}
```

The screenshot shows an IDE with a code snippet. A red box highlights the suggestion "Replace '==' with 'equals()'" in the context menu for the `str3 == str4` comparison. Other suggestions include "Replace '==' with null-safe 'equals()'", "Flip '=='", and "Negate '==' to '!='".

对于编译期可以确定值的字符串，也就是常量字符串，jvm 会将其存入字符串常量池。并且，字符串常量拼接得到的字符串常量在编译阶段就已经被存放字符串常量池，这个得益于编译器的优化。

在编译过程中，Javac 编译器（下文中统称为编译器）会进行一个叫做 **常量折叠(Constant Folding)** 的代码优化。《深入理解 Java 虚拟机》中是也有介绍到：

```
int d = a + c;
int d = b + c;
char d = a + c;
```

后续代码中如果出现了如上3种赋值运算的话，那它们都能构成结构正确的抽象语法树，但是只有第一种写法在语义上是没有错误的，能够通过检查和编译。其余两种在Java语言中是不合逻辑的，无法编译（是否合乎语义逻辑必须限定在具体的语言与具体的上下文环境之中才有意义。如在C语言中，a、b、c的上下文定义不变，第二、三种写法都是可以正确编译的）。我们编码时经常能在IDE中看到由红线标注的错误提示，其中绝大部分都是来源于语义分析阶段的检查结果。

### 1. 标注检查

JavaC在编译过程中，语义分析过程可分为标注检查和数据及控制流分析两个步骤，分别由图10-5的attribute()和flow()方法（分别对应图10-5中的过程3.1和过程3.2）完成。

标注检查步骤要检查的内容包括诸如变量使用前是否已被声明、变量与赋值之间的数据类型是否能够匹配，等等，刚才3个变量定义的例子就属于标注检查的处理范畴。在标注检查中，还会顺便进行一个称为常量折叠（Constant Folding）的代码优化，这是JavaC编译器会对源代码做的极少量优化措施之一（代码优化几乎都在即时编译器中进行）。如果我们在Java代码中写下如下所示的变量定义：

常量折叠会把常量表达式的值求出来作为常量嵌在最终生成的代码中，这是JavaC编译器会对源代码做的极少量优化措施之一（代码优化几乎都在即时编译器中进行）。

对于 `String str3 = "str" + "ing";` 编译器会给你优化成 `String str3 = "string";`。

并不是所有的常量都会进行折叠，只有编译器在程序编译期就可以确定值的常量才可以：

- 基本数据类型( `byte` 、 `boolean` 、 `short` 、 `char` 、 `int` 、 `float` 、 `long` 、 `double` )以及字符串常量。
- `final` 修饰的基本数据类型和字符串变量
- 字符串通过“+”拼接得到的字符串、基本数据类型之间算数运算（加减乘除）、基本数据类型的位运算（<<、>>、>>>）

引用的值在程序编译期是无法确定的，编译器无法对其进行优化。

对象引用和“+”的字符串拼接方式，实际上是通过 `StringBuilder` 调用 `append()` 方法实现的，拼接完成之后调用 `toString()` 得到一个 `String` 对象。

```
String str4 = new StringBuilder().append(str1).append(str2).toString();
```

我们在平时写代码的时候，尽量避免多个字符串对象拼接，因为这样会重新创建对象。如果需要改变字符串的话，可以使用 `StringBuilder` 或者 `StringBuffer`。

不过，字符串使用 `final` 关键字声明之后，可以让编译器当做常量来处理。

示例代码：



```
final String str1 = "str";
final String str2 = "ing";
// 下面两个表达式其实是等价的
String c = "str" + "ing";// 常量池中的对象
String d = str1 + str2; // 常量池中的对象
System.out.println(c == d);// true
```

被 `final` 关键字修改之后的 `String` 会被编译器当做常量来处理，编译器在程序编译期就可以确定它的值，其效果就相当于访问常量。

如果，编译器在运行时才能知道其确切值的话，就无法对其优化。

示例代码（`str2` 在运行时才能确定其值）：

```
final String str1 = "str";
final String str2 = getStr();
String c = "str" + "ing";// 常量池中的对象
String d = str1 + str2; // 在堆上创建的新的对象
System.out.println(c == d);// false
public static String getStr() {
    return "ing";
}
```

## Exception 和 Error 有什么区别？

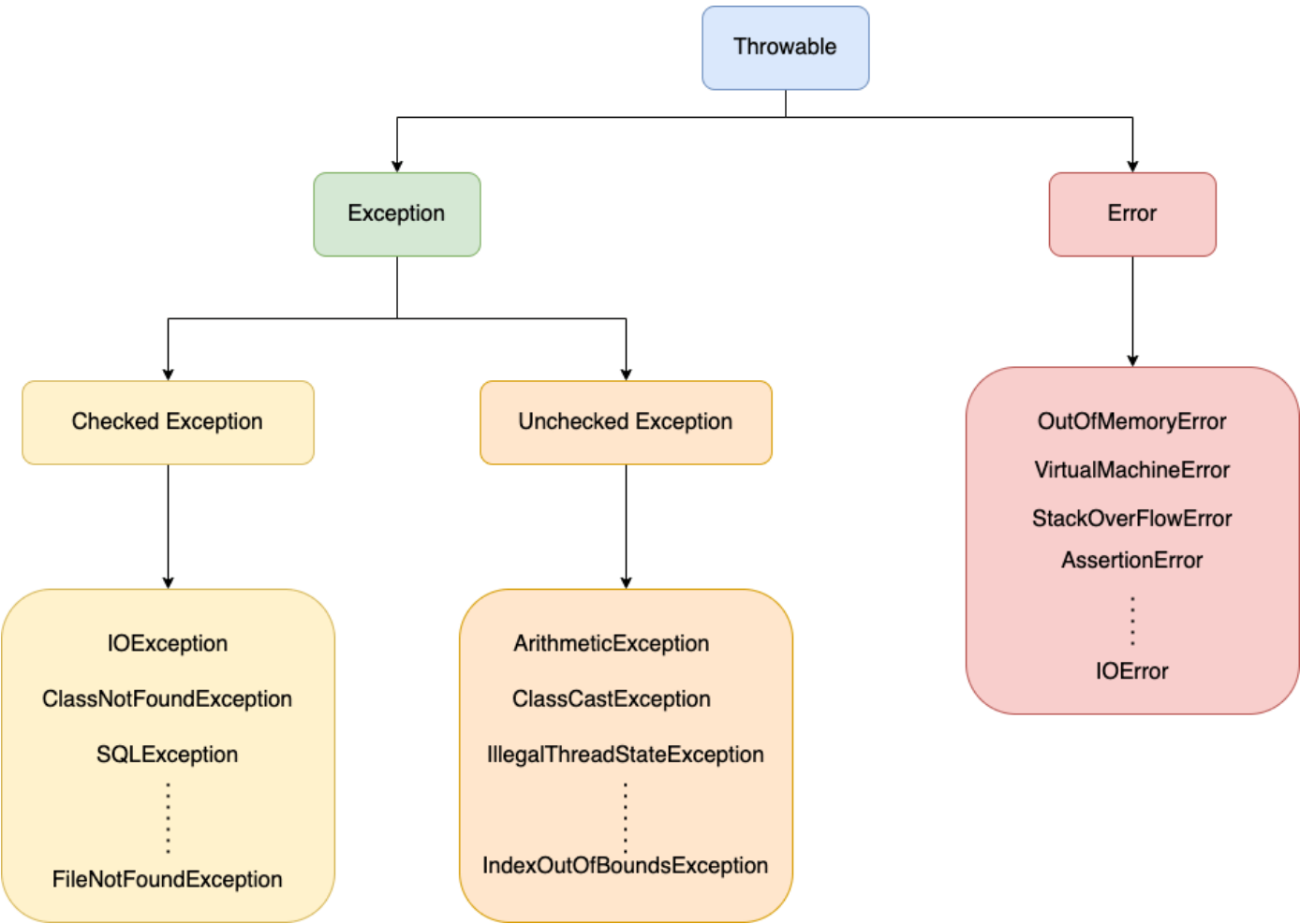
在 Java 中，所有的异常都有一个共同的祖先 `java.lang` 包中的 `Throwable` 类。`Throwable` 类有两个重要的子类：

- **Exception** :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 Checked Exception (受检查异常，必须处理) 和 Unchecked Exception (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` 来进行捕获不建议通过 `catch` 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。



# Checked Exception 和 Unchecked Exception 有什么区别？

Java 异常类层次结构图概览：



**Checked Exception** 即 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` 或者 `throws` 关键字处理的话，就没办法通过编译。

比如下面这段 IO 操作的代码：

```
String fileName = "file does not exist";
File file = new File(fileName);
FileInputStream stream = new FileInputStream(file);
}
```

Unhandled exception: java.io.FileNotFoundException  
Add exception to method signature More actions...

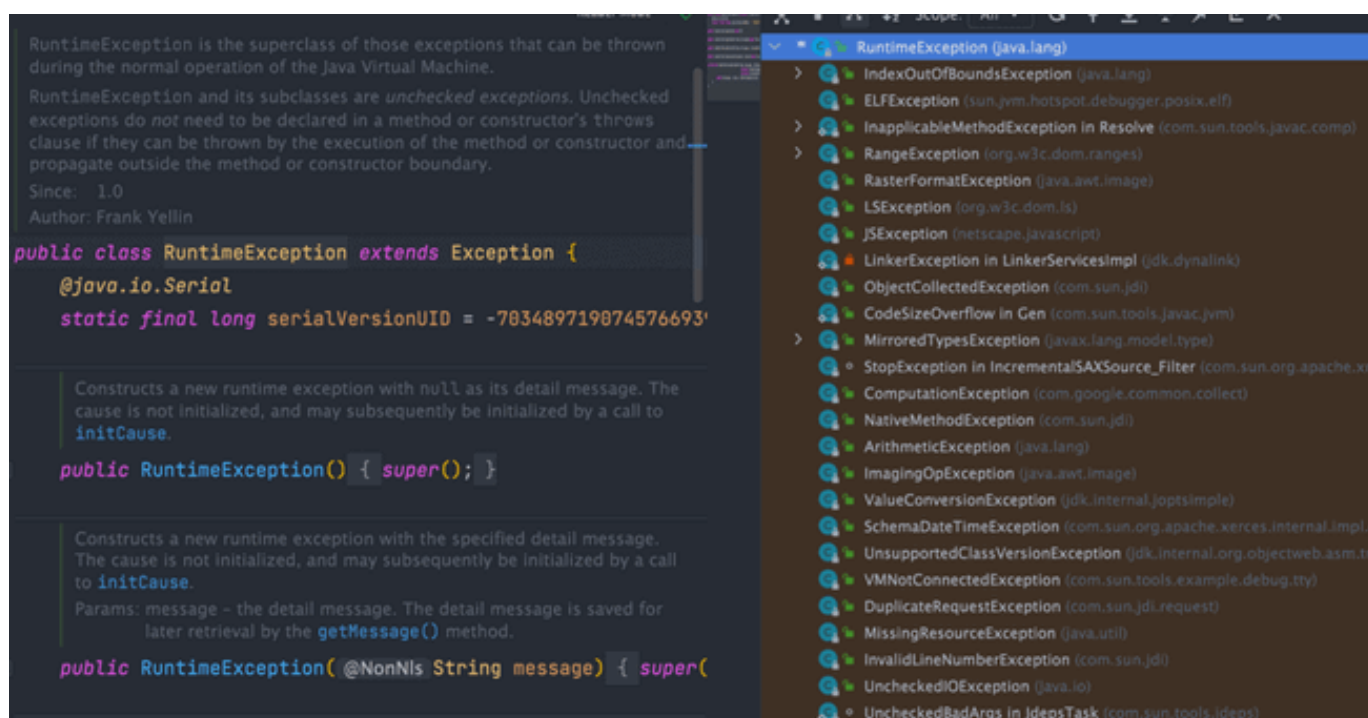
```
java.io.FileInputStream
public FileInputStream(@NotNull java.io.File file)
throws java.io.FileNotFoundException
Creates a FileInputStream by opening a connection to a
```

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

**Unchecked Exception** 即 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，常见的有（建议记下来，日常开发中会经常用到）：

- `NullPointerException`（空指针错误）
- `IllegalArgumentException`（参数错误比如方法入参类型错误）
- `NumberFormatException`（字符串转换为数字格式错误，`IllegalArgumentException` 的子类）
- `ArrayIndexOutOfBoundsException`（数组越界错误）
- `ClassCastException`（类型转换错误）
- `ArithmeticException`（算术错误）
- `SecurityException`（安全错误比如权限不够）
- `UnsupportedOperationException`（不支持的操作错误比如重复创建同一用户）
- .....



## Throwable 类常用方法有哪些？

- `String getMessage()`：返回异常发生时的简要描述
- `String toString()`：返回异常发生时的详细信息
- `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
- `void printStackTrace()`：在控制台上打印 `Throwable` 对象封装的异常信息

## try-catch-finally 如何使用？

- `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。

代码示例：

```
try {
    System.out.println("Try to do something");
    throw new RuntimeException("RuntimeException");
} catch (Exception e) {
    System.out.println("Catch Exception -> " + e.getMessage());
} finally {
    System.out.println("Finally");
}
```

输出：

```
Try to do something
Catch Exception -> RuntimeException
Finally
```

**注意：不要在 `finally` 语句块中使用 `return`!** 当 `try` 语句和 `finally` 语句中都有 `return` 语句时，`try` 语句块中的 `return` 语句会被忽略。这是因为 `try` 语句中的 `return` 返回值会先被暂存在一个本地变量中，当执行到 `finally` 语句中的 `return` 之后，这个本地变量的值就变为了 `finally` 语句中的 `return` 返回值。

[jvm 官方文档](#)中有明确提到：

If the `try` clause executes a *return*, the compiled code does the following:

1. Saves the return value (if any) in a local variable.
2. Executes a *jsr* to the code for the `finally` clause.
3. Upon return from the `finally` clause, returns the value saved in the local variable.

代码示例：

```
public static void main(String[] args) {  
    System.out.println(f(2));  
}  
  
public static int f(int value) {  
    try {  
        return value * value;  
    } finally {  
        if (value == 2) {  
            return 0;  
        }  
    }  
}
```

输出：

0

## finally 中的代码一定会执行吗？

不一定的！在某些情况下，finally 中的代码不会被执行。

就比如说 finally 之前虚拟机被终止运行的话，finally 中的代码就不会被执行。

```
try {
    System.out.println("Try to do something");
    throw new RuntimeException("RuntimeException");
} catch (Exception e) {
    System.out.println("Catch Exception -> " + e.getMessage());
    // 终止当前正在运行的Java虚拟机
    System.exit(1);
} finally {
    System.out.println("Finally");
}
```


输出：

```
Try to do something
Catch Exception -> RuntimeException
```

另外，在以下 2 种特殊情况下，`finally` 块的代码也不会被执行：

1. 程序所在的线程死亡。
2. 关闭 CPU。

相关 issue：<https://github.com/Snailclimb/JavaGuide/issues/190>。

 进阶一下：从字节码角度分析 `try catch finally` 这个语法糖背后的实现原理。

## 如何使用 `try-with-resources` 代替 `try-catch-finally` ?

1. 适用范围（资源的定义）：任何实现 `java.lang.AutoCloseable` 或者 `java.io.Closeable` 的对象
2. 关闭资源和 `finally` 块的执行顺序：在 `try-with-resources` 语句中，任何 `catch` 或 `finally` 块在声明的资源关闭后运行

《Effective Java》中明确指出：

面对必须要关闭的资源，我们总是应该优先使用 `try-with-resources` 而不是 `try-finally`。随之产生的代码更简短，更清晰，产生的异常对我们也更有用。`try-with-resources` 语句让我们更容易编写必须要关闭的资源的代码，若采用 `try-finally` 则几乎做不到这点。

Java 中类似于 `InputStream`、`OutputStream`、`Scanner`、`PrintWriter` 等的资源都需要我们调用 `close()` 方法来手动关闭，一般情况下我们都是通过 `try-catch-finally` 语句来实现这个需求，如下：

```
//读取文本文件的内容
Scanner scanner = null;
try {
    scanner = new Scanner(new File("D://read.txt"));
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} finally {
    if (scanner != null) {
        scanner.close();
    }
}
```

使用 Java 7 之后的 `try-with-resources` 语句改造上面的代码：

```
try (Scanner scanner = new Scanner(new File("test.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException fnfe) {
    fnfe.printStackTrace();
}
```

当然多个资源需要关闭的时候，使用 `try-with-resources` 实现起来也非常简单，如果你还是用 `try-catch-finally` 可能会带来很多问题。

通过使用分号分隔，可以在 `try-with-resources` 块中声明多个资源。

```
try (BufferedInputStream bin = new BufferedInputStream(new FileInputStream(new
File("test.txt")));
    BufferedOutputStream bout = new BufferedOutputStream(new
FileOutputStream(new File("out.txt")))) {
    int b;
    while ((b = bin.read()) != -1) {
        bout.write(b);
    }
}
catch (IOException e) {
    e.printStackTrace();
}
```

## 异常使用有哪些需要注意的地方？

- 不要把异常定义为静态变量，因为这样会导致异常栈信息错乱。每次手动抛出异常，我们都需要手动 `new` 一个异常对象抛出。
- 抛出的异常信息一定要有意义。
- 建议抛出更加具体的异常比如字符串转换为数字格式错误的时候应该抛出 `NumberFormatException` 而不是其父类 `IllegalArgumentException` 。
- 使用日志打印异常之后就不要再抛出异常了（两者不要同时存在一段代码逻辑中）。
- .....

## 何谓反射？

如果说大家研究过框架的底层原理或者咱们自己写过框架的话，一定对反射这个概念不陌生。反射之所以被称为框架的灵魂，主要是因为它赋予了我们在运行时分析类以及执行类中方法的能力。通过反射你可以获取任意一个类的所有属性和方法，你还可以调用这些方法和属性。

## 反射的优缺点？

反射可以让我们的代码更加灵活、为各种框架提供开箱即用的功能提供了便利。

不过，反射让我们在运行时有了分析操作类的能力的同时，也增加了安全问题，比如可以无视泛型参数的安全检查（泛型参数的安全检查发生在编译时）。另外，反射的性能也要稍差点，不过，对于框架来说实际是影响不大的。

相关阅读: [Java Reflection: Why is it so slow?](#) 。

## 反射的应用场景?

像咱们平时大部分时候都是在写业务代码, 很少会接触到直接使用反射机制的场景。但是! 这并不代表反射没有用。相反, 正是因为反射, 你才能这么轻松地使用各种框架。像 Spring/Spring Boot、MyBatis 等等框架中都大量使用了反射机制。

这些框架中也大量使用了动态代理, 而动态代理的实现也依赖反射。

比如下面是通过 JDK 实现动态代理的示例代码, 其中就使用了反射类 `Method` 来调用指定的方法。

```
public class DebugInvocationHandler implements InvocationHandler {  
    /**  
     * 代理类中的真实对象  
     */  
    private final Object target;  
  
    public DebugInvocationHandler(Object target) {  
        this.target = target;  
    }  
  
    public Object invoke(Object proxy, Method method, Object[] args) throws  
    InvocationTargetException, IllegalAccessException {  
        System.out.println("before method " + method.getName());  
        Object result = method.invoke(target, args);  
        System.out.println("after method " + method.getName());  
        return result;  
    }  
}
```

另外, 像 Java 中的一大利器 **注解** 的实现也用到了反射。

为什么你使用 Spring 的时候, 一个 `@Component` 注解就声明了一个类为 Spring Bean 呢? 为什么你通过一个 `@Value` 注解就读取到配置文件中的值呢? 究竟是怎么起作用的呢?



这些都是因为你可以基于反射分析类，然后获取到类/属性/方法/方法的参数上的注解。你获取到注解之后，就可以做进一步的处理。

## 何谓 SPI?

SPI 即 Service Provider Interface，字面意思就是：“服务提供者的接口”，我的理解是：专门提供给服务提供者或者扩展框架功能的开发者去使用的一个接口。

SPI 将服务接口和具体的服务实现分离开来，将服务调用方和服务实现者解耦，能够提升程序的扩展性、可维护性。修改或者替换服务实现并不需要修改调用方。

很多框架都使用了 Java 的 SPI 机制，比如：Spring 框架、数据库加载驱动、日志接口、以及 Dubbo 的扩展实现等等。

## SPI 扩展实现



协议扩展

调用拦截扩展

引用监听扩展

暴露监听扩展

集群扩展

路由扩展

负载均衡扩展

合并结果扩展

注册中心扩展

监控中心扩展

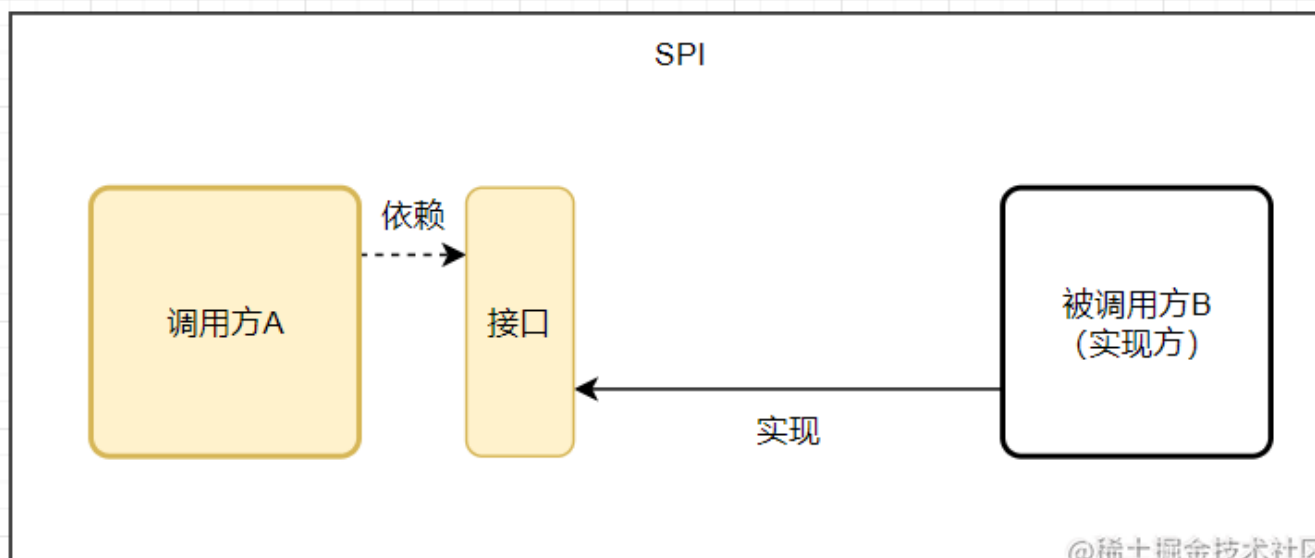
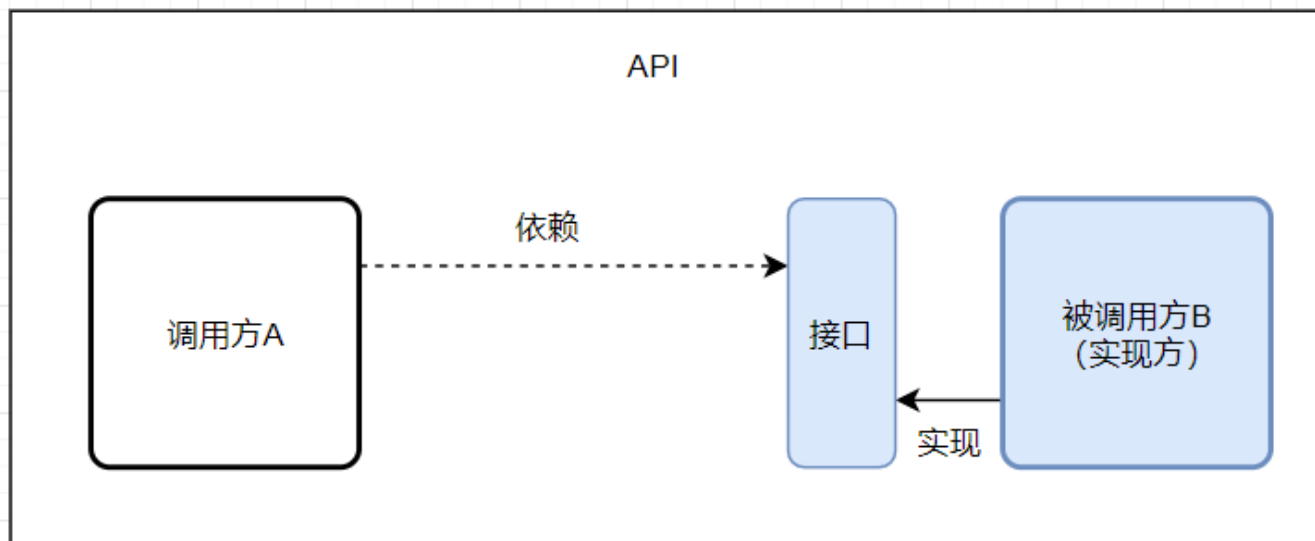
扩展点加载扩展

动态代理扩展

## SPI 和 API 有什么区别？

那 SPI 和 API 有啥区别？

说到 SPI 就不得不说一下 API 了，从广义上来说它们都属于接口，而且很容易混淆。下面先用一张图说明一下：



@稀土掘金技术社区

一般模块之间都是通过通过接口进行通讯，那我们在服务调用方和服务实现方（也称服务提供者）之间引入一个“接口”。

当实现方提供了接口和实现，我们可以通过调用实现方的接口从而拥有实现方给我们提供的的能力，这就是 API，这种接口和实现都是放在实现方的。

当接口存在于调用方这边时，就是 SPI，由接口调用方确定接口规则，然后由不同的厂商去根绝这个规则对这个接口进行实现，从而提供服务。

举个通俗易懂的例子：公司 H 是一家科技公司，新设计了一款芯片，然后现在需要量产了，而市面上有好几家芯片制造业公司，这个时候，只要 H 公司指定好了这芯片生产的标准（定义好了接口标准），那么这些合作的芯片公司（服务提供者）就按照标准交付自家特色的芯片（提供不同方案的实现，但是给出来的结果是一样的）。

## SPI 的优缺点？

通过 SPI 机制能够大大地提高接口设计的灵活性，但是 SPI 机制也存在一些缺点，比如：

- 需要遍历加载所有的实现类，不能做到按需加载，这样效率还是相对较低的。
- 当多个 `ServiceLoader` 同时 `load` 时，会有并发问题。

## 什么是序列化？什么是反序列化？

如果我们需要持久化 Java 对象比如将 Java 对象保存在文件中，或者在网络传输 Java 对象，这些场景都需要用到序列化。

简单来说：

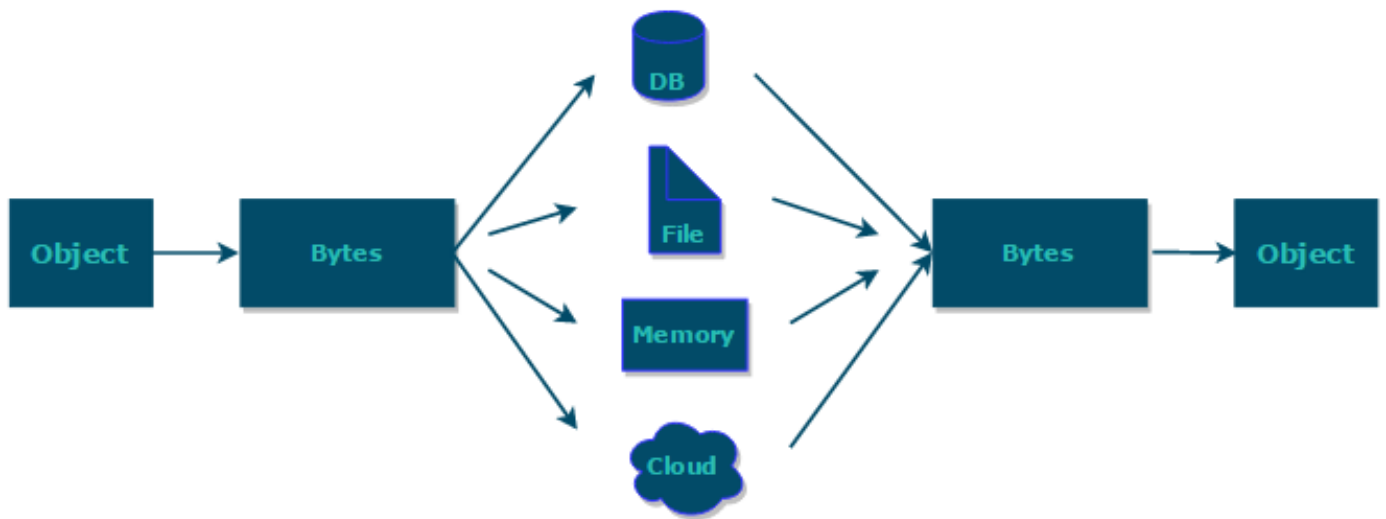
- **序列化**：将数据结构或对象转换成二进制字节流的过程
- **反序列化**：将在序列化过程中所生成的二进制字节流转换成数据结构或者对象的过程

对于 Java 这种面向对象编程语言来说，我们序列化的都是对象（Object）也就是实例化后的类（Class），但是在 C++ 这种半面向对象的语言中，`struct`（结构体）定义的是数据结构类型，而 `class` 对应的是对象类型。

维基百科是如是介绍序列化的：

**序列化**（serialization）在计算机科学的数据处理中，是指将数据结构或对象状态转换成可取用格式（例如存成文件，存于缓冲，或经由网络中发送），以留待后续在相同或另一台计算机环境中，能恢复原先状态的过程。依照序列化格式重新获取字节的结果时，可以利用它来产生与原始对象相同语义的副本。对于许多对象，像是使用大量引用的复杂对象，这种序列化重建的过程并不容易。面向对象中的对象序列化，并不概括之前原始对象所关系的函数。这种过程也称为对象编组（marshalling）。从一系列字节提取数据结构的反向操作，是反序列化（也称为解编组、deserialization、unmarshalling）。

综上：序列化的主要目的是通过网络传输对象或者说是将对象存储到文件系统、数据库、内存中。



<https://www.corejavaguru.com/java/serialization/interview-questions-1>

## 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。

关于 `transient` 还有几点注意：

- `transient` 只能修饰变量，不能修饰类和方法。
- `transient` 修饰的变量，在反序列化后变量值将会被置成类型的默认值。例如，如果是修饰 `int` 类型，那么反序列化后结果就是 `0`。
- `static` 变量因为不属于任何对象(Object)，所以无论有没有 `transient` 关键字修饰，均不会被序列化。

## Java IO 流了解吗？

IO 即 `Input/Output`，输入和输出。数据输入到计算机内存的过程即输入，反之输出到外部存储（比如数据库，文件，远程主机）的过程即输出。数据传输过程类似于水流，因此称为 IO 流。IO 流在 Java 中分为输入流和输出流，而根据数据的处理方式又分为字节流和字符流。

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。

相关阅读：[Java IO 基础知识总结](#)。

## I/O 流为什么要分为字节流和字符流呢？

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？

个人认为主要有两点原因：

- 字符流是由 Java 虚拟机将字节转换得到的，这个过程还算是比较耗时；
- 如果我们不知道编码类型的话，使用字节流的过程中很容易出现乱码问题。

## Java IO 中的设计模式有哪些？

[Java IO 设计模式总结](#)。

## BIO、NIO 和 AIO 的区别？

[Java IO 模型详解](#)。

---

## 2.2. Java集合

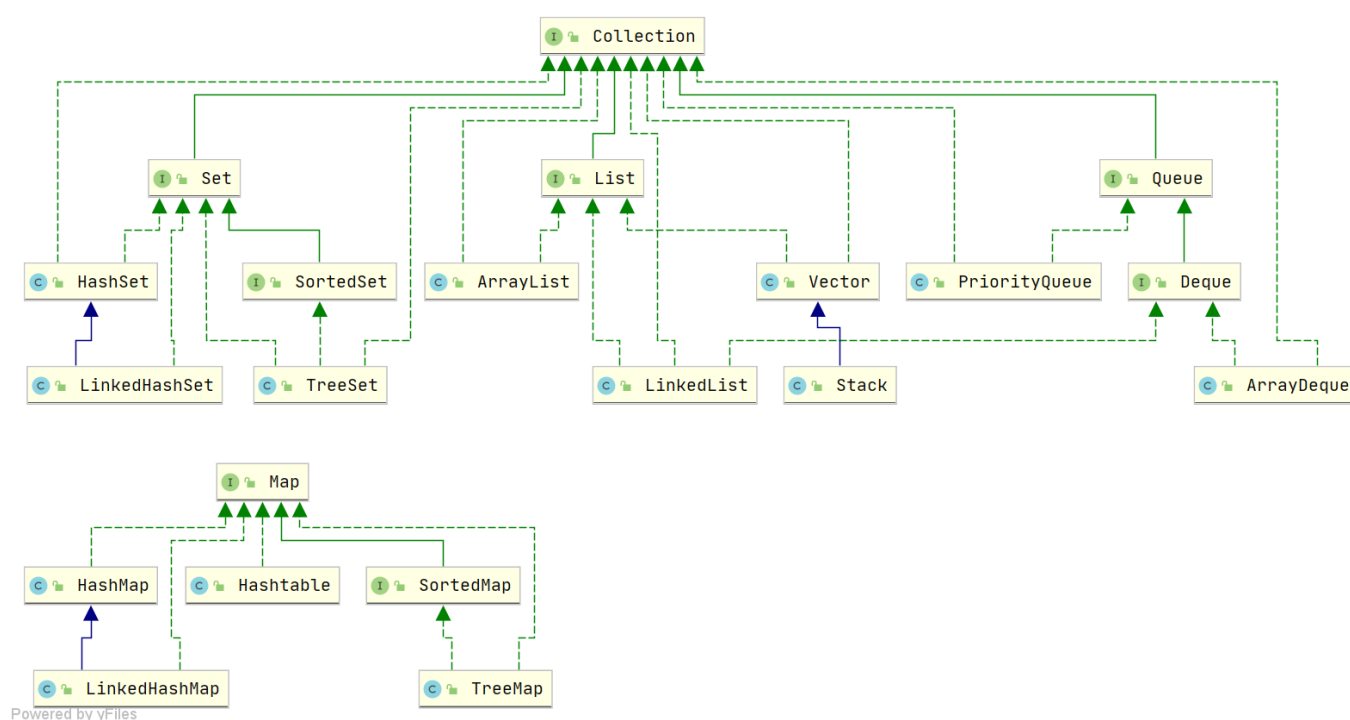
[JavaGuide](#)：「Java学习+面试指南」一份涵盖大部分 Java 程序员所需要掌握的核心知识。准备 Java 面试，首选 JavaGuide！

这部分内容摘自 [JavaGuide](#) 下面几篇文章：

- [Java集合常见面试题总结\(上\)](#)
- [Java集合常见面试题总结\(下\)](#)

Java 集合，也叫作容器，主要是由两大接口派生而来：一个是 `Collection` 接口，主要用于存放单一元素；另一个是 `Map` 接口，主要用于存放键值对。对于 `Collection` 接口，下面又有三个主要的子接口：`List`、`Set` 和 `Queue`。

Java 集合框架如下图所示：



注：图中只列举了主要的继承派生关系，并没有列举所有关系。比方省略了 `AbstractList`，`NavigableSet` 等抽象类以及其他的一些辅助类，如想深入了解，可自行查看源码。

## 说说 List, Set, Queue, Map 四者的区别?

- 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。
- Queue (实现排队功能的叫号机): 按特定的排队规则来确定先后顺序, 存储的元素是有序的、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。

## 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。

### 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)

### 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)

### Queue

- PriorityQueue : Object[] 数组来实现二叉堆
- ArrayQueue : Object[] 数组 + 双指针

再看看 Map 接口下面的集合。

### 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 `Hashtable` 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）

## 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。

## 为什么要使用集合？

当我们需要保存一组类型相同的数据的时候，我们应该是用一个容器来保存，这个容器就是数组，但是，使用数组存储对象具有一定的弊端，因为我们在实际开发中，存储的数据的类型是多种多样的，于是，就出现了“集合”，集合同样也是用来存储多个数据的。

数组的缺点是一旦声明之后，长度就不可变了；同时，声明数组时的数据类型也决定了该数组存储的数据的类型；而且，数组存储的数据是有序的、可重复的，特点单一。但是集合提高了数据存储的灵活性，Java 集合不仅可以用来存储不同类型不同数量的对象，还可以保存具有映射关系的数据。

## ArrayList 和 Vector 的区别？

- `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。

## ArrayList 与 LinkedList 区别？

- **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；
- **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！）
- **插入和删除是否受元素位置的影响：**
  - `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。
  - `LinkedList` 采用链表存储，所以，如果是在头尾插入或者删除元素不受元素位置的影响（`add(E e)`、`addFirst(E e)`、`addLast(E e)`、`removeFirst()`、`removeLast()`），时间复杂度为  $O(1)$ ，如果是要在指定位置 `i` 插入和删除元素的话（`add(int index, E element)`，`remove(Object o)`），时间复杂度为  $O(n)$ ，因为需要先移动到指定位置再插入。
- **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)。
- **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）。

我们在项目中一般是不会使用到 `LinkedList` 的，需要用到 `LinkedList` 的场景几乎都可以使用 `ArrayList` 来代替，并且，性能通常会更好！就连 `LinkedList` 的作者约书亚·布洛克（Josh Bloch）自己都说从来不会使用 `LinkedList`。



Joshua Bloch

@joshbloch

关注



回复 @jerrykuch

@jerrykuch @shipilev @AmbientLion Does anyone actually use LinkedList? I wrote it, and I never use it.

下午7:10 - 2015年4月2日

272 转推 313 喜欢



20

272

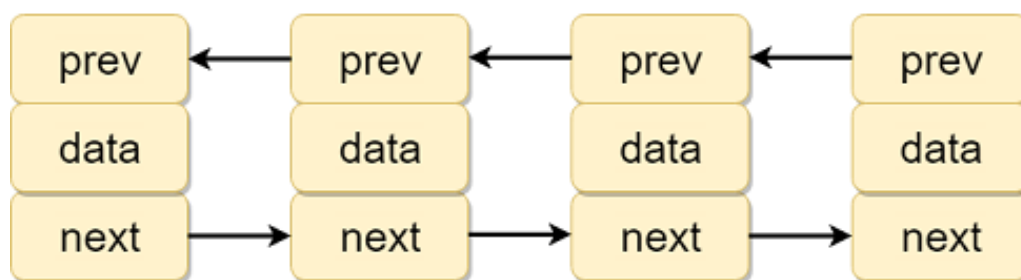
313

另外，不要下意识地认为 `LinkedList` 作为链表就最适合元素增删的场景。我在上面也说了，`LinkedList` 仅仅在头尾插入或者删除元素的时候时间复杂度近似  $O(1)$ ，其他情况增删元素的时间复杂度都是  $O(n)$ 。

## 补充内容:双向链表和双向循环链表

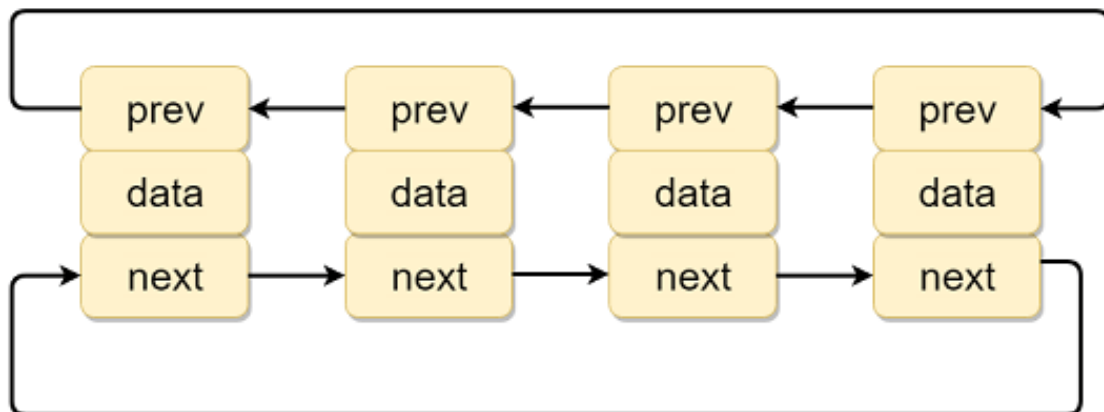
双向链表：包含两个指针，一个 `prev` 指向前一个节点，一个 `next` 指向后一个节点。

### 双向链表



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



### 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！

## 说一说 ArrayList 的扩容机制吧

详见笔主的这篇文章: [ArrayList 扩容机制分析](#)

## comparable 和 Comparator 的区别

- comparable 接口实际上是出自 java.lang 包 它有一个 compareTo(Object obj) 方法用来排序
- comparator 接口实际上是出自 java.util 包 它有一个 compare(Object obj1, Object obj2) 方法用来排序

一般我们需要对一个集合使用自定义排序时，我们就要重写 compareTo() 方法或 compare() 方法，当我们需要对某一个集合实现两种排序方式，比如一个 song 对象中的歌名和歌手名分别采用一种排序方法的话，我们可以重写 compareTo() 方法和使用自制的 Comparator 方法或者以两个 Comparator 来实现歌名排序和歌星名排序，第二种代表我们只能使用两个参数版的 Collections.sort() 。

## Comparator 定制排序

```
ArrayList<Integer> arrayList = new ArrayList<Integer>();
arrayList.add(-1);
arrayList.add(3);
arrayList.add(3);
arrayList.add(-5);
arrayList.add(7);
arrayList.add(4);
arrayList.add(-9);
arrayList.add(-7);

System.out.println("原始数组:");
System.out.println(arrayList);
// void reverse(List list): 反转
Collections.reverse(arrayList);
System.out.println("Collections.reverse(arrayList):");
System.out.println(arrayList);

// void sort(List list),按自然排序的升序排序
Collections.sort(arrayList);
System.out.println("Collections.sort(arrayList):");
```

```

        System.out.println(arrayList);

        // 定制排序的用法

        Collections.sort(arrayList, new Comparator<Integer>() {

            @Override

            public int compare(Integer o1, Integer o2) {

                return o2.compareTo(o1);

            }

        });

        System.out.println("定制排序后: ");

        System.out.println(arrayList);

```

Output:

```

原始数组:
[-1, 3, 3, -5, 7, 4, -9, -7]
Collections.reverse(arrayList):
[-7, -9, 4, 7, -5, 3, 3, -1]
Collections.sort(arrayList):
[-9, -7, -5, -1, 3, 3, 4, 7]
定制排序后:
[7, 4, 3, 3, -1, -5, -7, -9]

```

## 重写 compareTo 方法实现按年龄来排序

```

// person对象没有实现Comparable接口，所以必须实现，这样才不会出错，才可以使treemap中的数据
按顺序排列

// 前面一个例子的String类已经默认实现了Comparable接口，详细可以查看String类的API文档，另外
其他

// 像Integer类等都已经实现了Comparable接口，所以不需要另外实现了

public class Person implements Comparable<Person> {

    private String name;

    private int age;

    public Person(String name, int age) {

        super();

```

```
        this.name = name;

        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    /**
     * T重写compareTo方法实现按年龄来排序
     */
    @Override
    public int compareTo(Person o) {
        if (this.age > o.getAge()) {
            return 1;
        }

        if (this.age < o.getAge()) {
            return -1;
        }

        return 0;
    }
}
```

```

public static void main(String[] args) {
    TreeMap<Person, String> pdata = new TreeMap<Person, String>();
    pdata.put(new Person("张三", 30), "zhangsan");
    pdata.put(new Person("李四", 20), "lisi");
    pdata.put(new Person("王五", 10), "wangwu");
    pdata.put(new Person("小红", 5), "xiaohong");
    // 得到key的值的的同时得到key所对应的值
    Set<Person> keys = pdata.keySet();
    for (Person key : keys) {
        System.out.println(key.getAge() + "-" + key.getName());
    }
}

```

Output:

```

5-小红
10-王五
20-李四
30-张三

```

## 无序性和不可重复性的含义是什么

- 无序性不等于随机性，无序性是指存储的数据在底层数组中并非按照数组索引的顺序添加，而是根据数据的哈希值决定的。
- 不可重复性是指添加的元素按照 `equals()` 判断时，返回 `false`，需要同时重写 `equals()` 方法和 `hashCode()` 方法。

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

- `HashSet`、`LinkedHashSet` 和 `TreeSet` 都是 `Set` 接口的实现类，都能保证元素唯一，并且都不是线程安全的。
- `HashSet`、`LinkedHashSet` 和 `TreeSet` 的主要区别在于底层数据结构不同。`HashSet` 的底层数据结构是哈希表（基于 `HashMap` 实现）。`LinkedHashSet` 的底层数据结构是链表和哈希表，元素的插入和取出顺序满足 FIFO。`TreeSet` 底层数据结构是红黑树，元素是有序的，排序的方式有自然排序和定制排序。



- 底层数据结构不同又导致这三者的应用场景不同。`HashSet` 用于不需要保证元素插入和取出顺序的场景，`LinkedHashSet` 用于保证元素的插入和取出顺序满足 FIFO 的场景，`TreeSet` 用于支持对元素自定义排序规则的场景。

## Queue 与 Deque 的区别

`Queue` 是单端队列，只能从一端插入元素，另一端删除元素，实现上一般遵循 **先进先出（FIFO）** 规则。

`Queue` 扩展了 `Collection` 的接口，根据 **因为容量问题而导致操作失败后处理方式的不同** 可以分为两类方法：一种在操作失败后会抛出异常，另一种则会返回特殊值。

<code>Queue</code> 接口	抛出异常	返回特殊值
插入队尾	<code>add(E e)</code>	<code>offer(E e)</code>
删除队首	<code>remove()</code>	<code>poll()</code>
查询队首元素	<code>element()</code>	<code>peek()</code>

`Deque` 是双端队列，在队列的两端均可以插入或删除元素。

`Deque` 扩展了 `Queue` 的接口，增加了在队首和队尾进行插入和删除的方法，同样根据失败后处理方式的不同分为两类：

<code>Deque</code> 接口	抛出异常	返回特殊值
插入队首	<code>addFirst(E e)</code>	<code>offerFirst(E e)</code>
插入队尾	<code>addLast(E e)</code>	<code>offerLast(E e)</code>
删除队首	<code>removeFirst()</code>	<code>pollFirst()</code>
删除队尾	<code>removeLast()</code>	<code>pollLast()</code>
查询队首元素	<code>getFirst()</code>	<code>peekFirst()</code>
查询队尾元素	<code>getLast()</code>	<code>peekLast()</code>

事实上，`Deque` 还提供有 `push()` 和 `pop()` 等其他方法，可用于模拟栈。

## ArrayDeque 与 LinkedList 的区别

`ArrayDeque` 和 `LinkedList` 都实现了 `Deque` 接口，两者都具有队列的功能，但两者有什么区别呢？

- `ArrayDeque` 是基于可变长的数组和双指针来实现，而 `LinkedList` 则通过链表来实现。
- `ArrayDeque` 不支持存储 `NULL` 数据，但 `LinkedList` 支持。
- `ArrayDeque` 是在 JDK1.6 才被引入的，而 `LinkedList` 早在 JDK1.2 时就已经存在。
- `ArrayDeque` 插入时可能存在扩容过程，不过均摊后的插入操作依然为  $O(1)$ 。虽然 `LinkedList` 不需要扩容，但是每次插入数据时均需要申请新的堆空间，均摊性能相比更慢。

从性能的角度上，选用 `ArrayDeque` 来实现队列要比 `LinkedList` 更好。此外，`ArrayDeque` 也可以用于实现栈。

## 说一说 PriorityQueue

`PriorityQueue` 是在 JDK1.5 中被引入的，其与 `Queue` 的区别在于元素出队顺序是与优先级相关的，即总是优先级最高的元素先出队。

这里列举其相关的一些要点：

- `PriorityQueue` 利用了二叉堆的数据结构来实现的，底层使用可变长的数组来存储数据
- `PriorityQueue` 通过堆元素的上浮和下沉，实现了在  $O(\log n)$  的时间复杂度内插入元素和删除堆顶元素。
- `PriorityQueue` 是非线程安全的，且不支持存储 `NULL` 和 `non-comparable` 的对象。
- `PriorityQueue` 默认是小顶堆，但可以接收一个 `Comparator` 作为构造参数，从而来自定义元素优先级的先后。

`PriorityQueue` 在面试中可能更多的会出现在手撕算法的时候，典型例题包括堆排序、求第K大的数、带权图的遍历等，所以需要会熟练使用才行。

## 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）时，将链表转化为红黑树（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），以减少搜索时间（后文中我会结合源码对这一过程进行分析）。`Hashtable` 没有这样的机制。

`HashMap` 中带有初始容量的构造函数：

```
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                                           initialCapacity);

    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                                           loadFactor);

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

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;
}

```

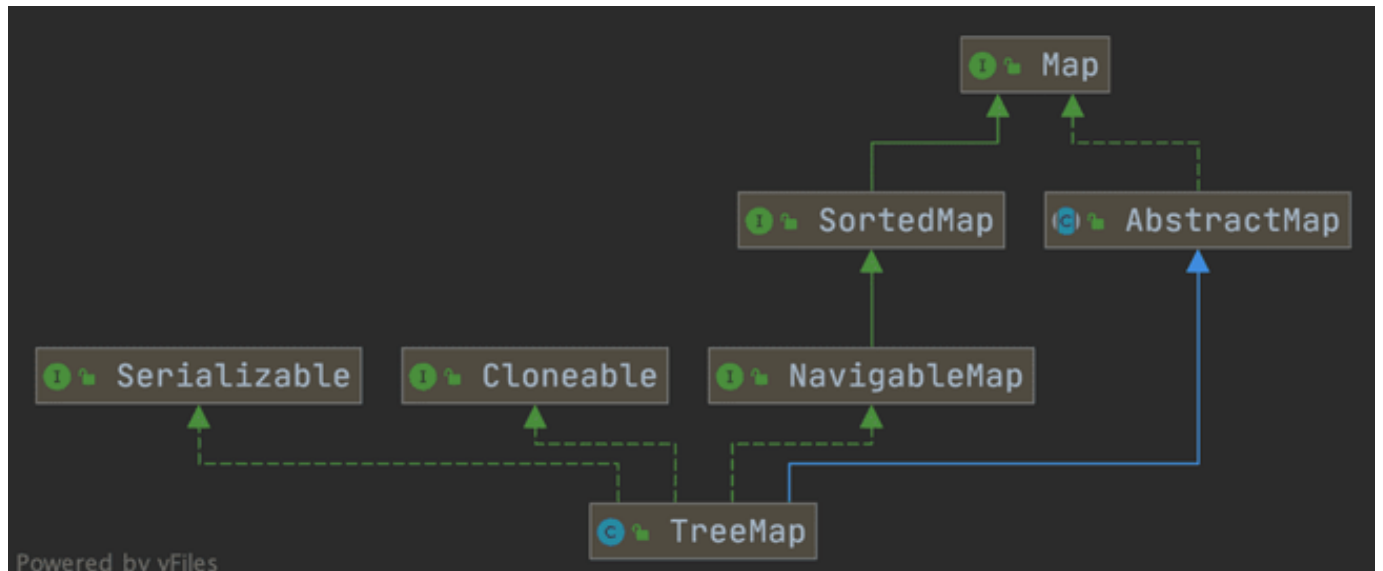
## HashMap 和 HashSet 区别

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 中添加元素	调用 add() 方法向 Set 中添加元素
HashMap 使用键 (Key) 计算 hashCode	HashSet 使用成员对象来计算 hashCode 值，对于两个对象来说 hashCode 可能相同，所以 equals() 方法用来判断对象的相等性

## HashMap 和 TreeMap 区别

TreeMap 和 HashMap 都继承自 AbstractMap ，但是需要注意的是 TreeMap 它还实现了 NavigableMap 接口和 SortedMap 接口。



实现 `NavigableMap` 接口让 `TreeMap` 有了对集合内元素的搜索的能力。

实现 `SortedMap` 接口让 `TreeMap` 有了对集合中的元素根据键排序的能力。默认是按 `key` 的升序排序，不过我们也可以指定排序的比较器。示例代码如下：

```
/**
 * @author shuang.kou
 * @createTime 2020年06月15日 17:02:00
 */
public class Person {
    private Integer age;

    public Person(Integer age) {
        this.age = age;
    }

    public Integer getAge() {
        return age;
    }

    public static void main(String[] args) {
        TreeMap<Person, String> treeMap = new TreeMap<>(new Comparator<Person>())
        {
            @Override
            public int compare(Person person1, Person person2) {
```

```

        int num = person1.getAge() - person2.getAge();
        return Integer.compare(num, 0);
    }
});
treeMap.put(new Person(3), "person1");
treeMap.put(new Person(18), "person2");
treeMap.put(new Person(35), "person3");
treeMap.put(new Person(16), "person4");
treeMap.entrySet().stream().forEach(personStringEntry -> {
    System.out.println(personStringEntry.getValue());
});
}
}

```

输出:

```

person1
person4
person2
person3

```

可以看出，`TreeMap` 中的元素已经是按照 `Person` 的 `age` 字段的升序来排列了。

上面，我们是通过传入匿名内部类的方式实现的，你可以将代码替换成 Lambda 表达式实现的方式：

```

TreeMap<Person, String> treeMap = new TreeMap<>((person1, person2) -> {
    int num = person1.getAge() - person2.getAge();
    return Integer.compare(num, 0);
});

```

综上，相比于 `HashMap` 来说 `TreeMap` 主要多了对集合中的元素根据键排序的能力以及对集合内元素的搜索的能力。

## HashSet 如何检查重复？

以下内容摘自我的 Java 启蒙书《Head first 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。

在 JDK1.8 中，HashSet 的 add() 方法只是简单的调用了 HashMap 的 put() 方法，并且判断了一下返回值以确保是否有重复元素。直接看一下 HashSet 中的源码：

```
// Returns: true if this set did not already contain the specified element
// 返回值：当 set 中没有包含 add 的元素时返回真
public boolean add(E e) {
    return map.put(e, PRESENT) == null;
}
```

而在 HashMap 的 putVal() 方法中也能看到如下说明：

```
// Returns : previous value, or null if none
// 返回值：如果插入位置没有元素返回null，否则返回上一个元素
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    ...
}
```

也就是说，在 JDK1.8 中，实际上无论 HashSet 中是否已经存在了某元素，HashSet 都会直接插入，只是会在 add() 方法的返回值处告诉我们插入前是否存在相同元素。

# HashMap 的底层实现

## 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);  
}
```

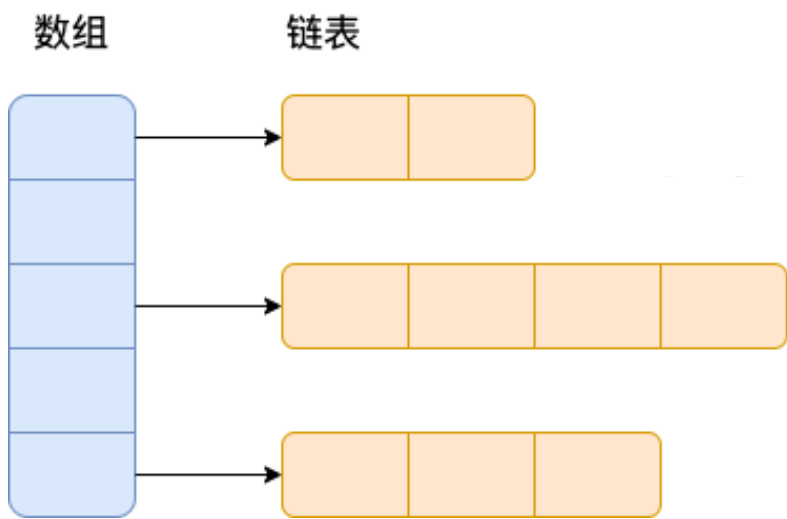
对比一下 JDK1.7 的 `HashMap` 的 `hash` 方法源码.

```
static int hash(int h) {  
    // This function ensures that hashCodes that 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。

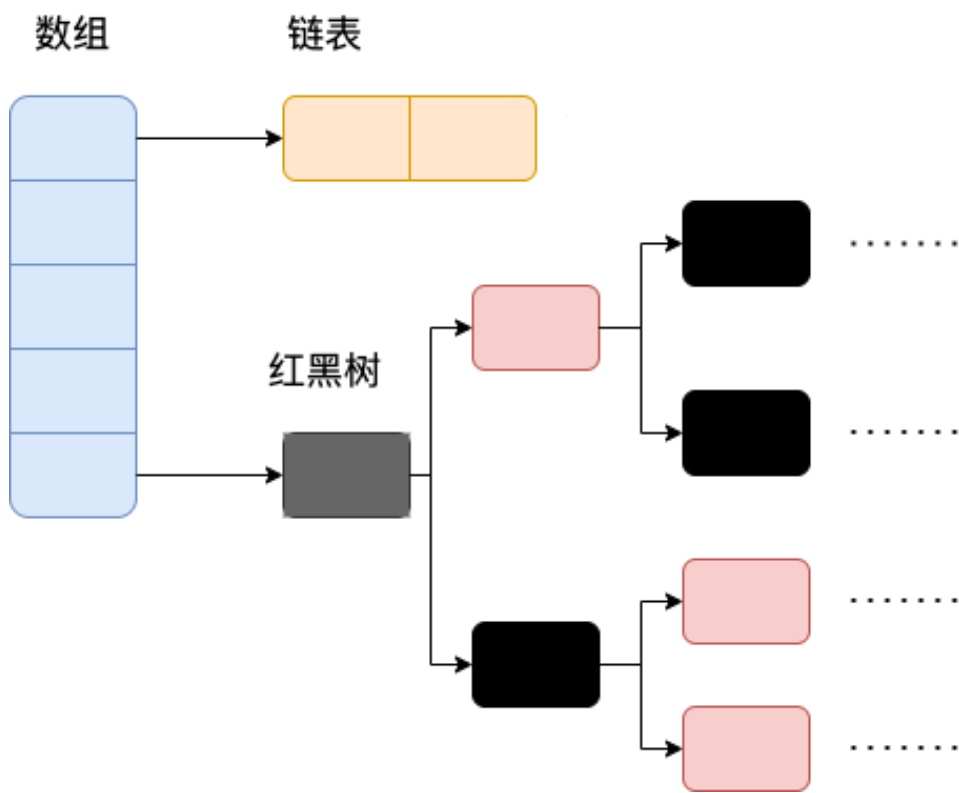


所谓“拉链法”就是：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。



### 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。

我们来结合源码分析一下 `HashMap` 链表到红黑树的转换。

## 1、`putVal` 方法中执行链表转红黑树的判断逻辑。

链表的长度大于 8 的时候，就执行 `treeifyBin`（转换红黑树）的逻辑。

```
// 遍历链表
for (int binCount = 0; ; ++binCount) {
    // 遍历到链表最后一个节点
    if ((e = p.next) == null) {
        p.next = newNode(hash, key, value, null);
        // 如果链表元素个数大于等于TREEIFY_THRESHOLD (8)
        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
            // 红黑树转换（并不会直接转换成红黑树）
            treeifyBin(tab, hash);
        break;
    }
    if (e.hash == hash &&
        ((k = e.key) == key || (key != null && key.equals(k))))
        break;
    p = e;
}
```

## 2、`treeifyBin` 方法中判断是否真的转换为红黑树。

```
final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    // 判断当前数组的长度是否小于 64
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        // 如果当前数组的长度小于 64，那么会选择先进行数组扩容
        resize();
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        // 否则才将列表转换为红黑树

        TreeNode<K,V> hd = null, tl = null;
        do {
            TreeNode<K,V> p = replacementTreeNode(e, null);
```

```

        if (tl == null)
            hd = p;
        else {
            p.prev = tl;
            tl.next = p;
        }
        tl = p;
    } while ((e = e.next) != null);
    if ((tab[index] = hd) != null)
        hd.treeify(tab);
}
}

```

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。

## 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“ $(n - 1) \& \text{hash}$ ”。（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  $\text{hash} \% \text{length} == \text{hash} \& (\text{length} - 1)$  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。

## 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>

# HashMap 有哪几种常见的遍历方式？

HashMap 的 7 种遍历方式与性能分析！

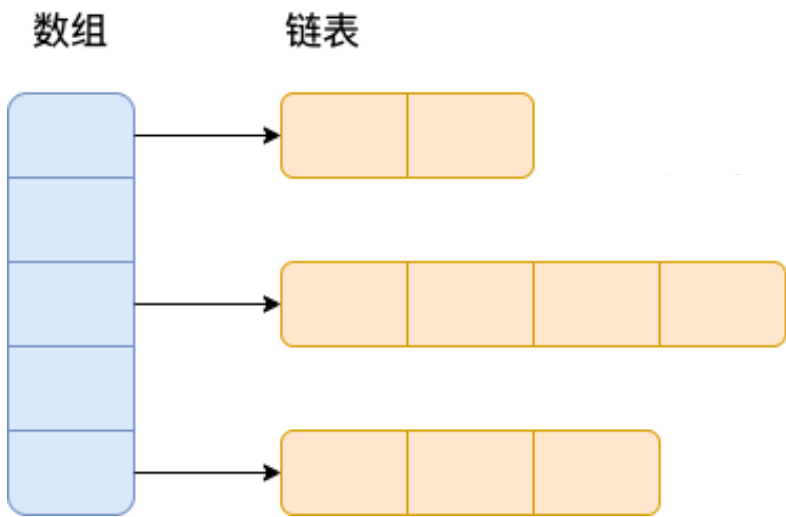
## ConcurrentHashMap 和 Hashtable 的区别

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

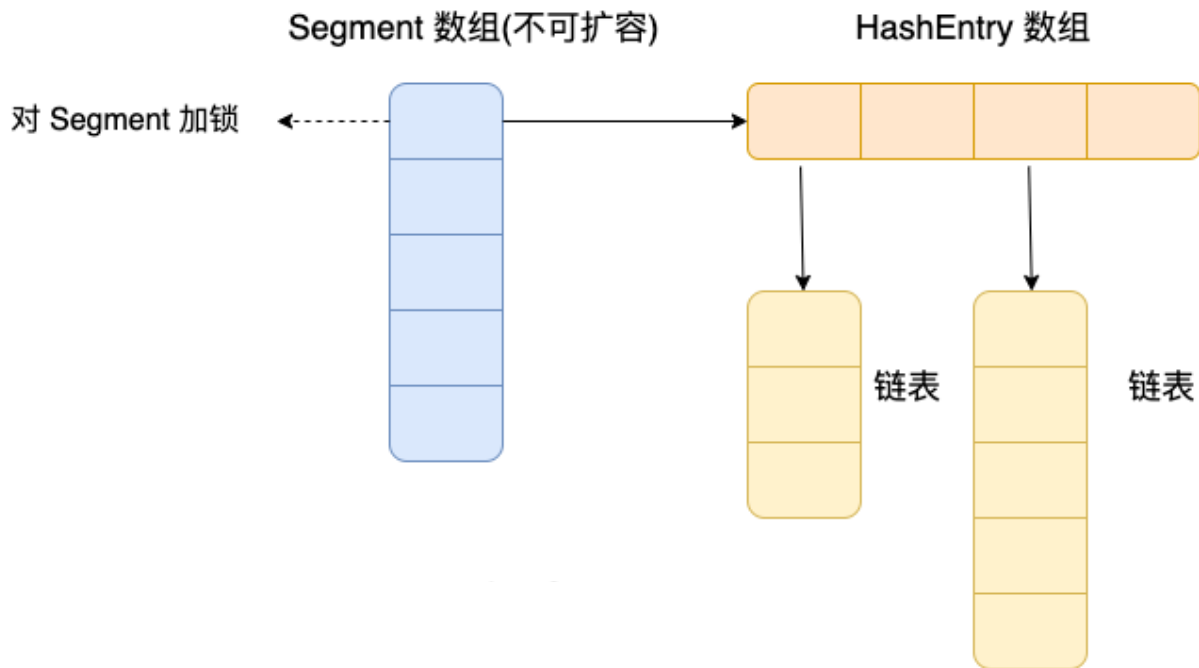
- **底层数据结构：** JDK1.7 的 ConcurrentHashMap 底层采用 分段的数组+链表 实现，JDK1.8 采用的数据结构跟 HashMap1.8 的结构一样，数组+链表/红黑二叉树。 Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 数组+链表 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；
- **实现线程安全的方式（重要）：**
  - 在 JDK1.7 的时候， ConcurrentHashMap 对整个桶数组进行了分割分段( Segment ，分段锁)，每一把锁只锁容器其中一部分数据（下面有示意图），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， ConcurrentHashMap 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 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；
  - **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。

下面，我们再来看看两者底层数据结构的对比图。

Hashtable :



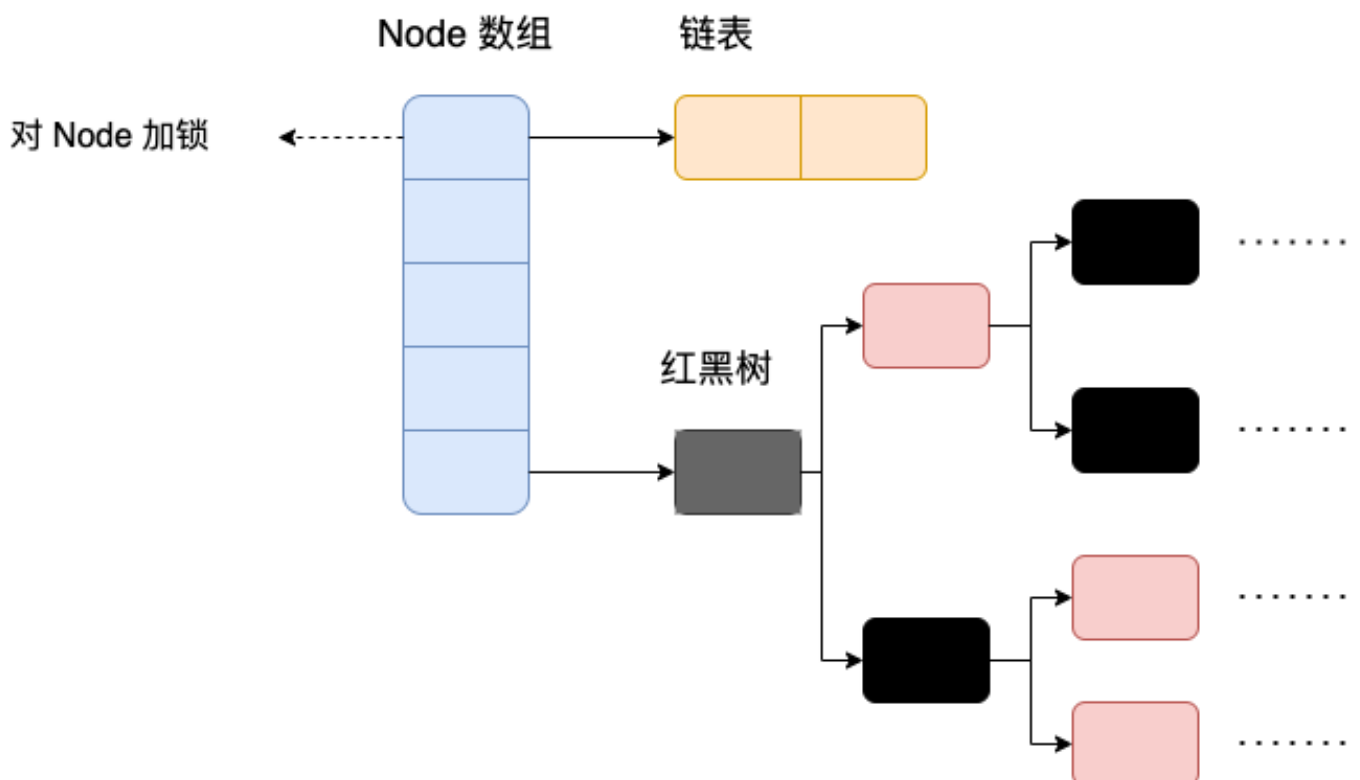
## JDK1.7 的 ConcurrentHashMap :



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

Segment 数组中的每个元素包含一个 HashEntry 数组，每个 HashEntry 数组属于链表结构。

## 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。

`TreeNode` 是存储红黑树节点，被 `TreeBin` 包装。`TreeBin` 通过 `root` 属性维护红黑树的根结点，因为红黑树在旋转的时候，根结点可能会被它原来的子节点替换掉，在这个时间点，如果有其他线程要写这棵红黑树就会发生线程不安全问题，所以在 `ConcurrentHashMap` 中 `TreeBin` 通过 `waiter` 属性维护当前使用这棵红黑树的线程，来防止其他线程的进入。

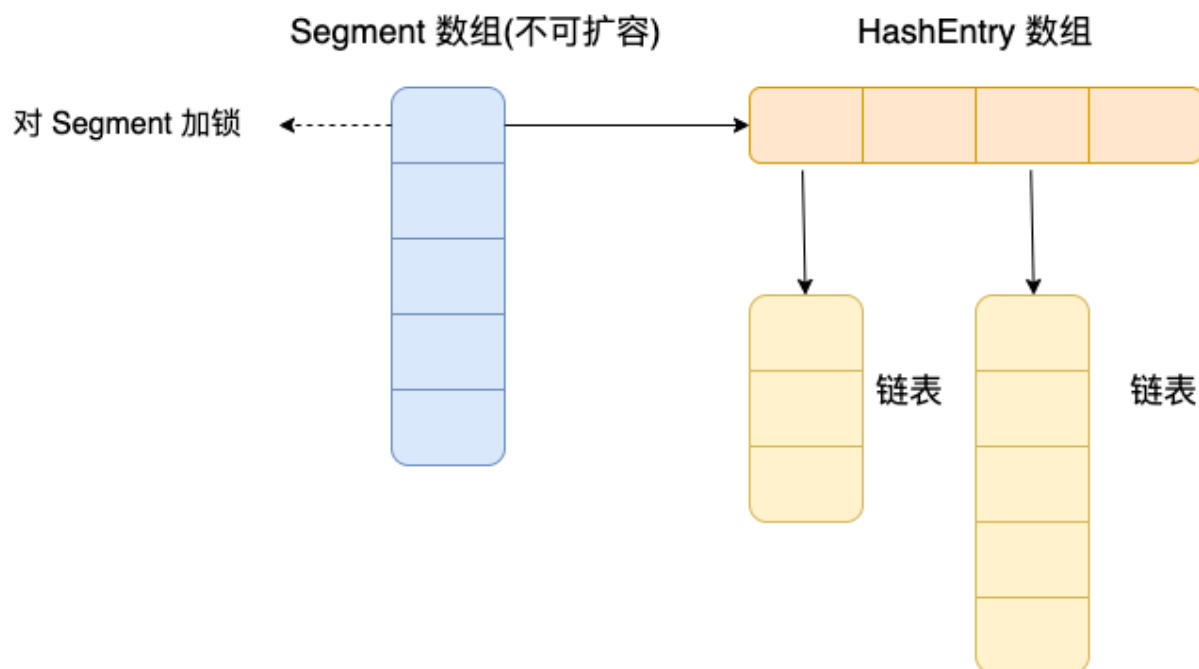
```
static final class TreeBin<K,V> extends Node<K,V> {
    TreeNode<K,V> root;

    volatile TreeNode<K,V> first;
    volatile Thread waiter;
    volatile int lockState;
    // values for lockState
    static final int WRITER = 1; // set while holding write lock
    static final int WAITER = 2; // set when waiting for write lock
    static final int READER = 4; // increment value for setting read lock

    ...
}
```

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

### JDK1.8 之前



首先将数据分为一段一段（这个“段”就是 Segment）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。

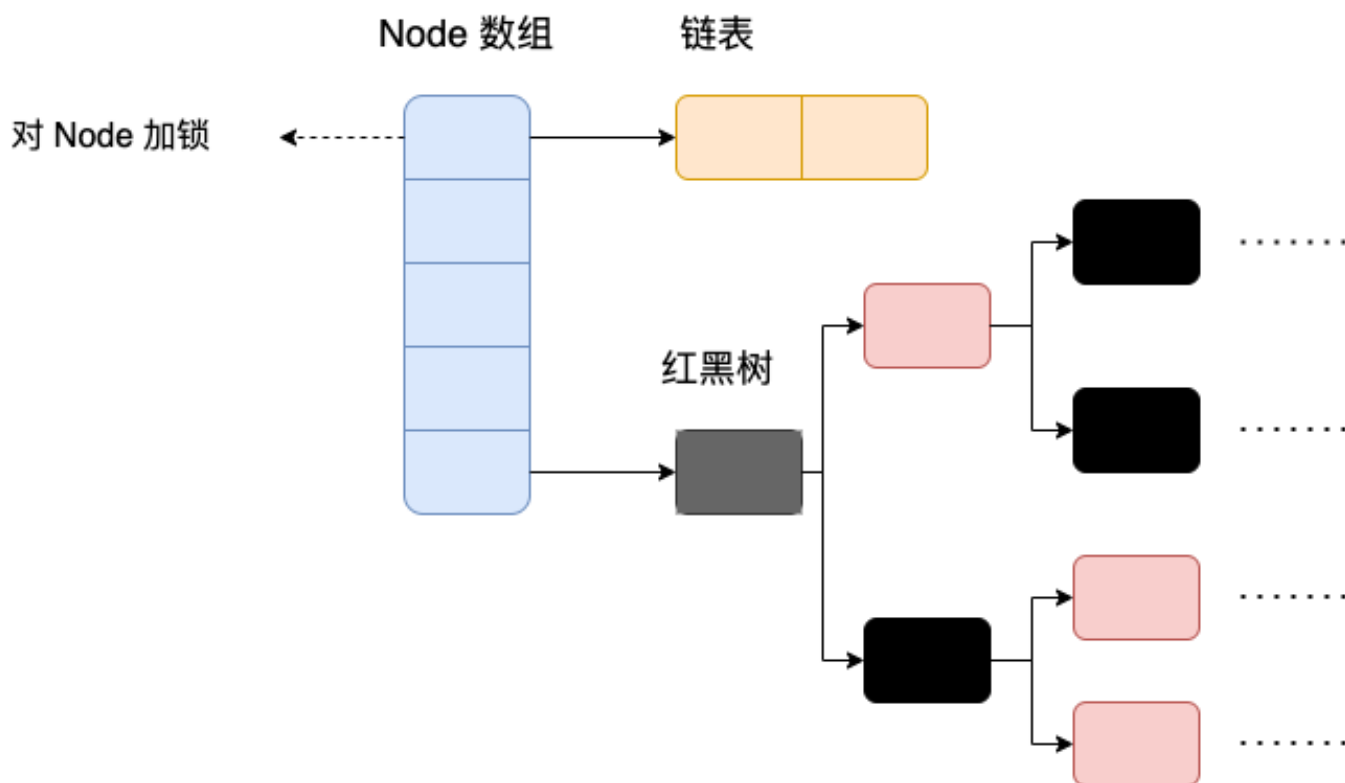
Segment 继承了 ReentrantLock, 所以 Segment 是一种可重入锁，扮演锁的角色。HashEntry 用于存储键值对数据。

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

一个 ConcurrentHashMap 里包含一个 Segment 数组，Segment 的个数一旦初始化就不能改变。Segment 数组的大小默认是 16，也就是说默认可以同时支持 16 个线程并发写。

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。也就是说，对同一 Segment 的并发写入会被阻塞，不同 Segment 的写入是可以并发执行的。

## JDK1.8 之后



Java 8 几乎完全重写了 `ConcurrentHashMap`，代码量从原来 Java 7 中的 1000 多行，变成了现在的 6000 多行。

`ConcurrentHashMap` 取消了 `Segment` 分段锁，采用 `Node + CAS + synchronized` 来保证并发安全。数据结构跟 `HashMap` 1.8 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  $O(\log(N))$ ）。

Java 8 中，锁粒度更细，`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，就不会影响其他 Node 的读写，效率大幅提升。

## JDK 1.7 和 JDK 1.8 的 `ConcurrentHashMap` 实现有什么不同？

- **线程安全实现方式**：JDK 1.7 采用 `Segment` 分段锁来保证安全，`Segment` 是继承自 `ReentrantLock`。JDK1.8 放弃了 `Segment` 分段锁的设计，采用 `Node + CAS + synchronized` 保证线程安全，锁粒度更细，`synchronized` 只锁定当前链表或红黑二叉树的首节点。
- **Hash 碰撞解决方法**：JDK 1.7 采用拉链法，JDK1.8 采用拉链法结合红黑树（链表长度超过一定阈值时，将链表转换为红黑树）。
- **并发度**：JDK 1.7 最大并发度是 `Segment` 的个数，默认是 16。JDK 1.8 最大并发度是 Node 数组的大小，并发度更大。



---

## 2.3. 多线程

[JavaGuide](#)：「Java学习+面试指南」一份涵盖大部分 Java 程序员所需要掌握的核心知识。准备 Java 面试，首选 JavaGuide！

这部分内容摘自 [JavaGuide](#) 下面几篇文章：

- [Java 并发常见面试题总结（上）](#)
- [Java 并发常见面试题总结（中）](#)
- [Java 并发常见面试题总结（下）](#)

## 什么是线程和进程？

### 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 Windows 当前运行的进程（.exe 文件的运行）。

任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	状态	11% CPU	57% 内存	1% 磁盘	0% 网络
应用 (6)					
> 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
后台进程 (71)					
> 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)

## 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 来看看一个普通的 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 MBean
        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

```

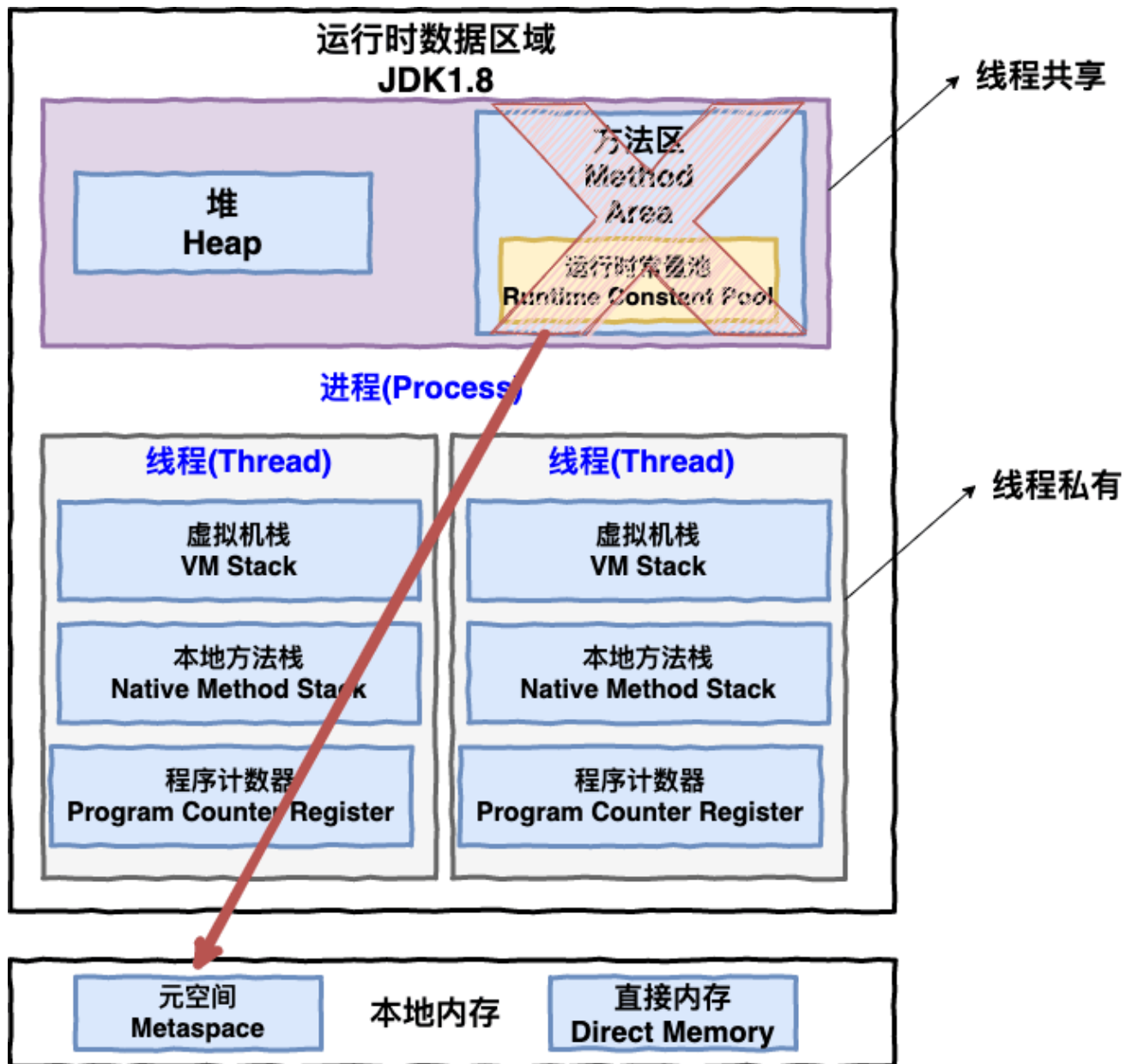
从上面的输出内容可以看出：一个 Java 程序的运行是 main 线程和多个其他线程同时运行。

## 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。

### 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。



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？

## 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。

## 虚拟机栈和本地方法栈为什么是私有的？

- **虚拟机栈：** 每个 Java 方法在运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。

## 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（几乎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。

## 并发与并行的区别

- **并发：** 两个及两个以上的作业在同一 **时间段** 内执行。
- **并行：** 两个及两个以上的作业在同一 **时刻** 执行。

最关键的点是：是否是 **同时** 执行。

## 同步和异步的区别

- **同步**：发出一个调用之后，在没有得到结果之前，该调用就不可以返回，一直等待。
- **异步**：调用在发出之后，不用等待返回结果，该调用直接返回。

## 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单进程利用 CPU 和 IO 系统的效率。假设只运行了一个 Java 进程的情况，当我们请求 IO 的时候，如果 Java 进程中只有一个线程，此线程被 IO 阻塞则整个进程被阻塞。CPU 和 IO 设备只有一个在运行，那么可以简单地说系统整体效率只有 50%。当使用多线程的时候，一个线程被 IO 阻塞，其他线程还可以继续使用 CPU。从而提高了 Java 进程利用系统资源的整体效率。
- **多核时代**：多核时代多线程主要是为了提高进程利用多核 CPU 的能力。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，不论系统有几个 CPU 核心，都只会有一个 CPU 核心被利用到。而创建多个线程，这些线程可以被映射到底层多个 CPU 上执行，在任务中的多个线程没有资源竞争的情况下，任务执行的效率会有显著性的提高，约等于（单核时执行时间/CPU 核心数）。

## 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、死锁、线程不安全等等。

## 说说线程的生命周期和状态？

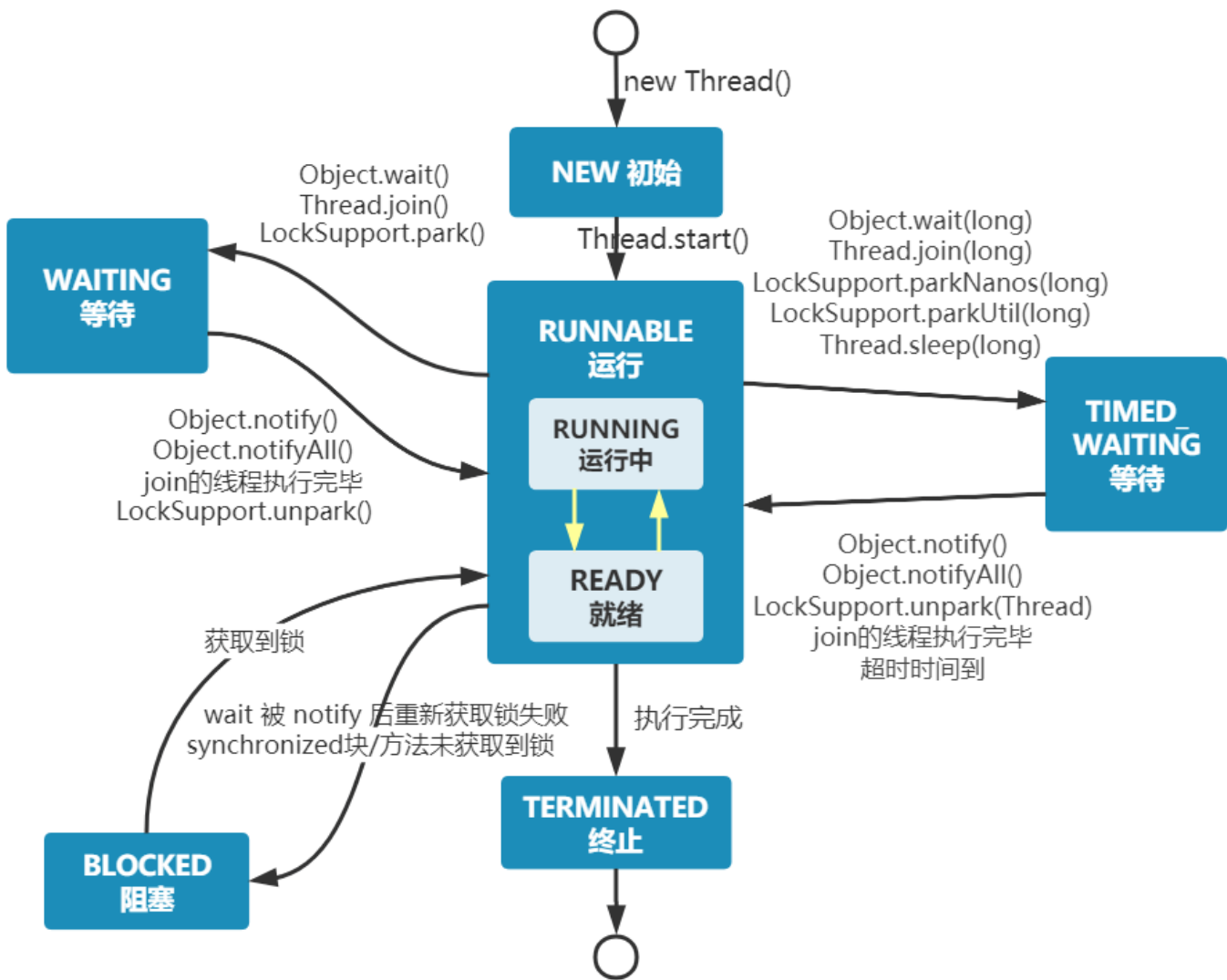
Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态：

- **NEW**: 初始状态，线程被创建出来但没有被调用 `start()`。
- **RUNNABLE**: 运行状态，线程被调用了 `start()` 等待运行的状态。

- **BLOCKED**：阻塞状态，需要等待锁释放。
- **WAITING**：等待状态，表示该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，可以在指定的时间后自行返回而不是像 **WAITING** 那样一直等待。
- **TERMINATED**：终止状态，表示该线程已经运行完毕。

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 线程状态变迁图(图源：[挑错 I 《Java 并发编程的艺术》中关于线程状态的三处错误](#))：

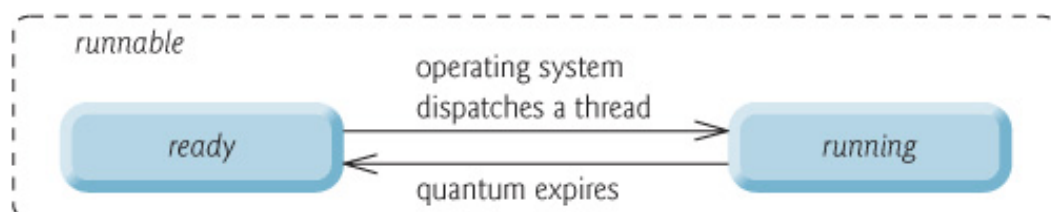


由上图可以看出：线程创建之后它将处于 **NEW**（新建）状态，调用 `start()` 方法后开始运行，线程这时候处于 **READY**（可运行）状态。可运行状态的线程获得了 CPU 时间片（timeslice）后就处于 **RUNNING**（运行）状态。



在操作系统层面，线程有 **READY** 和 **RUNNING** 状态；而在 JVM 层面，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。

为什么 JVM 没有区分这两种状态呢？（摘自：[Java 线程运行怎么有第六种状态？ - Dawell 的回答](#)）现在的时分（time-sharing）多任务（multi-task）操作系统架构通常都是用所谓的“时间分片（time quantum or time slice）”方式进行抢占式（preemptive）轮转调度（round-robin 式）。这个时间分片通常是很小的，一个线程一次最多只能在 CPU 上运行比如 10-20ms 的时间（此时处于 running 状态），也即大概只有 0.01 秒这一量级，时间片用后就要被切换下来放入调度队列的末尾等待再次调度。（也即回到 ready 状态）。线程切换的如此之快，区分这两种状态就没什么意义了。



- 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。
- **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)` 方法可以将线程置于 **TIMED\_WAITING** 状态。当超时时间结束后，线程将会返回到 **RUNNABLE** 状态。
- 当线程进入 `synchronized` 方法/块或者调用 `wait` 后（被 `notify`）重新进入 `synchronized` 方法/块，但是锁被其它线程占有，这个时候线程就会进入 **BLOCKED（阻塞）** 状态。
- 线程在执行完了 `run()` 方法之后将会进入到 **TERMINATED（终止）** 状态。

相关阅读：[线程的几种状态你真的了解么？](#)。

## 什么是上下文切换？

线程在执行过程中会有自己的运行条件和状态（也称上下文），比如上文所说到过的程序计数器，栈信息等。当出现如下情况的时候，线程会从占用 CPU 状态中退出。

- 主动让出 CPU，比如调用了 `sleep()`，`wait()` 等。
- 时间片用完，因为操作系统要防止一个线程或者进程长时间占用 CPU 导致其他线程或者进程饿死。
- 调用了阻塞类型的系统中断，比如请求 IO，线程被阻塞。
- 被终止或结束运行



这其中前三种都会发生线程切换，线程切换意味着需要保存当前线程的上下文，留待线程下次占用 CPU 的时候恢复现场。并加载下一个将要占用 CPU 的线程上下文。这就是所谓的 上下文切换。

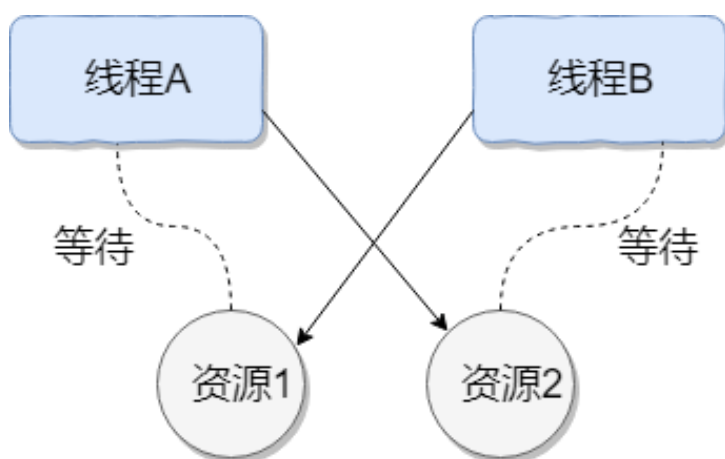
上下文切换是现代操作系统的基本功能，因其每次需要保存信息恢复信息，这将会占用 CPU，内存等系统资源进行处理，也就意味着效率会有一定损耗，如果频繁切换就会造成整体效率低下。

## 什么是线程死锁?如何避免死锁?

### 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：

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：一个线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：若干线程之间形成一种头尾相接的循环等待资源关系。

## 如何预防和避免线程死锁？

如何预防死锁？ 破坏死锁的产生的必要条件即可：

1. 破坏请求与保持条件：一次性申请所有的资源。
2. 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。
3. 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。

## 如何避免死锁？

避免死锁就是在资源分配时，借助于算法（比如银行家算法）对资源分配进行计算评估，使其进入安全状态。

**安全状态** 指的是系统能够按照某种线程推进顺序（`P1、P2、P3.....Pn`）来为每个线程分配所需资源，直到满足每个线程对资源的最大需求，使每个线程都可顺利完成。称 `<P1、P2、P3.....Pn>` 序列为安全序列。

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();
    }
    System.out.println(Thread.currentThread() + "waiting get
resource2");

    synchronized (resource2) {
        System.out.println(Thread.currentThread() + "get
resource2");
    }
}
}, "线程 2").start();

```

输出：

```

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 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。

## sleep() 方法和 wait() 方法对比

共同点：两者都可以暂停线程的执行。

区别：

- `sleep()` 方法没有释放锁，而 `wait()` 方法释放了锁。

- `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, 或者也可以使用 `wait(long timeout)` 超时后线程会自动苏醒。
- `sleep()` 是 `Thread` 类的静态本地方法, `wait()` 则是 `Object` 类的本地方法。为什么这样设计呢?

## 为什么 `wait()` 方法不定义在 `Thread` 中?

`wait()` 是让获得对象锁的线程实现等待, 会自动释放当前线程占有的对象锁。每个对象 ( `Object` ) 都拥有对象锁, 既然要释放当前线程占有的对象锁并让其进入 `WAITING` 状态, 自然是要操作对应的对象 ( `Object` ) 而非当前的线程 ( `Thread` ) 。

类似的问题: 为什么 `sleep()` 方法定义在 `Thread` 中?

因为 `sleep()` 是让当前线程暂停执行, 不涉及到对象类, 也不需要获得对象锁。

## 可以直接调用 `Thread` 类的 `run` 方法吗?

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。

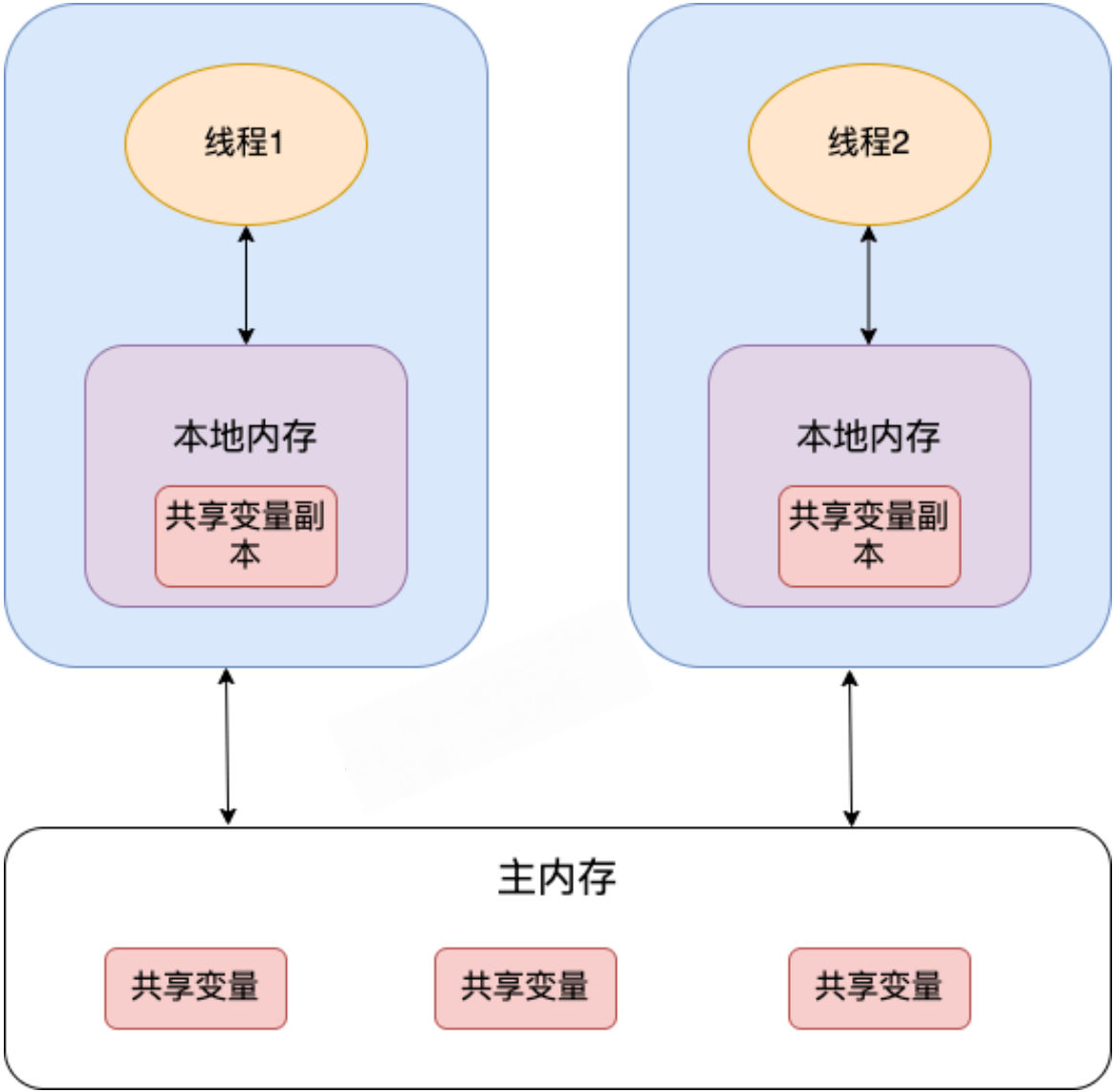
## JMM(Java Memory Model)

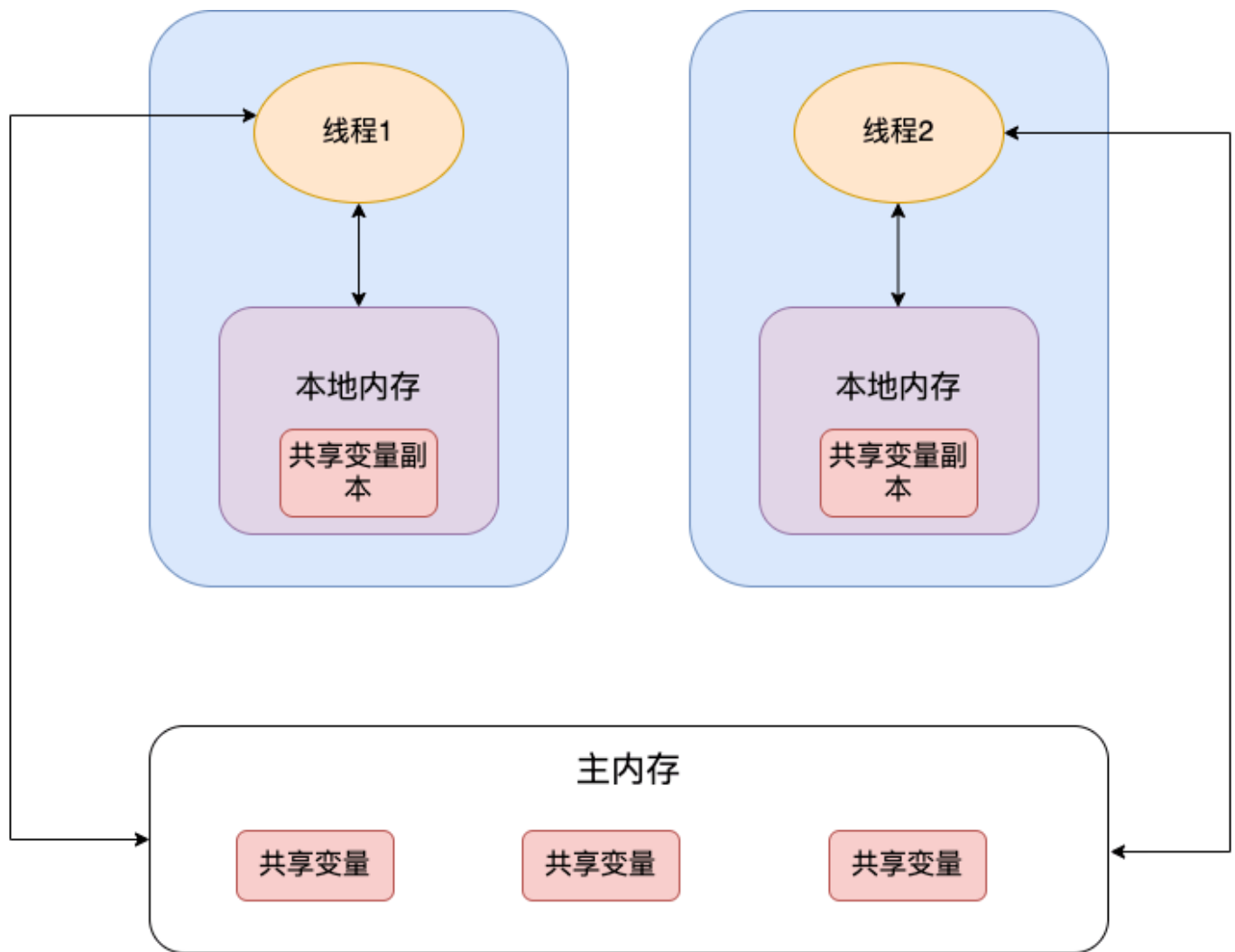
JMM (Java 内存模型) 相关的问题比较多, 也比较重要, 于是我单独抽了一篇文章来总结 JMM 相关的知识点和问题: [JMM \(Java 内存模型\) 详解](#)。

# volatile 关键字

## 如何保证变量的可见性？

在 Java 中，`volatile` 关键字可以保证变量的可见性，如果我们将变量声明为 `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。





`volatile` 关键字其实并非是 Java 语言特有的，在 C 语言里也有，它最原始的意义就是禁用 CPU 缓存。如果我们将一个变量使用 `volatile` 修饰，这就指示 编译器，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。

`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。

## 如何禁止指令重排序？

在 Java 中， `volatile` 关键字除了可以保证变量的可见性，还有一个重要的作用就是防止 JVM 的指令重排序。如果我们将变量声明为 `volatile` ，在对这个变量进行读写操作的时候，会通过插入特定的 内存屏障 的方式来禁止指令重排序。

在 Java 中， `Unsafe` 类提供了三个开箱即用的内存屏障相关的方法，屏蔽了操作系统底层的差异：

```
public native void loadFence();
public native void storeFence();
public native void fullFence();
```

理论上来说，你通过这三个方法也可以实现和 `volatile` 禁止重排序一样的效果，只是会麻烦一些。

下面我以一个常见的面试题为例讲解一下 `volatile` 关键字禁止指令重排序的效果。

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 getUniqueInstance() {
        //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;
    }
}
```

`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 `getUniqueInstance()` 后发现 `uniqueInstance` 不为空，因此返回 `uniqueInstance`，但此时 `uniqueInstance` 还未被初始化。

## volatile 可以保证原子性么？

`volatile` 关键字能保证变量的可见性，但不能保证对变量的操作是原子性的。

我们通过下面的代码即可证明：

```
/**
 * 微信搜 JavaGuide 回复"面试突击"即可免费领取个人原创的 Java 面试手册
 *
 * @author Guide哥
 * @date 2022/08/03 13:40
 **/

public class VolatoleAtomicityDemo {
    public volatile static int inc = 0;

    public void increase() {
        inc++;
    }

    public static void main(String[] args) throws InterruptedException {
        ExecutorService threadPool = Executors.newFixedThreadPool(5);
        VolatoleAtomicityDemo volatoleAtomicityDemo = new
VolatoleAtomicityDemo();
        for (int i = 0; i < 5; i++) {
            threadPool.execute(() -> {
                for (int j = 0; j < 500; j++) {
                    volatoleAtomicityDemo.increase();
                }
            });
        }
    }
}
```

```
// 等待1.5秒，保证上面程序执行完成
Thread.sleep(1500);
System.out.println(inc);
threadPool.shutdown();
}
}
```

正常情况下，运行上面的代码理应输出 2500。但你真正运行了上面的代码之后，你会发现每次输出结果都小于 2500。

为什么会出现这种情况呢？不是说好了，`volatile` 可以保证变量的可见性嘛！

也就是说，如果 `volatile` 能保证 `inc++` 操作的原子性的话。每个线程中对 `inc` 变量自增完之后，其他线程可以立即看到修改后的值。5 个线程分别进行了 500 次操作，那么最终 `inc` 的值应该是  $5 \times 500 = 2500$ 。

很多人会误认为自增操作 `inc++` 是原子性的，实际上，`inc++` 其实是一个复合操作，包括三步：

1. 读取 `inc` 的值。
2. 对 `inc` 加 1。
3. 将 `inc` 的值写回内存。

`volatile` 是无法保证这三个操作是具有原子性的，有可能导致下面这种情况出现：

1. 线程 1 对 `inc` 进行读取操作之后，还未对其进行修改。线程 2 又读取了 `inc` 的值并对其进行修改 (+1)，再将 `inc` 的值写回内存。
2. 线程 2 操作完毕后，线程 1 对 `inc` 的值进行修改 (+1)，再将 `inc` 的值写回内存。

这也就导致两个线程分别对 `inc` 进行了一次自增操作后，`inc` 实际上只增加了 1。

其实，如果想要保证上面的代码运行正确也非常简单，利用 `synchronized`、`Lock` 或者 `AtomicInteger` 都可以。

使用 `synchronized` 改进：

```
public synchronized void increase() {  
    inc++;  
}
```

使用 `AtomicInteger` 改进:

```
public AtomicInteger inc = new AtomicInteger();  
  
public void increase() {  
    inc.getAndIncrement();  
}
```

使用 `ReentrantLock` 改进:

```
Lock lock = new ReentrantLock();  
public void increase() {  
    lock.lock();  
    try {  
        inc++;  
    } finally {  
        lock.unlock();  
    }  
}
```

## synchronized 关键字

### 说一说自己对于 synchronized 关键字的了解

synchronized 翻译成中文是同步的意思，主要解决的是多个线程之间访问资源的同步性，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。

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

不过，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。

### 如何使用 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
2. 修饰静态方法
3. 修饰代码块

## 1、修饰实例方法（锁当前对象实例）

给当前对象实例加锁，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（锁当前类）

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**。

这是因为静态成员不属于任何一个实例对象，归整个类所有，不依赖于类的特定实例，被类的所有实例共享。

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

静态 `synchronized` 方法和非静态 `synchronized` 方法之间的调用互斥么？不互斥！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。

## 3、修饰代码块（锁指定对象/类）

对括号里指定的对象/类加锁：

- `synchronized(object)` 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 **给定 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 类上锁；
- `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。

## 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。

## 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。

### `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
    4: getstatic #2                // Field java/lang/System.out:Ljava/io/PrintStream;
    7: ldc #3                      // String Method 1 start
    9: invokevirtual #4            // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   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"

```

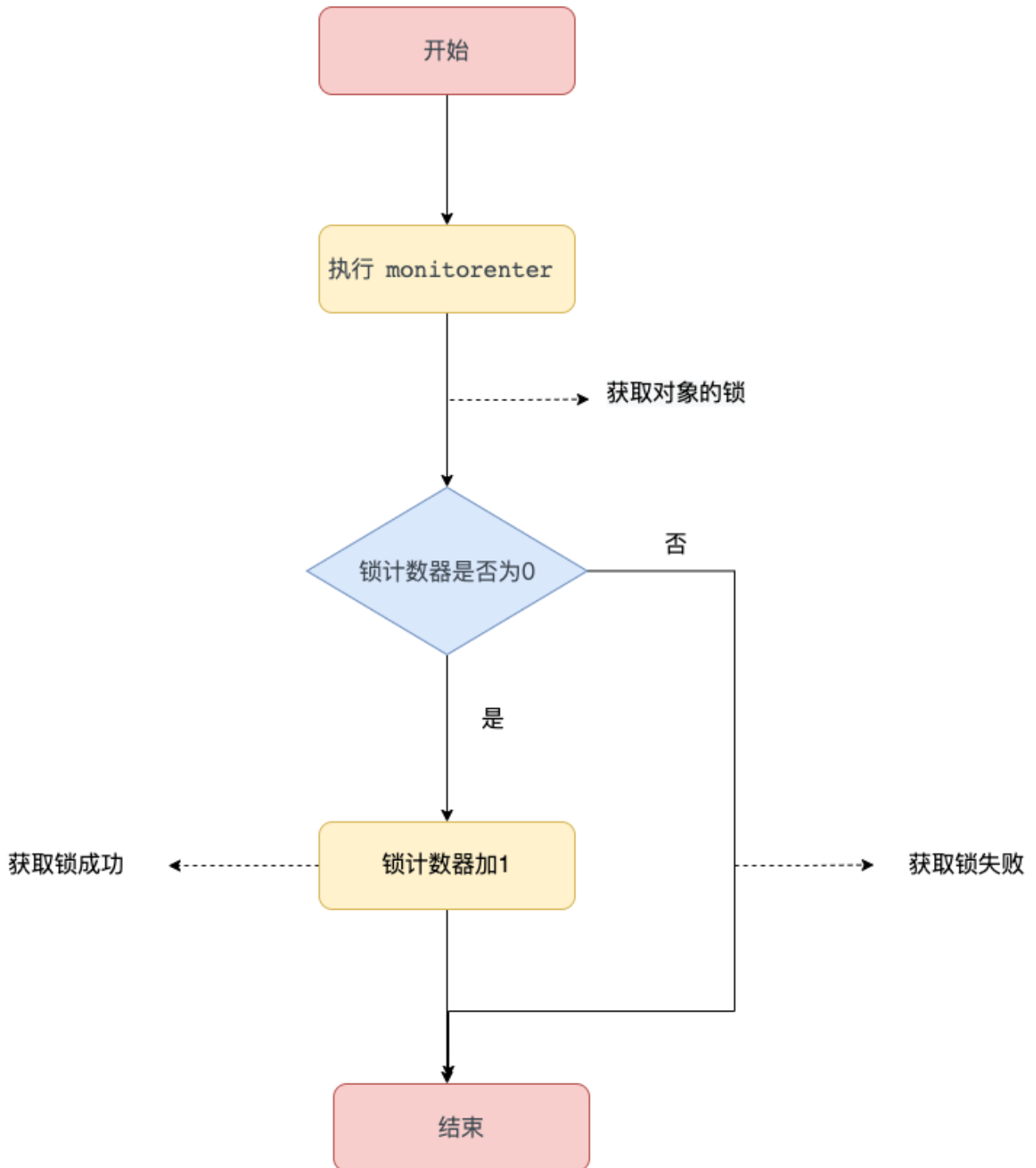
从上面我们可以看出：`synchronized` 同步语句块的实现使用的是 `monitorenter` 和 `monitorexit` 指令，其中 `monitorenter` 指令指向同步代码块的开始位置，`monitorexit` 指令则指明同步代码块的结束位置。

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。

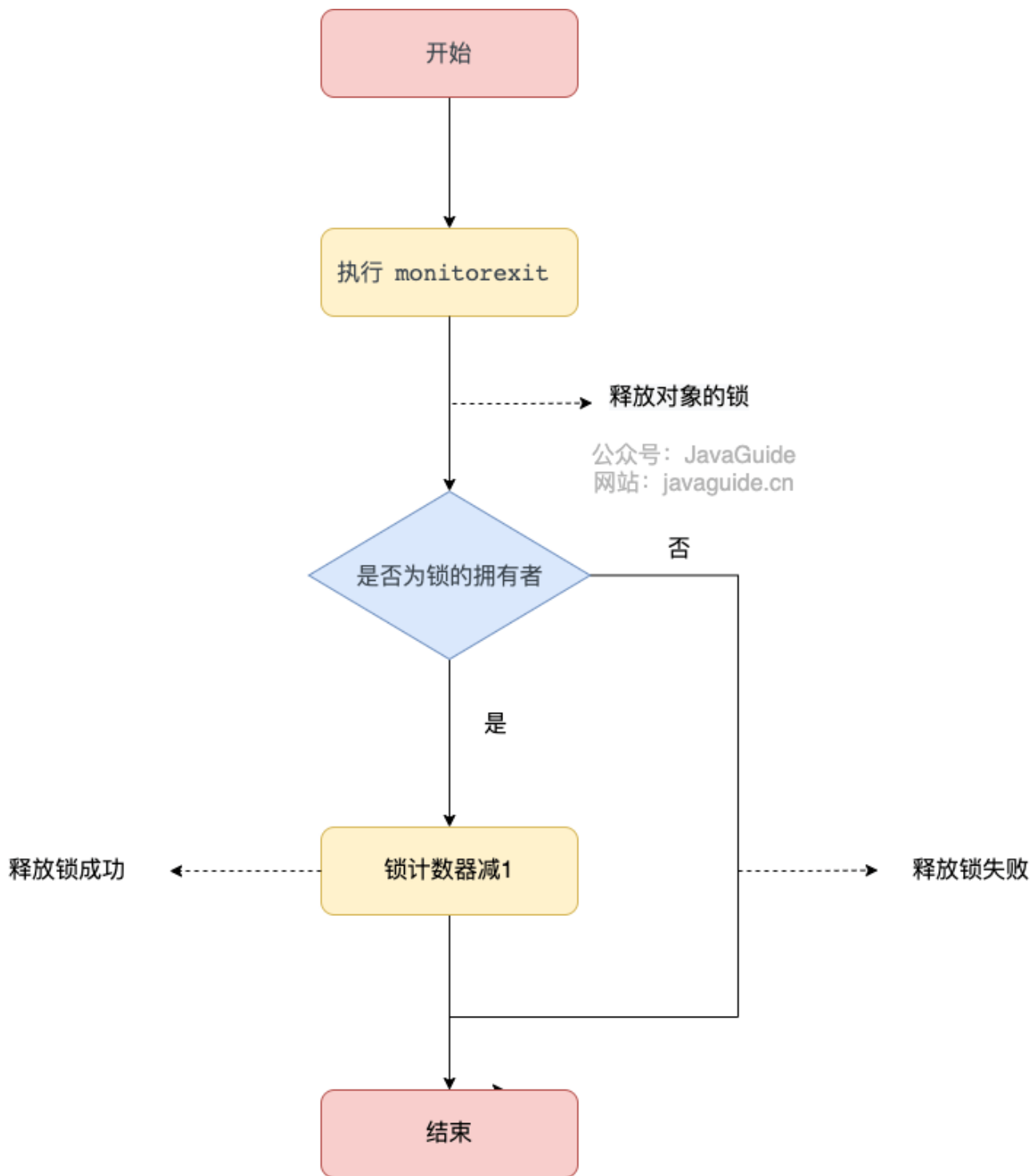
另外，`wait/notify` 等方法也依赖于 `monitor` 对象，这就是为什么只有在同步的块或者方法中才能调用 `wait/notify` 等方法，否则会抛出 `java.lang.IllegalMonitorStateException` 的异常的原因。

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。



对象锁的的拥有者线程才可以执行 `monitorexit` 指令来释放锁。在执行 `monitorexit` 指令后，将锁计数器设为 0，表明锁被释放，其他线程可以尝试获取锁。





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。

**synchronized** 修饰方法的的情况

```
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 钀堿瑙
      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。

如果是实例方法，JVM 会尝试获取实例对象的锁。如果是静态方法，JVM 会尝试获取当前 class 的锁。

## 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。

相关推荐：[Java 锁与线程的那些事 - 有赞技术团队](#)。

 进阶一下：学有余力的小伙伴可以抽时间详细研究一下对象监视器 `monitor`。

## JDK1.6 之后的 `synchronized` 关键字底层做了哪些优化？

JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

关于这几种优化的详细信息可以查看下面这篇文章：[Java6 及以上版本对 `synchronized` 的优化](#)

## `synchronized` 和 `volatile` 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。

## `synchronized` 和 `ReentrantLock` 的区别

两者都是可重入锁

“可重入锁”指的是自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果是不可重入锁的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增 1，所以要等到锁的计数器下降为 0 时才能释放锁。

`synchronized` 依赖于 JVM 而 `ReentrantLock` 依赖于 API

`synchronized` 是依赖于 JVM 实现的，前面我们也讲到了虚拟机团队在 JDK1.6 为 `synchronized` 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。`ReentrantLock` 是 JDK 层面实现的（也就是 API 层面，需要 `lock()` 和 `unlock()` 方法配合 `try/finally` 语句块来完成），所以我们可以查看它的源代码，来看它是如何实现的。

## ReentrantLock 比 synchronized 增加了一些高级功能

相比 `synchronized`，`ReentrantLock` 增加了一些高级功能。主要来说主要有三点：

- **等待可中断**：`ReentrantLock` 提供了一种能够中断等待锁的线程的机制，通过 `lock.lockInterruptibly()` 来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。
- **可实现公平锁**：`ReentrantLock` 可以指定是公平锁还是非公平锁。而 `synchronized` 只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。`ReentrantLock` 默认情况是非公平的，可以通过 `ReentrantLock` 类的 `ReentrantLock(boolean fair)` 构造方法来制定是否是公平的。
- **可实现选择性通知（锁可以绑定多个条件）**：`synchronized` 关键字与 `wait()` 和 `notify()` / `notifyAll()` 方法相结合可以实现等待/通知机制。`ReentrantLock` 类当然也可以实现，但是需要借助于 `Condition` 接口与 `newCondition()` 方法。

`Condition` 是 JDK1.5 之后才有的，它具有很好的灵活性，比如可以实现多路通知功能也就是在一个 `Lock` 对象中可以创建多个 `Condition` 实例（即对象监视器），线程对象可以注册在指定的 `Condition` 中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。在使用 `notify()/notifyAll()` 方法进行通知时，被通知的线程是由 JVM 选择的，用 `ReentrantLock` 类结合 `Condition` 实例可以实现“选择性通知”，这个功能非常重要，而且是 `Condition` 接口默认提供的。而 `synchronized` 关键字就相当于整个 `Lock` 对象中只有一个 `Condition` 实例，所有的线程都注册在它一个身上。如果执行 `notifyAll()` 方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而 `Condition` 实例的 `signalAll()` 方法只会唤醒注册在该 `Condition` 实例中的所有等待线程。

如果你想使用上述功能，那么选择 `ReentrantLock` 是一个不错的选择。性能已不是选择标准

## ThreadLocal

### ThreadLocal 有什么用？

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 中自带的 `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：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。

## 如何使用 `ThreadLocal`?

相信看了上面的解释，大家已经搞懂 `ThreadLocal` 类是个什么东西了。下面简单演示一下如何在项目中实际使用 `ThreadLocal`。

```
import java.text.SimpleDateFormat;
import java.util.Random;

public class ThreadLocalExample implements Runnable{

    // SimpleDateFormat 不是线程安全的，所以每个线程都要有自己独立的副本
    private static final ThreadLocal<SimpleDateFormat> formatter =
ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyyMMdd HHmm"));

    public static void main(String[] args) throws InterruptedException {
        ThreadLocalExample obj = new ThreadLocalExample();
        for(int i=0 ; i<10; i++){
            Thread t = new Thread(obj, "+" + i);
            Thread.sleep(new Random().nextInt(1000));
            t.start();
        }
    }

    @Override
    public void run() {
        System.out.println("Thread Name= " + Thread.currentThread().getName() + "
default Formatter = " + formatter.get().toPattern());
        try {
            Thread.sleep(new Random().nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```

    }

    //formatter pattern is changed here by thread, but it won't reflect to
    other threads

    formatter.set(new SimpleDateFormat());

    System.out.println("Thread Name= "+Thread.currentThread().getName()+"
formatter = "+formatter.get().toPattern());
    }

}

```

输出结果：

```

Thread Name= 0 default Formatter = yyyyMMdd HHmm
Thread Name= 0 formatter = yy-M-d ah:mm
Thread Name= 1 default Formatter = yyyyMMdd HHmm
Thread Name= 2 default Formatter = yyyyMMdd HHmm
Thread Name= 1 formatter = yy-M-d ah:mm
Thread Name= 3 default Formatter = yyyyMMdd HHmm
Thread Name= 2 formatter = yy-M-d ah:mm
Thread Name= 4 default Formatter = yyyyMMdd HHmm
Thread Name= 3 formatter = yy-M-d ah:mm
Thread Name= 4 formatter = yy-M-d ah:mm
Thread Name= 5 default Formatter = yyyyMMdd HHmm
Thread Name= 5 formatter = yy-M-d ah:mm
Thread Name= 6 default Formatter = yyyyMMdd HHmm
Thread Name= 6 formatter = yy-M-d ah:mm
Thread Name= 7 default Formatter = yyyyMMdd HHmm
Thread Name= 7 formatter = yy-M-d ah:mm
Thread Name= 8 default Formatter = yyyyMMdd HHmm
Thread Name= 9 default Formatter = yyyyMMdd HHmm
Thread Name= 8 formatter = yy-M-d ah:mm
Thread Name= 9 formatter = yy-M-d ah:mm

```

从输出中可以看出，虽然 Thread-0 已经改变了 formatter 的值，但 Thread-1 默认格式化值与初始值相同，其他线程也一样。

上面有一段代码用到了创建 `ThreadLocal` 变量的那段代码用到了 Java8 的知识，它等于下面这段代码，如果你写了下面这段代码的话，IDEA 会提示你转换为 Java8 的格式(IDEA 真的不错！)。因为 `ThreadLocal` 类在 Java 8 中扩展，使用一个新的方法 `withInitial()`，将 `Supplier` 功能接口作为参数。

```
private static final ThreadLocal<SimpleDateFormat> formatter = new
ThreadLocal<SimpleDateFormat>(){
    @Override
    protected SimpleDateFormat initialValue(){
        return new SimpleDateFormat("yyyyMMdd HHmm");
    }
};
```

## ThreadLocal 原理了解吗？

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。`ThreadLocal` 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。

每个 `Thread` 中都具备一个 `ThreadLocalMap`，而 `ThreadLocalMap` 可以存储以 `ThreadLocal` 为 key，Object 对象为 value 的键值对。

```

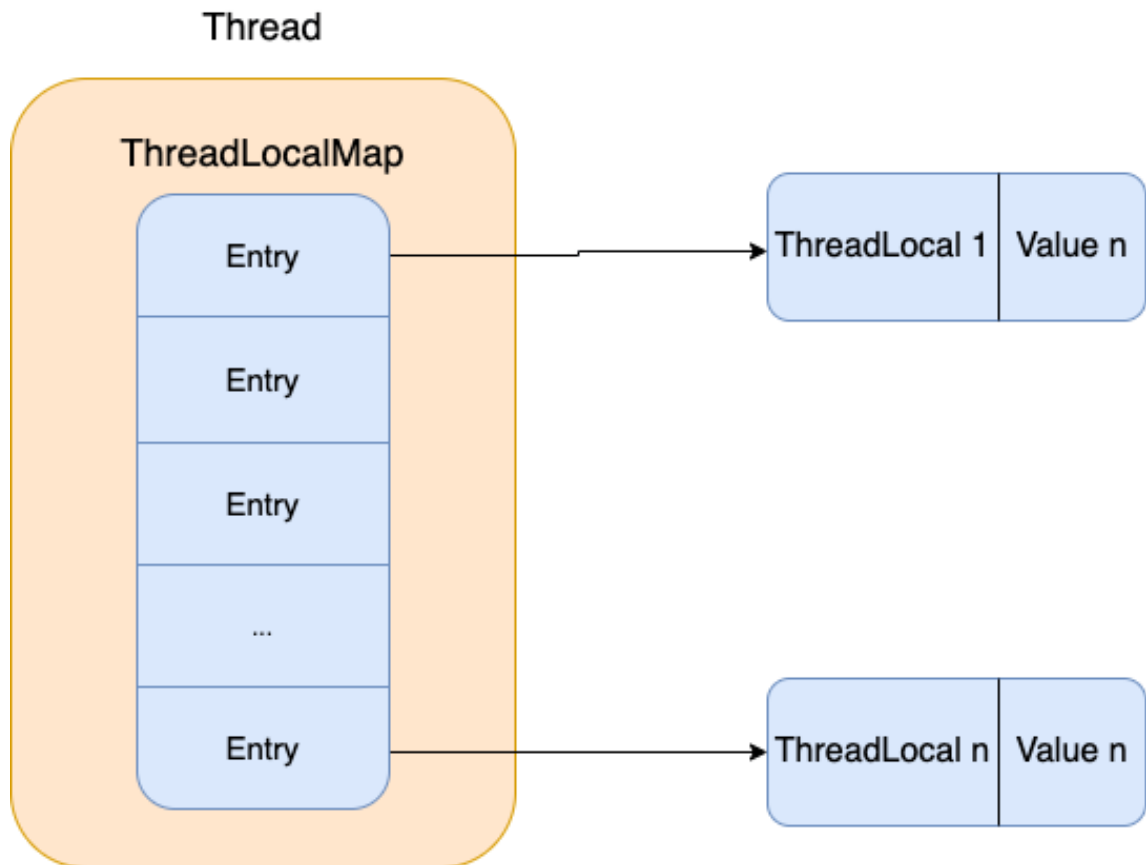
ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
    //.....
}

```

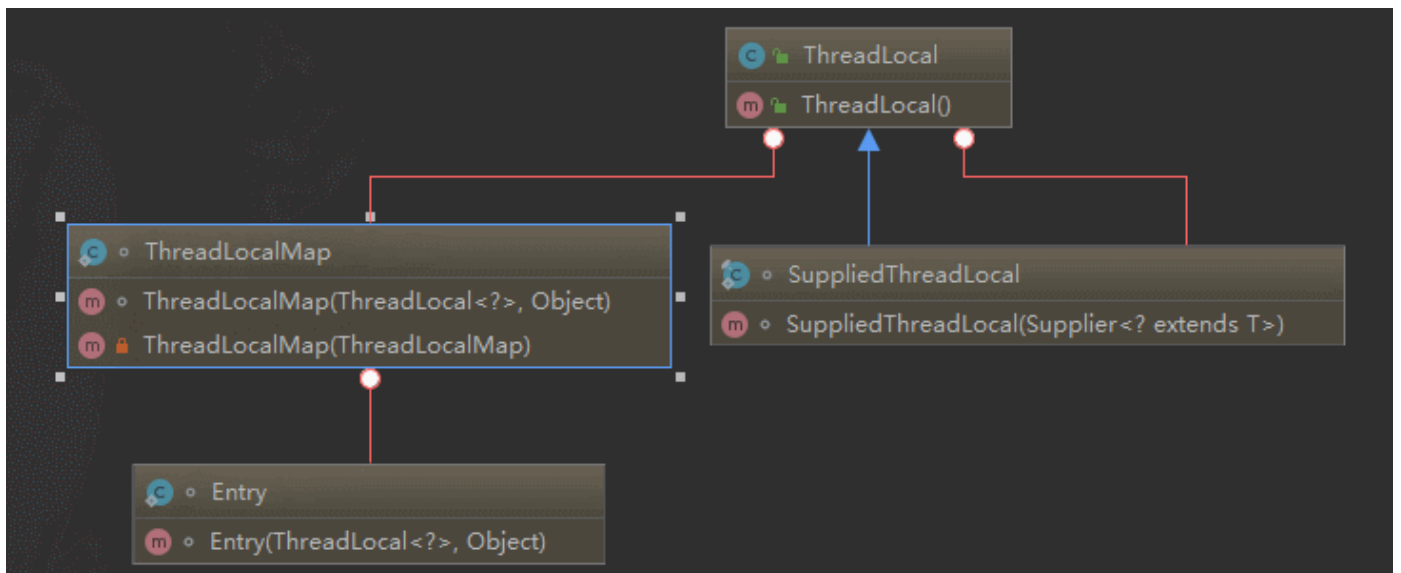
比如我们在同一个线程中声明了两个 `ThreadLocal` 对象的话，`Thread` 内部都是使用仅有的那个 `ThreadLocalMap` 存放数据的，`ThreadLocalMap` 的 key 就是 `ThreadLocal` 对象，value 就是 `ThreadLocal` 对象调用 `set` 方法设置的值。

`ThreadLocal` 数据结构如下图所示：





`ThreadLocalMap` 是 `ThreadLocal` 的静态内部类。



## ThreadLocal 内存泄露问题是怎么导致的？

`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 不会被清理掉。

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。

## 线程池

线程池相关的知识点和面试题总结请看这篇文章：[Java 线程池详解](#)（由于内容比较多就不放在 PDF 里面了）。

## AQS

AQS 相关的知识点和面试题总结请看这篇文章：[AQS 详解](#)（由于内容比较多就不放在 PDF 里面了）。