

Table of Contents

Introduction	1.1
Java/J2EE 基础	1.2
Java 基础知识回顾	1.2.1
J2EE 基础知识回顾	1.2.2
static、final、this、super关键字总结	1.2.3
static 关键字详解	1.2.4
Java 集合框架	1.3
这几道Java集合框架面试题几乎必问	1.3.1
Java 集合框架常见面试题总结	1.3.2
ArrayList 源码学习	1.3.3
【面试必备】透过源码角度一步一步带你分析 ArrayList 扩容机制	1.3.4
LinkedList 源码学习	1.3.5
HashMap(JDK1.8)源码学习	1.3.6
Java 多线程	1.4
多线程系列文章	1.4.1
值得立马保存的 synchronized 关键字总结	1.4.2
Java IO 与 NIO	1.5
Java IO 与 NIO系列文章	1.5.1
Java虚拟机 (jvm)	1.6
可能是把Java内存区域讲的最清楚的一篇文章	1.6.1
搞定JVM垃圾回收就是这么简单	1.6.2
Java虚拟机 (jvm) 学习与面试	1.6.3
设计模式	1.7
设计模式系列文章	1.7.1
数据结构	1.8
数据结构知识学习与面试	1.8.1
算法	1.9
算法学习与面试	1.9.1
常见安全算法（MD5、SHA1、Base64等等）总结	1.9.2
算法总结——几道常见的子串算法题	1.9.3
算法总结——几道常见的链表算法题	1.9.4
网络相关	1.10
计算机网络常见面试题	1.10.1
计算机网络基础知识总结	1.10.2
数据通信(RESTful、RPC、消息队列)	1.11
数据通信(RESTful、RPC、消息队列)相关知识点总结	1.11.1
Linux相关	1.12

后端程序员必备的 Linux 基础知识	1.12.1
主流框架/软件	1.13
Spring	1.13.1
Spring 学习与面试	1.13.1.1
Spring中bean的作用域与生命周期	1.13.1.2
ZooKeeper	1.13.2
可能是把 ZooKeeper 概念讲的最清楚的一篇文章	1.13.2.1
数据存储	1.14
MySQL	1.14.1
MySQL 学习与面试	1.14.1.1
Redis	1.14.2
Redis 总结	1.14.2.1
Redlock分布式锁	1.14.2.2
如何做可靠的分布式锁, Redlock真的可行么	1.14.2.3
春夏秋冬又一春之Redis持久化	1.14.2.4
架构	1.15
分布式相关	1.15.1
分布式学习与面试	1.15.1.1
面试必备	1.16
面试必备知识点	1.16.1
面试必备之乐观锁与悲观锁	1.16.1.1
最最最常见的Java面试题总结	1.16.2
第一周 (2018-8-7)	1.16.2.1
第二周 (2018-8-13)	1.16.2.2
第三周 (2018-08-22)	1.16.2.3
第四周(2018-8-30)	1.16.2.4
程序员如何写简历	1.16.3
程序员的简历之道	1.16.3.1
手把手教你用Markdown写一份高质量的简历	1.16.3.2
其他	1.17
个人书单推荐	1.17.1
技术方向选择	1.17.2

Java学习指南：一份涵盖大部分Java程序员所需要掌握的核心知识，正在一步一步慢慢完善，期待您的参与。

I	II	III	IV	V	VI	VII	VIII	IX	X
Java	数据结构与算法	计算机网络与数据通信	操作系统	主流框架	数据存储	架构	面试必备	其他	说明

:coffee: Java

- **Java/J2EE 基础**

- [Java 基础知识回顾](#)
- [J2EE 基础知识回顾](#)
- [static、final、this、super关键字总结](#)
- [static 关键字详解](#)

- **Java 集合框架**

- [这几道Java集合框架面试题几乎必问](#)
- [Java 集合框架常见面试题总结](#)
- [ArrayList 源码学习](#)
- [【面试必备】透过源码角度一步一步带你分析 ArrayList 扩容机制](#)
- [LinkedList 源码学习](#)
- [HashMap\(JDK1.8\)源码学习](#)

- **Java 多线程**

- [多线程系列文章](#)
- [值得立马保存的 synchronized 关键字总结](#)

- **Java IO 与 NIO**

- [Java IO 与 NIO系列文章](#)

- **Java虚拟机 (jvm)**

- [可能是把Java内存区域讲的最清楚的一篇文章](#)
- [搞定JVM垃圾回收就是这么简单](#)
- [Java虚拟机 \(jvm\) 学习与面试](#)

- **设计模式**

- [设计模式系列文章](#)

:open_file_folder: 数据结构与算法

- **数据结构**

- [数据结构知识学习与面试](#)

- **算法**

- 算法学习与面试
- 常见安全算法（MD5、SHA1、Base64等等）总结
- 算法总结——几道常见的子字符串算法题
- 算法总结——几道常见的链表算法题

:computer: 计算机网络与数据通信

- 网络相关
 - 计算机网络常见面试题
 - 计算机网络基础知识总结
- 数据通信(RESTful、RPC、消息队列)
 - 数据通信(RESTful、RPC、消息队列)相关知识点总结.md)

:iphone: 操作系统

- Linux相关
 - 后端程序员必备的 Linux 基础知识

:pencil2: 主流框架/软件

- Spring
 - Spring 学习与面试
 - Spring中bean的作用域与生命周期
- ZooKeeper
 - 可能是把 ZooKeeper 概念讲的最清楚的一篇文章

:floppy_disk: 数据存储

- MySQL
 - MySQL 学习与面试
- Redis
 - Redis 总结
 - Redlock分布式锁
 - 如何做可靠的分布式锁，Redlock真的可行么\
 - 春夏秋冬又一春之Redis持久化

:punch: 架构

- 分布式相关

- 分布式学习与面试

:musical_note: 面试必备

- 面试必备知识点

- 面试必备之乐观锁与悲观锁

- 最最最常见的Java面试题总结

这里会分享一些出现频率极其极高的面试题，初定周更一篇，什么时候更完什么时候停止。

- 第一周 (2018-8-7) (值传递和引用传递、==与equals、hashCode与equals)
- 第二周 (2018-8-13) .md)(String和StringBuffer、StringBuilder的区别是什么？String为什么是不可变的？什么是反射机制？反射机制的应用场景有哪些？.....)
- 第三周 (2018-08-22) (ArrayList 与 LinkedList 异同、ArrayList 与 Vector 区别、HashMap 的底层实现、HashMap 和 Hashtable 的区别、HashMap 的长度为什么是2的幂次方、HashSet 和 HashMap 区别、ConcurrentHashMap 和 Hashtable 的区别、ConcurrentHashMap 线程安全的具体实现方式/底层具体实现、集合框架底层数据结构总结)
- 第四周(2018-8-30).md.md (主要内容是几道面试常问的多线程基础题。)

- 程序员如何写简历

- 程序员的简历之道
 - 手把手教你用Markdown写一份高质量的简历

:art: 其他

- 个人书单推荐

- 个人阅读书籍清单

- 技术方向选择

- 选择技术方向都要考虑哪些因素

:envelope: 该开源文档一些说明

介绍

该文档主要是笔主在学习Java的过程中的一些学习笔记，但是为了能够涉及到大部分后端学习所需的技术知识点我也会偶尔引用一些别人的优秀文章的链接。该文档涉及的主要内容包括：Java、数据结构与算法、计算机网络与数据通信、操作系统、主流框架、数据存储、架构、面试必备知识点等等。相信不论你是前端还是后端都能在这份文档中收获到东西。

关于转载

如果需要引用到本仓库的一些东西，必须注明转载地址！！！毕竟大多都是手敲的，或者引用的是我的原创文章，希望大家尊重一下作者的劳动:smiley::smiley::smiley:！

如何对该开源文档进行贡献

1. 笔记内容大多是手敲，所以难免会有笔误。
2. 你对其他知识点的补充。

为什么要写这个开源文档？

在我们学习Java的时候，很多人会面临我不知道继续学什么或者面试会问什么的尴尬情况（我本人之前就很迷茫:smile:）。所以，我决定通过这个开源平台来帮助一些有需要的人，通过下面的内容，你会掌握系统的Java学习以及面试的相关知识。本来是想通过Gitbook的形式来制作的，后来想了想觉得可能有点大题小做 :grin:。另外，我自己一个人的力量毕竟有限，希望各位有想法的朋友可以提issue。

最后

本人会利用业余时间一直更新下去，目前还有很多地方不完善，一些知识点我会原创总结，还有一些知识点如果说网上有比较好的文章了，我会把这些文章加入进去。您也可以关注我的微信公众号：“Java面试通关手册”，我会在这里分享一些自己的原创文章。另外该文档格式参考：[Github Markdown格式](#)，表情素材来自：[EMOJI CHEAT SHEET](#)。

你若盛开，清风自来。欢迎关注我的微信公众号：“Java面试通关手册”，一个有温度的微信公众号。公众号有大量资料，回复关键字“1”你可能看到想要的东西哦！：



- 1. 面向对象和面向过程的区别
 - 面向过程
 - 面向对象
- 2. Java 语言有哪些特点?
- 3. 什么是 JDK? 什么是 JRE? 什么是 JVM? 三者之间的联系与区别
- 4. 什么是字节码? 采用字节码的最大好处是什么?
 - 先看下 java 中的编译器和解释器:
 - 采用字节码的好处:
- 5. Java 和 C++ 的区别
- 6. 什么是 Java 程序的主类? 应用程序和小程序的主类有何不同?
- 7. Java 应用程序与小程序之间有那些差别?
- 8. 字符型常量和字符串常量的区别
- 9. 构造器 Constructor 是否可被 override
- 10. 重载和重写的区别
- 11. Java 面向对象编程三大特性: 封装、继承、多态
 - 封装
 - 继承
 - 多态
- 12. String 和 StringBuffer、StringBuilder 的区别是什么? String 为什么是不可变的?
- 13. 自动装箱与拆箱
- 14. 在一个静态方法内调用一个非静态成员为什么是非法的?
- 15. 在 Java 中定义一个不做事且没有参数的构造方法的作用
- 16. import java 和 javax 有什么区别
- 17. 接口和抽象类的区别是什么?
- 18. 成员变量与局部变量的区别有那些?
- 19. 创建一个对象用什么运算符? 对象实体与对象引用有何不同?
- 20. 什么是方法的返回值? 返回值在类的方法里的作用是什么?
- 21. 一个类的构造方法的作用是什么? 若一个类没有声明构造方法, 该程序能正确执行吗? 为什么?
- 22. 构造方法有哪些特性?
- 23. 静态方法和实例方法有何不同?
- 24. 对象的相等与指向他们的引用相等, 两者有什么不同?
- 25. 在调用子类构造方法之前会先调用父类没有参数的构造方法, 其目的是?
- 26. == 与 equals(重要)
- 27. hashCode 与 equals (重要)
 - hashCode () 介绍
 - 为什么要写 hashCode
 - hashCode () 与 equals () 的相关规定
- 28. Java 中的值传递和引用传递
- 29. 简述线程, 程序、进程的基本概念。以及他们之间关系是什么?
- 30. 线程有哪些基本状态? 这些状态是如何定义的?
- 31. 关于 final 关键字的一些总结
- Java 基础学习书籍推荐

1. 面向对象和面向过程的区别

面向过程

优点: 性能比面向对象高, 因为类调用时需要实例化, 开销比较大, 比较消耗资源; 比如单片机、嵌入式开发、Linux/Unix 等一般采用面向过程开发, 性能是最重要的因素。

缺点: 没有面向对象易维护、易复用、易扩展

面向对象

优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护

缺点：性能比面向过程低

2. Java 语言有哪些特点？

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

3. 什么是 JDK？什么是 JRE？什么是 JVM？三者之间的联系与区别

这几个是Java中很基本很基本的东西，但是我相信一定还有很多人搞不清楚！为什么呢？因为我们大多数时候在使用现成的编译工具以及环境的时候，并没有去考虑这些东西。

JDK：顾名思义它是给开发者提供的开发工具箱，是给程序开发者用的。它除了包括完整的JRE（Java Runtime Environment），Java运行环境，还包含了其他供开发者使用的工具包。

JRE：普通用户只需要安装 JRE（Java Runtime Environment）来运行 Java 程序。而程序开发者必须安装JDK来编译、调试程序。

JVM：当我们运行一个程序时，JVM 负责将字节码转换为特定机器代码，JVM 提供了内存管理/垃圾回收和安全机制等。这种独立于硬件和操作系统，正是 java 程序可以一次编写多处执行的原因。

区别与联系：

1. JDK 用于开发，JRE 用于运行java程序；
2. JDK 和 JRE 中都包含 JVM；
3. JVM 是 java 编程语言的核心并且具有平台独立性。

4. 什么是字节码？采用字节码的最大好处是什么？

先看下 java 中的编译器和解释器：

Java 中引入了虚拟机的概念，即在机器和编译程序之间加入了一层抽象的虚拟的机器。这台虚拟的机器在任何平台上都提供给编译程序一个共同的接口。

编译程序只需要面向虚拟机，生成虚拟机能够理解的代码，然后由解释器来将虚拟机代码转换为特定系统的机器码执行。在 Java 中，这种供虚拟机理解的代码叫做 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。

每一种平台的解释器是不同的，但是实现的虚拟机是相同的。Java 源程序经过编译器编译后变成字节码，字节码由虚拟机解释执行，虚拟机将每一条要执行的字节码送给解释器，解释器将其翻译成特定机器上的机器码，然后在特定的机器上运行。这也就是解释了 Java 的编译与解释并存的特点。

Java 源代码--->编译器--->jvm 可执行的 Java 字节码(即虚拟指令)--->jvm--->jvm 中解释器--->机器可执行的二进制机器码--->程序运行。

采用字节码的好处：

Java 语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以 Java 程序运行时比较高效，而且，由于字节码并不专对一种特定的机器，因此，Java 程序无须重新编译便可在多种不同的计算机上运行。

5. Java和C++的区别

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

- 都是面向对象的语言，都支持封装、继承和多态
- Java 不提供指针来直接访问内存，程序内存更加安全
- Java 的类是单继承的，C++ 支持多重继承；虽然 Java 的类不可以多继承，但是接口可以多继承。
- Java 有自动内存管理机制，不需要程序员手动释放无用内存

6. 什么是 Java 程序的主类？应用程序和小程序的主类有何不同？

一个程序中可以有多个类，但只能有一个类是主类。在 Java 应用程序中，这个主类是指包含 main () 方法的类。而在 Java 小程序中，这个主类是一个继承自系统类 JApplet 或 Applet 的子类。应用程序的主类不一定要求是 public 类，但小程序的主类要求必须是 public 类。主类是 Java 程序执行的入口点。

7. Java 应用程序与小程序之间有那些差别？

简单说应用程序是从主线程启动(也就是 main() 方法)。applet 小程序没有 main 方法，主要是嵌在浏览器页面上运行(调用 init() 线程或者 run() 来启动)，嵌入浏览器这点跟 flash 的小游戏类似。

8. 字符型常量和字符串常量的区别

1. 形式上：字符常量是单引号引起的一个字符 字符串常量是双引号引起的若干个字符
2. 含义上：字符常量相当于一个整形值(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(至少一个字符结束标志)(注意： char 在 Java 中占两个字节)

java编程思想第四版：2.2.2节

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

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

9. 构造器 Constructor 是否可被 override

在讲继承的时候我们就知道父类的私有属性和构造方法并不能被继承，所以 Constructor 也就不能被 override,但是可以 overload,所以你可以看到一个类中有多个构造函数的情况。

10. 重载和重写的区别

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

重写：发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，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；如果父类方法访问修饰符为 private 则子类就不能重写该方法。

11. Java 面向对象编程三大特性:封装、继承、多态

封装

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

继承

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

关于继承如下 3 点请记住：

1. 子类拥有父类非 private 的属性和方法。
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

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

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

12. String 和 StringBuffer、StringBuilder 的区别是什么？String 为什么是不可变的？

可变性

简单的来说：String 类中使用 final 关键字字符数组保存字符串，`private final char value[]`，所以 String 对象是不可变的。而 StringBuilder 与 StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中也是使用字符数组保存字符串 `char[] value` 但是没有用 final 关键字修饰，所以这两种对象都是可变的。

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

AbstractStringBuilder.java

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {
    char[] value;
    int count;
    AbstractStringBuilder() {
    }
    AbstractStringBuilder(int capacity) {
        value = new char[capacity];
    }
}
```

线程安全性

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

性能

每次对 String 类型进行改变的时候，都会生成一个新的 String 对象，然后将指针指向新的 String 对象。StringBuffer 每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用 StirngBuilder 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 = String
2. 单线程操作字符串缓冲区下操作大量数据 = StringBuilder
3. 多线程操作字符串缓冲区下操作大量数据 = StringBuffer

13. 自动装箱与拆箱

装箱：将基本类型用它们对应的引用类型包装起来；

拆箱：将包装类型转换为基本数据类型；

14. 在一个静态方法内调用一个非静态成员为什么是非法的？

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

15. 在 Java 中定义一个不做事且没有参数的构造方法的作用

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

16. import java和javax有什么区别

刚开始的时候 Java API 所必需的包是 java 开头的包，javax 当时只是扩展 API 包来说使用。然而随着时间的推移，javax 逐渐的扩展成为 Java API 的组成部分。但是，将扩展从 javax 包移动到 java 包将是太麻烦了，最终会破坏一堆现有的代码。因此，最终决定 javax 包将成为标准 API 的一部分。

所以，实际上 java 和 javax 没有区别。这都是一个名字。

17. 接口和抽象类的区别是什么？

1. 接口的方法默认是 public，所有方法在接口中不能有实现，抽象类可以有非抽象的方法
2. 接口中的实例变量默认是 final 类型的，而抽象类中则不一定
3. 一个类可以实现多个接口，但最多只能实现一个抽象类
4. 一个类实现接口的话要实现接口的所有方法，而抽象类不一定
5. 接口不能用 new 实例化，但可以声明，但是必须引用一个实现该接口的对象 从设计层面来说，抽象是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

18. 成员变量与局部变量的区别有哪些？

1. 从语法形式上，看成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被 public, private, static 等修饰符所修饰，而局部变量不能被访问控制修饰符及 static 所修饰；但是，成员变量和局部变量都能被 final 所修饰；
2. 从变量在内存中的存储方式来看，成员变量是对象的一部分，而对象存在于堆内存，局部变量存在于栈内存
3. 从变量在内存中的生存时间上看，成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动消失。
4. 成员变量如果没有被赋初值，则会自动以类型的默认值而赋值（一种情况例外被 final 修饰但没有被 static 修饰的成员变量必须显示地赋值）；而局部变量则不会自动赋值。

19. 创建一个对象用什么运算符？对象实体与对象引用有何不同？

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

20. 什么是方法的返回值？返回值在类的方法里的作用是什么？

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

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

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

22. 构造方法有哪些特性？

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

23. 静态方法和实例方法有何不同？

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

24. 对象的相等与指向他们的引用相等，两者有什么不同？

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

25. 在调用子类构造方法之前会先调用父类没有参数的构造方法，其目的是？

帮助子类做初始化工作。

26. == 与 equals(重要)

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

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

- 情况1：类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时，等价于通过“==”比较这两个对象。
- 情况2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来两个对象的内容相等；若它们的内容相等，则返回 true（即，认为这两个对象相等）。

举个例子：

```
public class test1 {
    public static void main(String[] args) {
        String a = new String("ab"); // a 为一个引用
        String b = new String("ab"); // b 为另一个引用, 对象的内容一样
```

```

String aa = "ab"; // 放在常量池中
String bb = "ab"; // 从常量池中查找
if (aa == bb) // true
    System.out.println("aa==bb");
if (a == b) // false, 非同一对象
    System.out.println("a==b");
if (a.equals(b)) // true
    System.out.println("aEQb");
if (42 == 42.0) { // true
    System.out.println("true");
}
}
}

```

说明：

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

27. hashCode 与 equals (重要)

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

hashCode () 介绍

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个 int 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义在 JDK 的 Object.java 中，这就意味着 Java 中的任何类都包含有 hashCode() 函数。

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：

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

hashCode () 与 equals () 的相关规定

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

28. Java 中的值传递和引用传递

值传递是指对象被值传递，意味着传递了对象的一个副本，即使副本被改变，也不会影响源对象。（因为值传递的时候，实际上是将实参的值复制一份给形参。）

引用传递是指对象被引用传递，意味着传递的并不是实际的对象，而是对象的引用。因此，外部对引用对象的改变会反映到所有的对象上。（因为引用传递的时候，实际上是将实参的地址值复制一份给形参。）

29. 简述线程、程序、进程的基本概念。以及他们之间关系是什么？

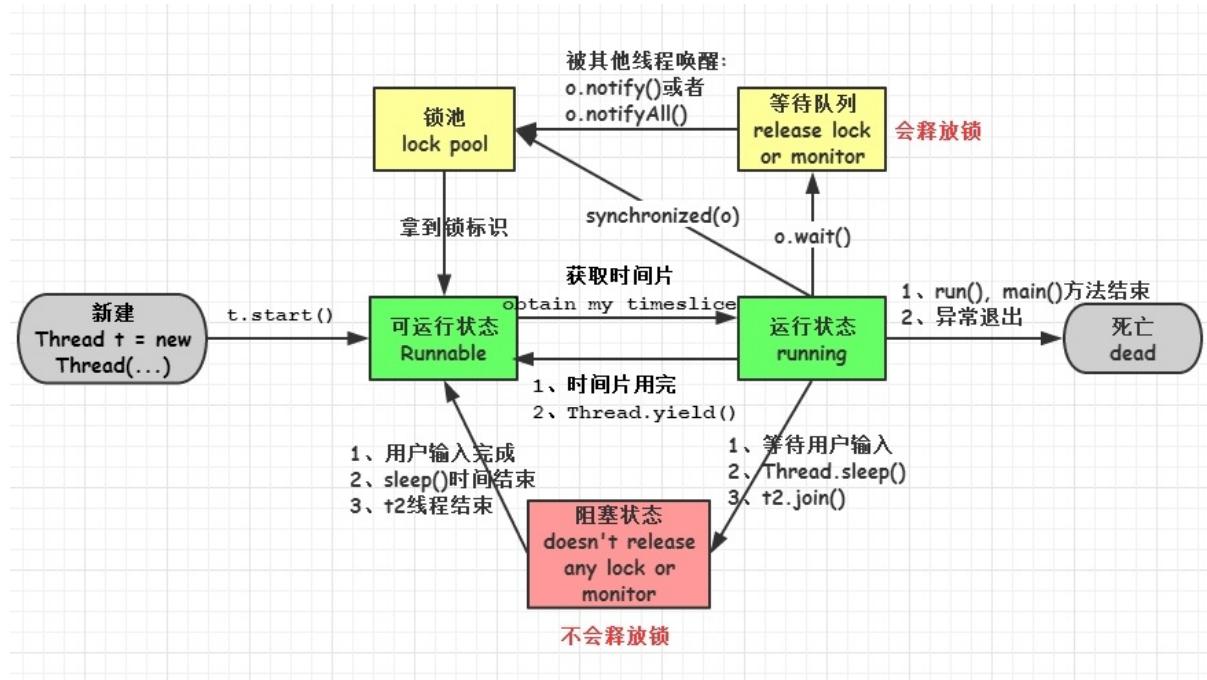
线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

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

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。简单来说，一个进程就是一个执行中的程序，它在计算机中一个指令接着一个指令地执行着，同时，每个进程还占有某些系统资源如CPU时间，内存空间，文件，文件，输入输出设备的使用权等等。换句话说，当程序在执行时，将被操作系统载入内存中。线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。从另一角度来说，进程属于操作系统的范畴，主要是同一段时间内，可以同时执行一个以上的程序，而线程则是在同一程序内几乎同时执行一个以上的程序段。

30. 线程有哪些基本状态？这些状态是如何定义的？

1. **新建(new)**: 新创建了一个线程对象。
2. **可运行(runnable)**: 线程对象创建后，其他线程(比如main线程) 调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu的使用权。
3. **运行(running)**: 可运行状态(runnable)的线程获得了cpu时间片 (timeslice)，执行程序代码。
4. **阻塞(block)**: 阻塞状态是指线程因为某种原因放弃了cpu使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice转到运行(running)状态。阻塞的情况分三种：
(一). 等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waitting queue)中。
(二). 同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。
(三). 其他阻塞：运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。
5. **死亡(dead)**: 线程run()、main()方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。



备注：可以用早起坐地铁来比喻这个过程：

还没起床：sleeping

起床收拾好了，随时可以坐地铁出发：Runnable

等地铁来：Waiting

地铁来了，但要排队上地铁：I/O阻塞

上了地铁，发现暂时没座位：synchronized阻塞

地铁上找到座位：Running

到达目的地：Dead

31 关于 final 关键字的一些总结

final关键字主要用在三个地方：变量、方法、类。

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

Java基础学习书籍推荐

《Head First Java.第二版》：可以说是我自己的 Java 启蒙书籍了，特别适合新手读当然也适合我们用来温故Java知识点。

《Java核心技术卷1+卷2》：很棒的两本书，建议有点 Java 基础之后再读，介绍的还是比较深入的，非常推荐。

《Java编程思想(第4版)》： 这本书要常读，初学者可以快速概览，中等程序员可以深入看看 Java，老鸟还可以用之回顾 Java 的体系。这本书之所以厉害，因为它在无形中整合了设计模式，这本书之所以难读，也恰恰在于他对设计模式的整合是无形的。

- [Servlet总结](#)
- [阐述Servlet和CGI的区别?](#)
 - [CGI的不足之处:](#)
 - [Servlet的优点:](#)
- [Servlet接口中有哪些方法及Servlet生命周期探秘](#)
- [get和post请求的区别](#)
- [什么情况下调用doGet\(\)和doPost\(\)](#)
- [转发（Forward）和重定向（Redirect）的区别](#)
- [自动刷新\(Refresh\)](#)
- [Servlet与线程安全](#)
- [JSP和Servlet是什么关系](#)
- [JSP工作原理](#)
- [JSP有哪些内置对象、作用分别是什么](#)
- [Request对象的主要方法有哪些](#)
- [request.getAttribute\(\)和 request.getParameter\(\)有何区别](#)
- [include指令include的行为的区别](#)
- [JSP九大内置对象，七大动作，三大指令](#)
- [讲解JSP中的四种作用域](#)
- [如何实现JSP或Servlet的单线程模式](#)
- [实现会话跟踪的技术有哪些](#)
- [Cookie和Session的区别](#)

Servlet总结

在Java Web程序中，**Servlet**主要负责接收用户请求**HttpServletRequest**，在**doGet(),doPost()**中做相应的处理，并将响应**HttpServletResponse**反馈给用户。Servlet可以设置初始化参数，供Servlet内部使用。一个Servlet类只会有一个实例，在它初始化时调用**init()**方法，销毁时调用**destroy()**方法。**Servlet**需要在**web.xml**中配置（MyEclipse中创建Servlet会自动配置），一个**Servlet**可以设置多个URL访问。**Servlet**不是线程安全，因此要谨慎使用类变量。

阐述Servlet和CGI的区别?

CGI的不足之处:

- 1, 需要为每个请求启动一个操作CGI程序的系统进程。如果请求频繁，这将会带来很大的开销。
- 2, 需要为每个请求加载和运行一个CGI程序，这将带来很大的开销
- 3, 需要重复编写处理网络协议的代码以及编码，这些工作都是非常耗时的。

Servlet的优点:

- 1, 只需要启动一个操作系统进程以及加载一个JVM，大大降低了系统的开销
- 2, 如果多个请求需要做同样处理的时候，这时候只需要加载一个类，这也大大降低了开销
- 3, 所有动态加载的类可以实现对网络协议以及请求解码的共享，大大降低了工作量。
- 4, Servlet能直接和Web服务器交互，而普通的CGI程序不能。Servlet还能在各个程序之间共享数据，使数据库连接池之类的功能很容易实现。

补充：Sun Microsystems公司在1996年发布Servlet技术就是为了和CGI进行竞争，Servlet是一个特殊的Java程序，一个基于Java的Web应用通常包含一个或多个Servlet类。Servlet不能够自行创建并执行，它是在Servlet容器中运行的，容器将用户的请求传递给Servlet程序，并将Servlet的响应回传给用户。通常一个Servlet会关联一个或多个JSP页面。以前CGI经常因为性能开销上的问题被诟病，然而Fast CGI早就已经解决了CGI效率上的问题，所以面试的时候大可不必信口开河的诟病CGI，事实上有很多你熟悉的网站都使用了CGI技术。

参考：《javaweb整合开发王者归来》P7

Servlet接口中有哪些方法及Servlet生命周期探秘

Servlet接口定义了5个方法，其中前三个方法与Servlet生命周期相关：

- **void init(ServletConfig config) throws ServletException**
- **void service(ServletRequest req, ServletResponse resp) throws ServletException, java.io.IOException**
- **void destroy()**
- **java.lang.String getServletInfo()**
- **ServletConfig getServletConfig()**

生命周期： Web容器加载Servlet并将其实例化后，Servlet生命周期开始，容器运行其init()方法进行Servlet的初始化；请求到达时调用Servlet的service()方法，service()方法会根据需要调用与请求对应的doGet或doPost等方法；当服务器关闭或项目被卸载时服务器会将Servlet实例销毁，此时会调用Servlet的destroy()方法。**init方法和destroy方法只会执行一次，service方法客户端每次请求Servlet都会执行。**Servlet中有时会用到一些需要初始化与销毁的资源，因此可以把初始化资源的代码放入init方法中，销毁资源的代码放入destroy方法中，这样就不需要每次处理客户端的请求都要初始化与销毁资源。

参考：《javaweb整合开发王者归来》P81

get和post请求的区别

①get请求用来从服务器上获得资源，而post是用来向服务器提交数据；

②get将表单中数据按照name=value的形式，添加到action所指向的URL后面，并且两者使用"?"连接，而各个变量之间使用"&"连接；post是将表单中的数据放在HTTP协议的请求头或消息体中，传递到action所指向URL；

③get传输的数据要受到URL长度限制（1024字节即256个字符）；而post可以传输大量的数据，上传文件通常要使用post方式；

④使用get时参数会显示在地址栏上，如果这些数据不是敏感数据，那么可以使用get；对于敏感数据还是应用使用post；

⑤get使用MIME类型application/x-www-form-urlencoded的URL编码（也叫百分号编码）文本的格式传递参数，保证被传送的参数由遵循规范的文本组成，例如一个空格的编码是"%20"。

补充：GET方式提交表单的典型应用是搜索引擎。GET方式就是被设计为查询用的。

还有另外一种回答。推荐大家看一下：

- <https://www.zhihu.com/question/28586791>
- https://mp.weixin.qq.com/s?__biz=MzI3NzIzMzg3Mw==&mid=100000054&idx=1&sn=71f6c214f3833d9ca20b9f7dc9d33e4#rd

什么情况下调用doGet()和doPost()

Form标签里的method的属性为get时调用doGet()，为post时调用doPost()。

转发(Forward)和重定向(Redirect)的区别

转发是服务器行为，重定向是客户端行为。

转发 (Forward) 通过RequestDispatcher对象的forward (HttpServletRequest request, HttpServletResponse response) 方法实现的。RequestDispatcher可以通过HttpServletRequest 的getRequestDispatcher()方法获得。例如下面的代码就是跳转到login_success.jsp页面。

```
request.getRequestDispatcher("login_success.jsp").forward(request, response);
```

重定向 (Redirect) 是利用服务器返回的状态码来实现的。客户端浏览器请求服务器的时候，服务器会返回一个状态码。服务器通过HttpServletRequestResponse的setStatus(int status)方法设置状态码。如果服务器返回301或者302，则浏览器会到新的网址重新请求该资源。

1. 从地址栏显示来说

forward是服务器请求资源,服务器直接访问目标地址的URL,把那个URL的响应内容读取过来,然后把这些内容再发给浏览器.浏览器根本不知道服务器发送的内容从哪里来的,所以它的地址栏还是原来的地址. redirect是服务端根据逻辑,发送一个状态码,告诉浏览器重新去请求那个地址.所以地址栏显示的是新的URL.

1. 从数据共享来说

forward:转发页面和转发到的页面可以共享request里面的数据. redirect:不能共享数据.

1. 从运用地方来说

forward:一般用于用户登陆的时候,根据角色转发到相应的模块. redirect:一般用于用户注销登陆时返回主页面和跳转到其它的网站等

1. 从效率来说

forward:高. redirect:低.

自动刷新(Refresh)

自动刷新不仅可以实现一段时间之后自动跳转到另一个页面，还可以实现一段时间之后自动刷新本页面。Servlet中通过HttpServletResponse对象设置Header属性实现自动刷新例如：

```
Response.setHeader ("Refresh", "1000;URL=http://localhost:8080/servlet/example.htm");
```

其中1000为时间，单位为毫秒。URL指定就是要跳转的页面（如果设置自己的路径，就会实现没过一秒自动刷新本页面一次）

Servlet与线程安全

Servlet不是线程安全的，多线程并发的读写会导致数据不同步的问题。解决的办法是尽量不要定义name属性，而是要把name变量分别定义在doGet()和doPost()方法内。虽然使用synchronized(name){}语句块可以解决问题，但是会造成线程的等待，不是很科学的办法。注意：多线程的并发的读写Servlet类属性会导致数据不同步。但是如果只是并发地读取属性而不写入，则不存在数据不同步的问题。因此Servlet里的只读属性最好定义为final类型的。

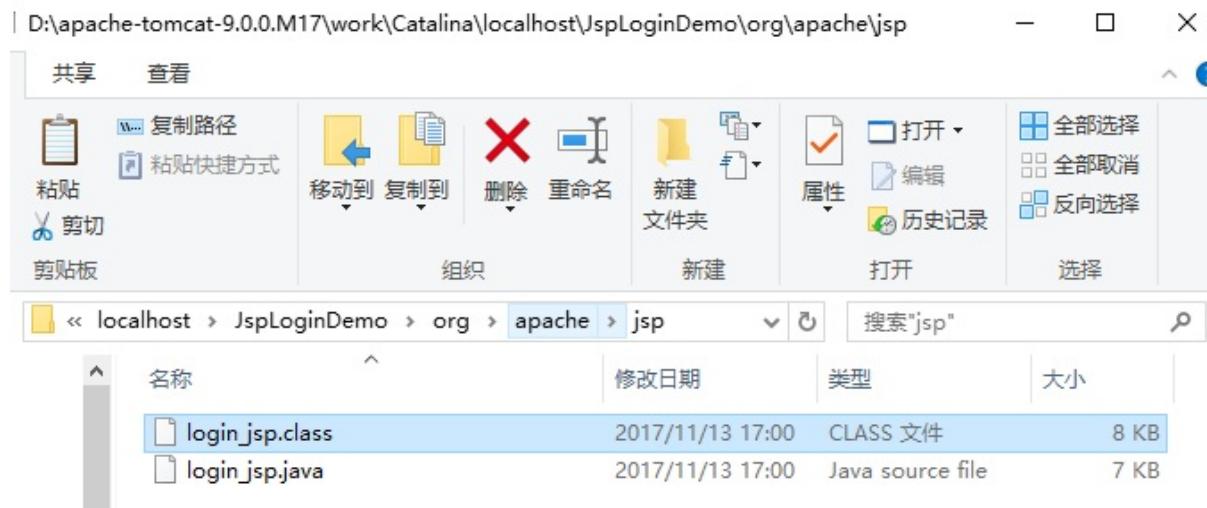
参考：《javaweb整合开发王者归来》P92

JSP和Servlet是什么关系

其实这个问题在上面已经阐述过了，Servlet是一个特殊的Java程序，它运行于服务器的JVM中，能够依靠服务器的支持向浏览器提供显示内容。JSP本质上是Servlet的一种简易形式，JSP会被服务器处理成一个类似于Servlet的Java程序，可以简化页面内容的生成。Servlet和JSP最主要的不同点在于，Servlet的应用逻辑是在Java文件中，并且完全从表示层中的HTML分离开来。而JSP的情况是Java和HTML可以组合成一个扩展名为.jsp的文件。有人说，Servlet就是在Java中写HTML，而JSP就是在HTML中写Java代码，当然这个说法是很片面且不够准确的。JSP侧重于视图，Servlet更侧重于控制逻辑，在MVC架构模式中，JSP适合充当视图（view）而Servlet适合充当控制器（controller）。

JSP工作原理

JSP是一种Servlet，但是与HttpServlet的工作方式不太一样。HttpServlet是先由源代码编译为class文件后部署到服务器下，为先编译后部署。而JSP则是先部署后编译。JSP会在客户端第一次请求JSP文件时被编译为HttpJspPage类（接口Servlet的一个子类）。该类会被服务器临时存放在服务器工作目录里面。下面通过实例给大家介绍。工程JspLoginDemo下有一个名为login.jsp的Jsp文件，把工程第一次部署到服务器上后访问这个Jsp文件，我们发现这个目录下多了下图这两个东东。.class文件便是JSP对应的Servlet。编译完毕后再运行class文件来响应客户端请求。以后客户端访问login.jsp的时候，Tomcat将不再重新编译JSP文件，而是直接调用class文件来响应客户端请求。



由于JSP只会在客户端第一次请求的时候被编译，因此第一次请求JSP时会感觉比较慢，之后就会感觉快很多。如果把服务器保存的class文件删除，服务器也会重新编译JSP。

开发Web程序时经常需要修改JSP。Tomcat能够自动检测到JSP程序的改动。如果检测到JSP源代码发生了改动。Tomcat会在下次客户端请求JSP时重新编译JSP，而不需要重启Tomcat。这种自动检测功能是默认开启的，检测改动会消耗少量的时间，在部署Web应用的时候可以在web.xml中将它关掉。

参考：《javaweb整合开发王者归来》P97

JSP有哪些内置对象、作用分别是什么

[JSP内置对象 - CSDN博客](#)

JSP有9个内置对象：

- request：封装客户端的请求，其中包含来自GET或POST请求的参数；
- response：封装服务器对客户端的响应；
- pageContext：通过该对象可以获取其他对象；
- session：封装用户会话的对象；
- application：封装服务器运行环境的对象；
- out：输出服务器响应的输出流对象；
- config：Web应用的配置对象；

- page: JSP 页面本身 (相当于 Java 程序中的 this) ;
- exception: 封装页面抛出异常的对象。

Request 对象的主要方法有哪些

- setAttribute(String name, Object): 设置名字为 name 的 request 的参数值
- getAttribute(String name): 返回由 name 指定的属性值
- getAttributeNames(): 返回 request 对象所有属性的名字集合, 结果是一个枚举的实例
- getCookies(): 返回客户端的所有 Cookie 对象, 结果是一个 Cookie 数组
- getCharacterEncoding(): 返回请求中的字符编码方式 = getContentType(): 返回请求的 Body 的长度
- getHeader(String name): 获得 HTTP 协议定义的文件头信息
- getHeaders(String name): 返回指定名字的 request Header 的所有值, 结果是一个枚举的实例
- getHeaderNames(): 返回所以 request Header 的名字, 结果是一个枚举的实例
- getInputStream(): 返回请求的输入流, 用于获得请求中的数据
- getMethod(): 获得客户端向服务器端传送数据的方法
- getParameter(String name): 获得客户端传送给服务器端的有 name 指定的参数值
- getParameterNames(): 获得客户端传送给服务器端的所有参数的名字, 结果是一个枚举的实例
- getParameterValues(String name): 获得有 name 指定的参数的所有值
- getProtocol(): 获取客户端向服务器端传送数据所依据的协议名称
- getQueryString(): 获得查询字符串
- getRequestURI(): 获取发出请求字符串的客户端地址
- getRemoteAddr(): 获取客户端的 IP 地址
- getRemoteHost(): 获取客户端的名字
- getSession([Boolean create]): 返回和请求相关 Session
- getServerName(): 获取服务器的名字
- getServletPath(): 获取客户端所请求的脚本文件的路径
- getServerPort(): 获取服务器的端口号
- removeAttribute(String name): 删除请求中的一个属性

request.getAttribute() 和 request.getParameter() 有何区别

从获取方向来看：

getParameter() 是获取 POST/GET 传递的参数值；

getAttribute() 是获取对象容器中的数据值；

从用途来看：

getParameter 用于客户端重定向时，即点击了链接或提交按钮时传值用，即用于在用表单或 url 重定向传值时接收数据用。

getAttribute 用于服务器端重定向时，即在 servlet 中使用了 forward 函数，或 struts 中使用了 mapping.findForward。getAttribute 只能收到程序用 setAttribute 传过来的值。

另外，可以用 setAttribute, getAttribute 发送接收对象，而 getParameter 显然只能传字符串。setAttribute 是应用服务器把这个对象放在该页面所对应的一块内存中去，当你的页面服务器重定向到另一个页面时，应用服务器会把这块内存拷贝另一个页面所对应的内存中。这样getAttribute 就能取得你所设下的值，当然这种方法可以传对象。session 也一样，只是对象在内存中的生命周期不一样而已。getParameter 只是应用服务器在分析你送上的 request 页面的文本时，取得你设在表单或 url 重定向时的值。

总结：

`getParameter` 返回的是String, 用于读取提交的表单中的值; (获取之后会根据实际需要转换为自己需要的相应类型, 比如整型, 日期类型啊等等)

`getAttribute` 返回的是Object, 需进行转换, 可用`setAttribute` 设置成任意对象, 使用很灵活, 可随时用

include指令和include动作的区别

include指令: JSP可以通过`include`指令来包含其他文件。被包含的文件可以是JSP文件、HTML文件或文本文件。包含的文件就好像是该JSP文件的一部分, 会被同时编译执行。语法格式如下: `<%@ include file="文件相对 url 地址" %>`

include动作: 动作元素用来包含静态和动态的文件。该动作把指定文件插入正在生成的页面。语法格式如下:

JSP九大内置对象, 七大动作, 三大指令

[JSP九大内置对象, 七大动作, 三大指令总结](#)

讲解JSP中的四种作用域

JSP中的四种作用域包括`page`、`request`、`session`和`application`, 具体来说:

- **page**代表与一个页面相关的对象和属性。
- **request**代表与Web客户机发出的一个请求相关的对象和属性。一个请求可能跨越多个页面, 涉及多个Web组件; 需要在页面显示的临时数据可以置于此作用域。
- **session**代表与某个用户与服务器建立的一次会话相关的对象和属性。跟某个用户相关的数据应该放在用户自己的`session`中。
- **application**代表与整个Web应用程序相关的对象和属性, 它实质上是跨越整个Web应用程序, 包括多个页面、请求和会话的一个全局作用域。

如何实现JSP或Servlet的单线程模式

对于JSP页面, 可以通过`page`指令进行设置。 `<%@page isThreadSafe="false"%>`

对于Servlet, 可以让自定义的Servlet实现`SingleThreadModel`标识接口。

说明: 如果将JSP或Servlet设置成单线程工作模式, 会导致每个请求创建一个Servlet实例, 这种实践将导致严重的性能问题(服务器的内存压力很大, 还会导致频繁的垃圾回收), 所以通常情况下并不会这么做。

实现会话跟踪的技术有哪些

1. 使用Cookie

向客户端发送Cookie

```
Cookie c =new Cookie("name","value"); //创建Cookie
c.setMaxAge(60*60*24); //设置最大时效, 此处设置的最大时效为一天
response.addCookie(c); //把Cookie放入到HTTP响应中
```

从客户端读取Cookie

```
String name ="name";
```

```

Cookie[]cookies =request.getCookies();
if(cookies !=null){
    for(int i= 0;i<cookies.length;i++){
        Cookie cookie =cookies[i];
        if(name.equals(cookie.getName()))
            //something is here.
            //you can get the value
            cookie.getValue();

    }
}

```

优点: 数据可以持久保存, 不需要服务器资源, 简单, 基于文本的Key-Value

缺点: 大小受到限制, 用户可以禁用Cookie功能, 由于保存在本地, 有一定的安全风险。

1. URL 重写

在URL中添加用户会话的信息作为请求的参数, 或者将唯一的会话ID添加到URL结尾以标识一个会话。

优点: 在Cookie被禁用的时候依然可以使用

缺点: 必须对网站的URL进行编码, 所有页面必须动态生成, 不能用预先记录下来的URL进行访问。

3. 隐藏的表单域

```
<input type="hidden" name ="session" value="..."/>
```

优点: Cookie被禁时可以使用

缺点: 所有页面必须是表单提交之后的结果。

1. HttpSession

在所有会话跟踪技术中, HttpSession对象是最强大也是功能最多的。当一个用户第一次访问某个网站时会自动创建 HttpSession, 每个用户可以访问他自己的HttpSession。可以通过HttpServletRequest对象的getSession方法获得 HttpSession, 通过HttpSession的setAttribute方法可以将一个值放在HttpSession中, 通过调用 HttpSession对象的getAttribute方法, 同时传入属性名就可以获取保存在HttpSession中的对象。与上面三种方式不同的是, HttpSession放在服务器的内存中, 因此不要将过大的对象放在里面, 即使目前的Servlet容器可以在内存将满时将HttpSession 中的对象移到其他存储设备中, 但是这样势必影响性能。添加到HttpSession中的值可以是任意Java对象, 这个对象最好实现了Serializable接口, 这样Servlet容器在必要的时候可以将其序列化到文件中, 否则在序列化时就会出现异常。

Cookie和Session的区别

1. 由于HTTP协议是无状态的协议, 所以服务端需要记录用户的状态时, 就需要用某种机制来识别具体的用户, 这个机制就是Session.典型的场景比如购物车, 当你点击下单按钮时, 由于HTTP协议无状态, 所以并不知道是哪个用户操作的, 所以服务端要为特定的用户创建了特定的Session, 用用于标识这个用户, 并且跟踪用户, 这样才知道购物车里面有几本书。这个Session是保存在服务端的, 有一个唯一标识。在服务端保存Session的方法很多, 内存、数据库、文件都有。集群的时候也要考虑Session的转移, 在大型的网站, 一般会有专门的Session服务器集群, 用来保存用户会话, 这个时候 Session 信息都是放在内存的, 使用一些缓存服务比如Memcached之类的来放 Session。
2. 思考一下服务端如何识别特定的客户? 这个时候Cookie就登场了。每次HTTP请求的时候, 客户端都会发送相应的Cookie信息到服务端。实际上大多数的应用都是用Cookie 来实现Session跟踪的, 第一次创建Session的时候, 服务端会在HTTP协议中告诉客户端, 需要在Cookie 里面记录一个Session ID, 以后每次请求把这个会话ID发送到服务器, 我就知道你是谁了。有人问, 如果客户端的浏览器禁用了Cookie 怎么办? 一般这种情况下, 会使用一种叫做URL重写的技术来进行会话跟踪, 即每次HTTP交互, URL后面都会被附加上一个诸如 sid=xxxxx 这样的参数, 服务端据此来识别用户。

3. Cookie其实还可以用在一些方便用户的场景下，设想你某次登陆过一个网站，下次登录的时候不想再次输入账号了，怎么办？这个信息可以写到Cookie里面，访问网站的时候，网站页面的脚本可以读取这个信息，就自动帮你把用户名给填了，能够方便一下用户。这也是Cookie名称的由来，给用户的一点甜头。所以，总结一下：Session是在服务端保存的一个数据结构，用来跟踪用户的状态，这个数据可以保存在集群、数据库、文件中；Cookie是客户端保存用户信息的一种机制，用来记录用户的一些信息，也是实现Session的一种方式。

参考：

<https://www.zhihu.com/question/19786827/answer/28752144>

《javaweb整合开发王者归来》P158 Cookie和Session的比较

final 关键字

final关键字主要用在三个地方：变量、方法、类。

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

static 关键字

static 关键字主要有以下四种使用场景：

- 修饰成员变量和成员方法**: 被 **static** 修饰的成员属于类，不属于单个这个类的某个对象，被类中所有对象共享，可以并且建议通过类名调用。被**static** 声明的成员变量属于静态成员变量，静态变量存放在 Java 内存区域的方法区。调用格式：`类名.静态变量名` `类名.静态方法名()`
- 静态代码块**: 静态代码块定义在类中方法外，静态代码块在非静态代码块之前执行(静态代码块—>非静态代码块—>构造方法)。该类不管创建多少对象，静态代码块只执行一次。
- 静态内部类 (static修饰类的话只能修饰内部类)** : 静态内部类与非静态内部类之间存在一个最大的区别：非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外包围，但是静态内部类却没有。没有这个引用就意味着：1. 它的创建是不需要依赖外围类的创建。2. 它不能使用任何外围类的非**static**成员变量和方法。
- 静态导包(用来导入类中的静态资源，1.5之后的新特性)**: 格式为：`import static` 这两个关键字连用可以指定导入某个类中的指定静态资源，并且不需要使用类名调用类中静态成员，可以直接使用类中静态成员变量和成员方法。

this 关键字

this关键字用于引用类的当前实例。例如：

```
class Manager {
    Employees[] employees;

    void manageEmployees() {
        int totalEmp = this.employees.length;
        System.out.println("Total employees: " + totalEmp);
        this.report();
    }

    void report() { }
}
```

在上面的示例中，**this**关键字用于两个地方：

- `this.employees.length`: 访问类**Manager**的当前实例的变量。
- `this.report()` : 调用类**Manager**的当前实例的方法。

此关键字是可选的，这意味着如果上面的示例在不使用此关键字的情况下表现相同。但是，使用此关键字可能会使代码更易读或易懂。

super 关键字

super关键字用于从子类访问父类的变量和方法。例如：

```
public class Super {
    protected int number;

    protected showNumber() {
        System.out.println("number = " + number);
    }
}

public class Sub extends Super {
    void bar() {
        super.number = 10;
        super.showNumber();
    }
}
```

在上面的例子中，Sub 类访问父类成员变量 number 并调用其父类 Super 的 showNumber () 方法。

使用 this 和 super 要注意的问题：

- super 调用父类中的其他构造方法时，调用时要放在构造方法的首行！this 调用本类中的其他构造方法时，也要放在首行。
- this、super不能用在static方法中。

简单解释一下：

被 static 修饰的成员属于类，不属于单个这个类的某个对象，被类中所有对象共享。而 this 代表对本类对象的引用，指向本类对象；而 super 代表对父类对象的引用，指向父类对象；所以， **this和super是属于对象范畴的东西，而静态方法是属于类范畴的东西。**

参考

- <https://www.codejava.net/java-core/the-javas-language/java-keywords>
- <https://blog.csdn.net/u013393958/article/details/79881037>

static 关键字

static 关键字主要有以下四种使用场景

1. 修饰成员变量和成员方法
2. 静态代码块
3. 修饰类(只能修饰内部类)
4. 静态导包(用来导入类中的静态资源, 1.5之后的新特性)

修饰成员变量和成员方法(常用)

被 static 修饰的成员属于类, 不属于单个这个类的某个对象, 被类中所有对象共享, 可以并且建议通过类名调用。被 static 声明的成员变量属于静态成员变量, 静态变量存放在 Java 内存区域的方法区。

方法区与 Java 堆一样, 是各个线程共享的内存区域, 它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分, 但是它却有一个别名叫做 Non-Heap (非堆), 目的应该是与 Java 堆区分开来。

HotSpot 虚拟机中方法区也常被称为“永久代”, 本质上两者并不等价。仅仅是因为 HotSpot 虚拟机设计团队用永久代来实现方法区而已, 这样 HotSpot 虚拟机的垃圾收集器就可以像管理 Java 堆一样管理这部分内存了。但是这并不是一个好主意, 因为这样更容易遇到内存溢出问题。

调用格式:

- 类名.静态变量名
- 类名.静态方法名()

如果变量或者方法被 private 则代表该属性或者该方法只能在类的内部被访问而不能在类的外部被方法。

测试方法:

```
public class StaticBean {

    String name;
    静态变量
    static int age;

    public StaticBean(String name) {
        this.name = name;
    }
    静态方法
    static void SayHello() {
        System.out.println(Hello i am java);
    }
    @Override
    public String toString() {
        return StaticBean{ +
            name=' + name + '\'' + age + age +
            '}';
    }
}
```

```
public class StaticDemo {

    public static void main(String[] args) {
        StaticBean staticBean = new StaticBean(1);
        StaticBean staticBean2 = new StaticBean(2);
```

```

        StaticBean staticBean3 = new StaticBean(3);
        StaticBean staticBean4 = new StaticBean(4);
        StaticBean.age = 33;
        StaticBean{name='1'age33} StaticBean{name='2'age33} StaticBean{name='3'age33} StaticBean{name='4'age33}
        System.out.println(staticBean+ +staticBean2+ +staticBean3+ +staticBean4);
        StaticBean.SayHello();Hello i am java
    }

}

```

静态代码块

静态代码块定义在类中方法外，静态代码块在非静态代码块之前执行(静态代码块—非静态代码块—构造方法)。该类不管创建多少对象，静态代码块只执行一次。

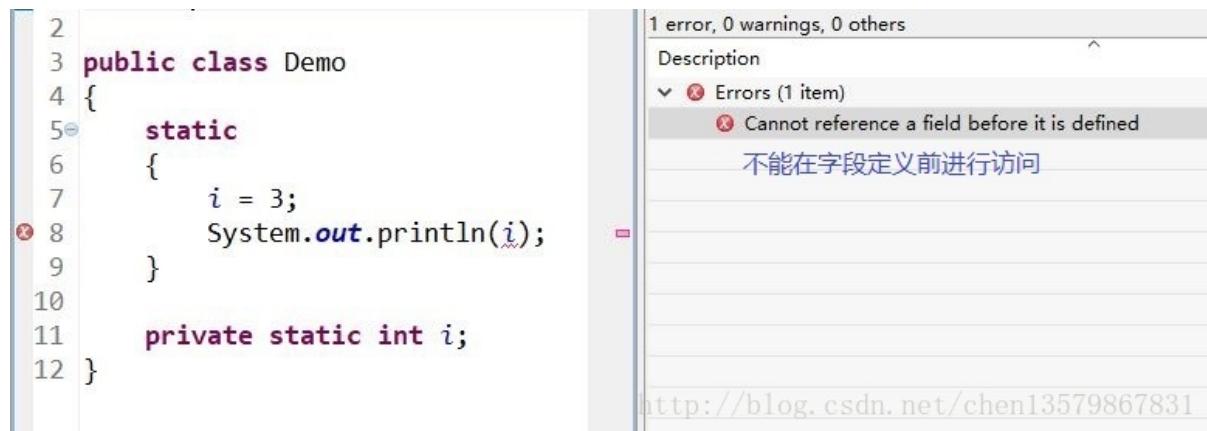
静态代码块的格式是

```

static {
    语句体;
}

```

一个类中的静态代码块可以有多个，位置可以随便放，它不在任何的方法体内，JVM加载类时会执行这些静态的代码块，如果静态代码块有多个，JVM将按照它们在类中出现的先后顺序依次执行它们，每个代码块只会被执行一次。



静态代码块对于定义在它之后的静态变量，可以赋值，但是不能访问。

静态内部类

静态内部类与非静态内部类之间存在一个最大的区别，我们知道非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外包围，但是静态内部类却没有。没有这个引用就意味着：

1. 它的创建是不需要依赖外围类的创建。
2. 它不能使用任何外围类的非static成员变量和方法。

Example (静态内部类实现单例模式)

```

public class Singleton {

    声明为 private 避免调用默认构造方法创建对象
    private Singleton() {
    }

    声明为 private 表明静态内部类只能在该 Singleton 类中被访问
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
}

```

```

    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}

```

当 Singleton 类加载时，静态内部类 SingletonHolder 没有被加载进内存。只有当调用 `getInstance()` 方法从而触发 `SingletonHolder.INSTANCE` 时 `SingletonHolder` 才会被加载，此时初始化 `INSTANCE` 实例，并且 JVM 能确保 `INSTANCE` 只被实例化一次。

这种方式不仅具有延迟初始化的好处，而且由 JVM 提供了对线程安全的支持。

静态导包

格式为：`import static`

这两个关键字连用可以指定导入某个类中的指定静态资源，并且不需要使用类名调用类中静态成员，可以直接使用类中静态成员变量和成员方法

`Math.` --- 将 `Math` 中的所有静态资源导入，这时候可以直接使用里面的静态方法，而不用通过类名进行调用
如果只想导入单一某个静态方法，只需要将换成对应的方法名即可

```

import static java.lang.Math.*;

换成import static java.lang.Math.max;具有一样的效果

public class Demo {
    public static void main(String[] args) {

        int max = max(1,2);
        System.out.println(max);
    }
}

```

补充内容

静态方法与非静态方法

静态方法属于类本身，非静态方法属于从该类生成的每个对象。如果您的方法执行的操作不依赖于其类的各个变量和方法，请将其设置为静态（这将使程序的占用空间更小）。否则，它应该是非静态的。

Example

```

class Foo {
    int i;
    public Foo(int i) {
        this.i = i;
    }

    public static String method1() {
        return An example string that doesn't depend on i (an instance variable);
    }

    public int method2() {
        return this.i + 1; Depends on i
    }
}

```

```
}
```

你可以像这样调用静态方法：`Foo.method1()`。如果您尝试使用这种方法调用 `method2` 将失败。但这样可行：`Foo bar = new Foo(1); bar.method2();`

总结：

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

static{}静态代码块与{}非静态代码块（构造代码块）

相同点：都是在JVM加载类时且在构造方法执行之前执行，在类中都可以定义多个，定义多个时按定义的顺序执行，一般在代码块中对一些static变量进行赋值。

不同点：静态代码块在非静态代码块之前执行（静态代码块—非静态代码块—构造方法）。静态代码块只在第一次new执行一次，之后不再执行，而非静态代码块在每new一次就执行一次。非静态代码块可在普通方法中定义（不过作用不大）；而静态代码块不行。

一般情况下，如果有些代码比如一些项目最常用的变量或对象必须在项目启动的时候就执行的时候，需要使用静态代码块，这种代码是主动执行的。如果我们想要设计不需要创建对象就可以调用类中的方法，例如：Arrays类，Character类，String类等，就需要使用静态方法，两者的区别是静态代码块是自动执行的而静态方法是被调用的时候才执行的。

Example

```
public class Test {
    public Test() {
        System.out.print("默认构造方法! --");
    }

    非静态代码块
    {
        System.out.print("非静态代码块! --");
    }

    静态代码块
    static {
        System.out.print("静态代码块! --");
    }

    public static void test() {
        System.out.print("静态方法中的内容! --");
        {
            System.out.print("静态方法中的代码块! --");
        }
    }

    public static void main(String[] args) {
        Test test = new Test();
        Test.test(); 静态代码块! -- 静态方法中的内容! -- 静态方法中的代码块! --
    }
}
```

当执行 `Test.test();` 时输出：

```
静态代码块! -- 静态方法中的内容! -- 静态方法中的代码块! --
```

当执行 `Test test = new Test();` 时输出：

静态代码块! --非静态代码块! --默认构造方法! --

非静态代码块与构造函数的区别是： 非静态代码块是给所有对象进行统一初始化，而构造函数是给对应的对象初始化，因为构造函数是可以多个的，运行哪个构造函数就会建立什么样的对象，但无论建立哪个对象，都会先执行相同的构造代码块。也就是说，构造代码块中定义的是不同对象共性的初始化内容。

参考

- <https://blog.csdn.net/chen13579867831/article/details/78995480>
- <http://www.cnblogs.com/chenssyp3388487.html>
- <http://www.cnblogs.com/Qian123/p/5713440.html>

本文是“最常见Java面试题总结”系列第三周的文章。主要内容：

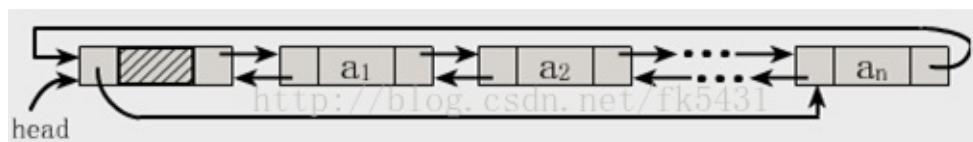
1. ArrayList 与 LinkedList 异同
2. ArrayList 与 Vector 区别
3. HashMap的底层实现
4. HashMap 和 Hashtable 的区别
5. HashMap 的长度为什么是2的幂次方
6. HashMap 多线程操作导致死循环问题
7. HashSet 和 HashMap 区别
8. ConcurrentHashMap 和 Hashtable 的区别
9. ConcurrentHashMap线程安全的具体实现方式/底层具体实现
10. 集合框架底层数据结构总结

ArrayList 与 LinkedList 异同

- 1. 是否保证线程安全： ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；
- 2. 底层数据结构： ArrayList 底层使用的是Object数组； LinkedList 底层使用的是双向循环链表数据结构；
- 3. 插入和删除是否受元素位置的影响： ① ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。比如：执行 `add(E e)` 方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是 $O(1)$ 。但是如果要在指定位置 i 插入和删除元素的话（`add(int index, E element)`）时间复杂度就为 $O(n-i)$ 。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的 $(n-i)$ 个元素都要执行向后位/向前移一位的操作。② LinkedList 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是近似 $O(1)$ 而数组为近似 $O(n)$ 。
- 4. 是否支持快速随机访问： LinkedList 不支持高效的随机元素访问，而 ArrayList 实现了 RandomAccess 接口，所以有随机访问功能。快速随机访问就是通过元素的序号快速获取元素对象(对应于 `get(int index)` 方法)。
- 5. 内存空间占用： ArrayList 的空间浪费主要体现在在 list 列表的结尾会预留一定的容量空间，而 LinkedList 的空间花费则体现在它的每一个元素都需要消耗比 ArrayList 更多的空间（因为要存放直接后继和直接前驱以及数据）。

补充：数据结构基础之双向链表

双向链表也叫双链表，是链表的一种，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。一般我们都构造双向循环链表，如下图所示，同时下图也是 LinkedList 底层使用的是双向循环链表数据结构。



ArrayList 与 Vector 区别

Vector类的所有方法都是同步的。可以由两个线程安全地访问一个Vector对象、但是一个线程访问Vector的话代码要在同步操作上耗费大量的时间。

ArrayList不是同步的，所以在不需要保证线程安全时建议使用ArrayList。

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);
}
```

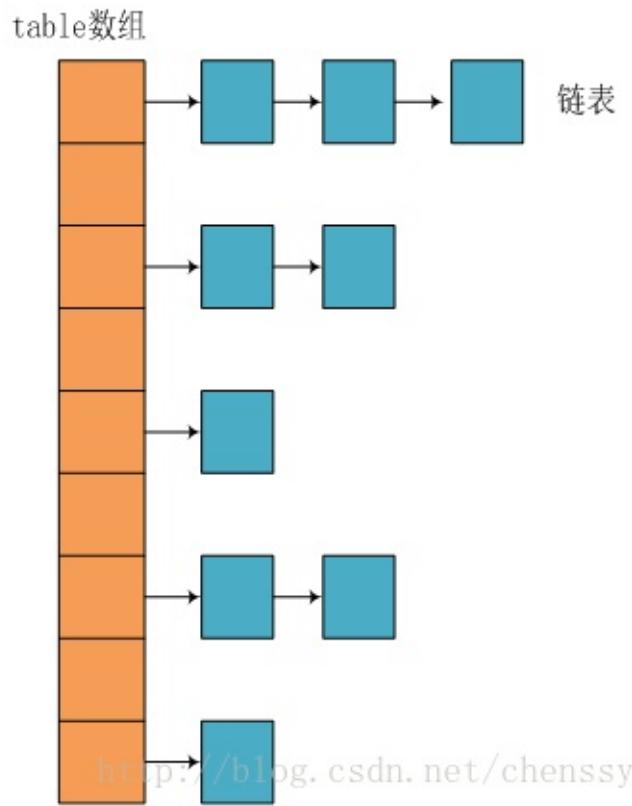
对比一下 JDK1.7 的 HashMap 的 hash 方法源码.

```
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);
}
```

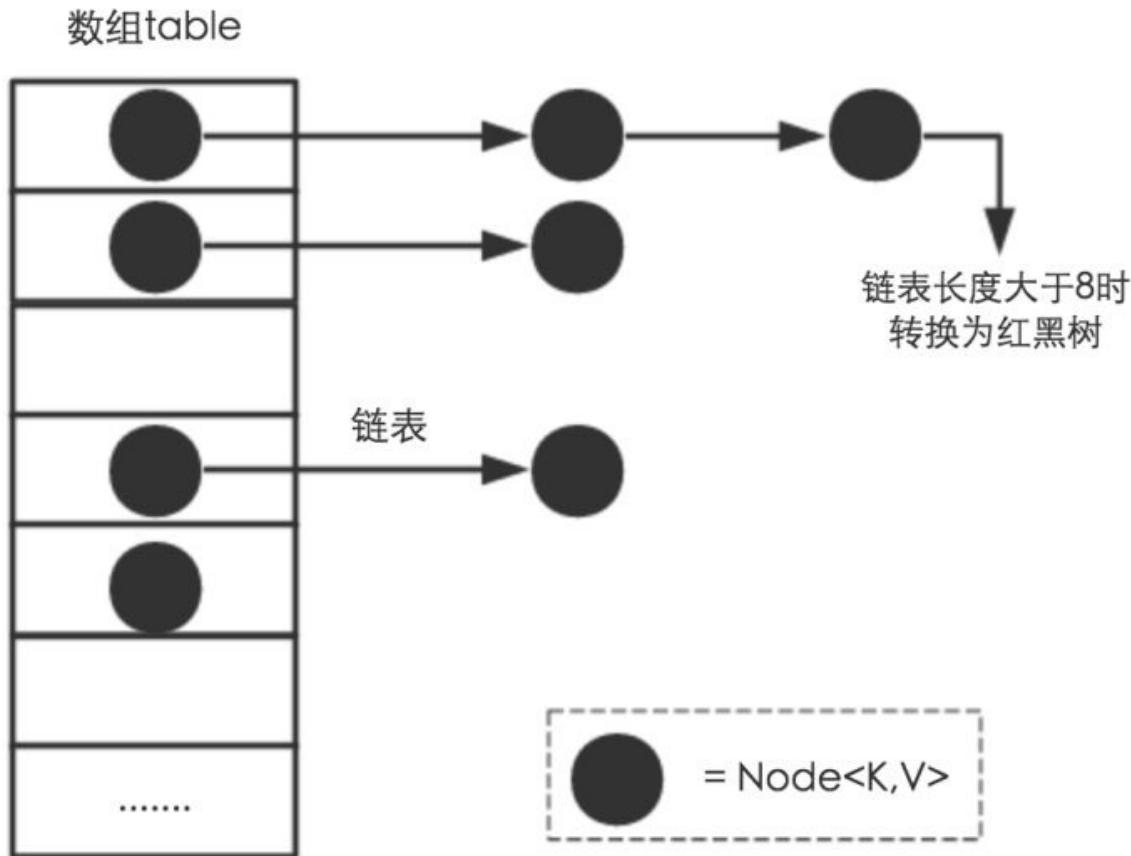
相比于 JDK1.8 的 hash 方法，JDK 1.7 的 hash 方法的性能会稍差一点点，因为毕竟扰动了 4 次。

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）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。

推荐阅读：

- 《Java 8系列之重新认识HashMap》：<https://zhuanlan.zhihu.com/p/21673805>

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；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 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛出 `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。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，以减少搜索时间。Hashtable 没有这样的机制。

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2147483648，前后加起来大概40亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个40亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法是“ $(n - 1) \& hash$ ”。（ n 代表数组长度）。这也解释了 HashMap 的长度为什么是2的幂次方。

这个算法应该如何设计呢？

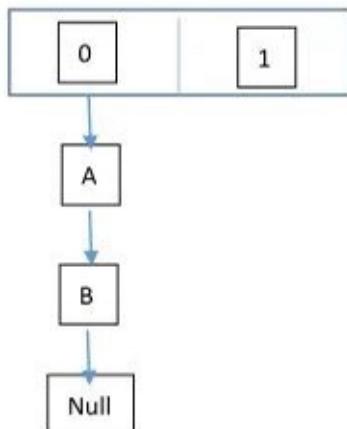
我们首先可能会想到采用%取余的操作来实现。但是，重点来了：“取余(%)操作中如果除数是2的幂次则等价于与其除数减一的与(&)操作（也就是说 $hash \% length == hash \& (length - 1)$ 的前提是 $length$ 是2的 n 次方；）。”并且采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是2的幂次方。

HashMap 多线程操作导致死循环问题

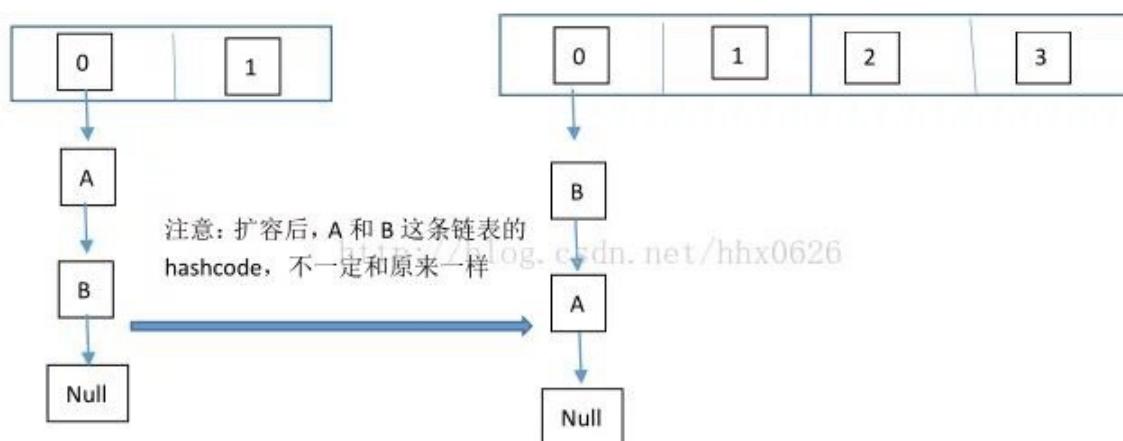
在多线程下，进行 put 操作会导致 HashMap 死循环，原因在于 HashMap 的扩容 `resize()` 方法。由于扩容是新建一个数组，复制原数据到数组。由于数组下标挂有链表，所以需要复制链表，但是多线程操作有可能导致环形链表。复制链表过程如下：

以下模拟2个线程同时扩容。假设，当前 HashMap 的空间为2（临界值为1），`hashcode` 分别为 0 和 1，在散列地址 0 处有元素 A 和 B，这时候要添加元素 C，C 经过 `hash` 运算，得到散列地址为 1，这时候由于超过了临界值，空间不够，需要调用 `resize` 方法进行扩容，那么在多线程条件下，会出现条件竞争，模拟过程如下：

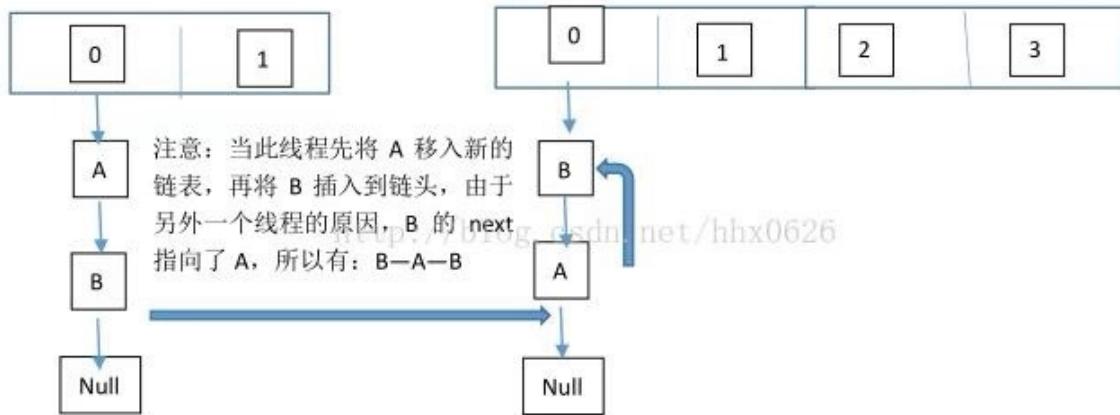
线程一：读取到当前的 HashMap 情况，在准备扩容时，线程二介入



线程二：读取 HashMap，进行扩容



线程一：继续执行



这个过程为，先将 A 复制到新的 hash 表中，然后接着复制 B 到链头（A 的前边：B.next=A），本来 B.next=null，到此也就结束了（跟线程二一样的过程），但是，由于线程二扩容的原因，将 B.next=A，所以，这里继续复制A，让 A.next=B，由此，环形链表出现：B.next=A; A.next=B

HashSet 和 HashMap 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() 方法、writeObject()方法、readObject()方法是 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， 如果两个对象不同的话，那么返回false
HashMap相对于HashSet较快，因为它是使用唯一的键获取对象	HashSet较HashMap来说比较慢

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。

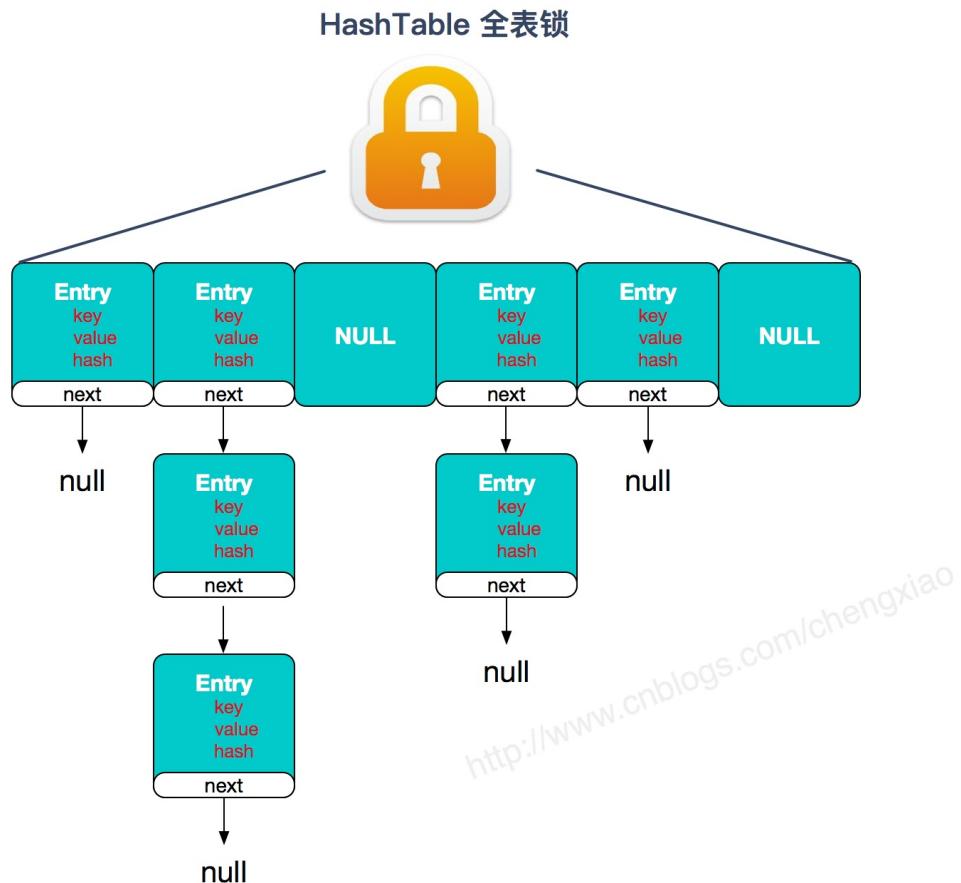
- 底层数据结构：JDK1.7的 ConcurrentHashMap 底层采用 分段的数组+链表 实现，JDK1.8 采用的数据结构跟 HashMap1.8的结构一样，数组+链表/红黑二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 数组+链表 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；
- 实现线程安全的方式（重要）：① 在JDK1.7的时候，ConcurrentHashMap（分段锁） 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。（默认分配16个Segment，比Hashtable效率提高16倍。）到了JDK1.8的时候已经摒弃了 Segment的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS

来操作。（JDK1.6以后对 `synchronized` 锁做了很多优化）整个看起来就像是优化过且线程安全的 `HashMap`，虽然在JDK1.8中还能看到 `Segment` 的数据结构，但是已经简化了属性，只是为了兼容旧版本；② **Hashtable(同一把锁)**：使用 `synchronized` 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 `put` 添加元素，另一个线程不能使用 `put` 添加元素，也不能使用 `get`，竞争会越来越激烈效率越低。

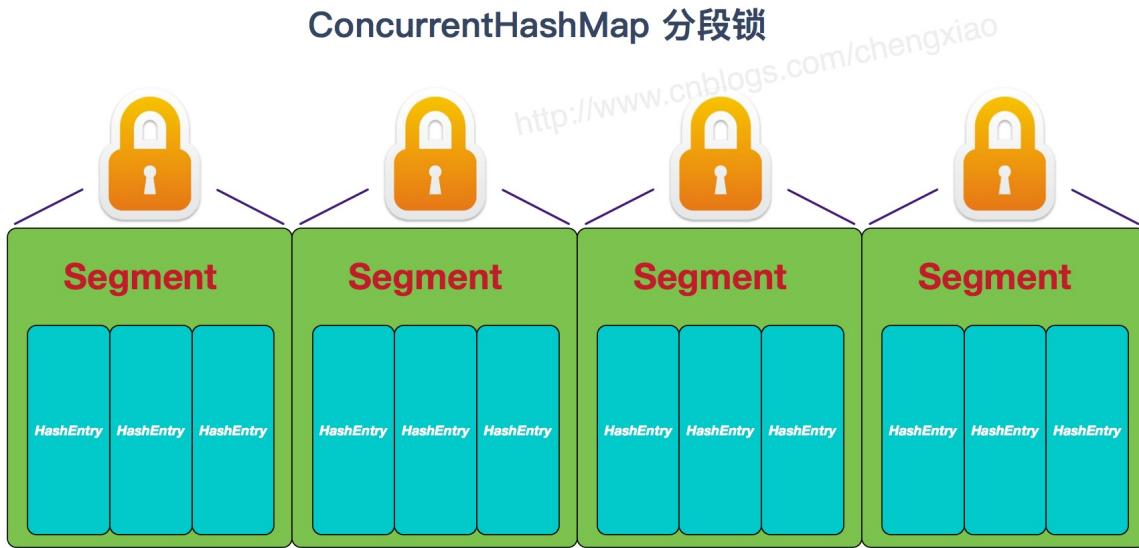
两者的对比图：

图片来源：<http://www.cnblogs.com/chengxiao/p/6842045.html>

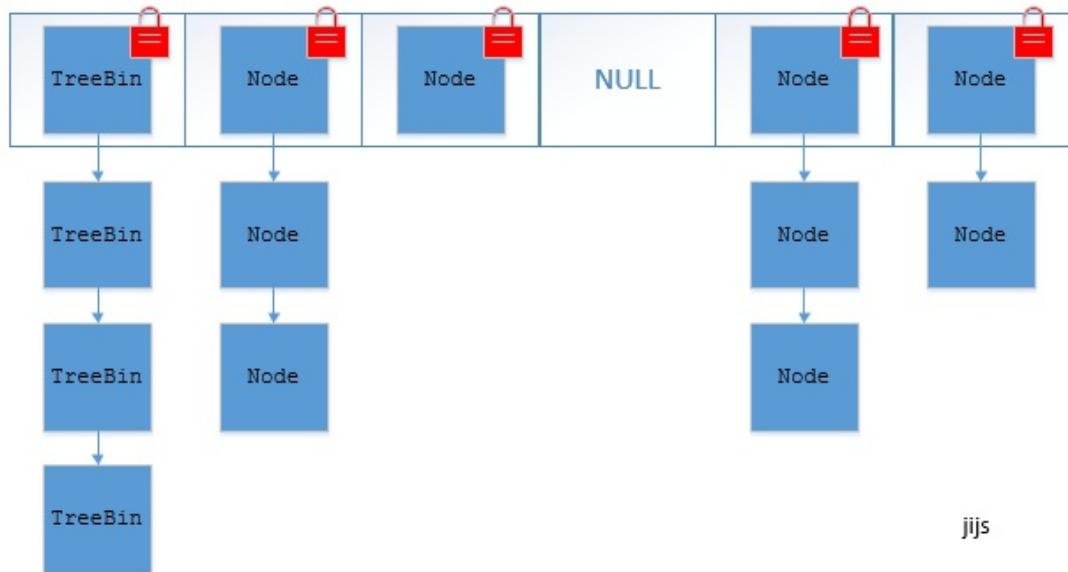
HashTable:



JDK1.7的ConcurrentHashMap：



JDK1.8的ConcurrentHashMap (TreeBin: 红黑二叉树节点 Node: 链表节点)：



ConcurrentHashMap线程安全的具体实现方式/底层具体实现

JDK1.7 (上面有示意图)

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

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

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

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

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

JDK1.8（上面有示意图）

ConcurrentHashMap取消了Segment分段锁，采用CAS和synchronized来保证并发安全。数据结构跟HashMap1.8的结构类似，数组+链表/红黑二叉树。

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

集合框架底层数据结构总结

Collection

1. List

- **ArrayList**: Object数组
- **Vector**: Object数组
- **LinkedList**: 双向循环链表

2. Set

- **HashSet** (无序, 唯一) : 基于 HashMap 实现的，底层采用 HashMap 来保存元素
- **LinkedHashSet**: LinkedHashSet 继承与 HashSet，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。)

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）时，将链表转化为红黑树，以减少搜索时间
- **LinkedHashMap**: LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外， LinkedHashMap 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。详细可以查看：[《LinkedHashMap 源码详细分析（JDK1.8）》](#)
- **HashTable**: 数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的
- **TreeMap**: 红黑树（自平衡的排序二叉树）

推荐阅读：

- [jdk1.8中ConcurrentHashMap的实现原理](#)
- [HashMap? ConcurrentHashMap? 相信看完这篇没人能难住你!](#)
- [HASHMAP、HASHTABLE、CONCURRENTHASHMAP的原理与区别](#)
- [ConcurrentHashMap实现原理及源码分析](#)
- [java-并发-ConcurrentHashMap高并发机制-jdk1.8](#)

1. List, Set, Map三者的区别及总结
2. ArrayList 与 LinkedList 区别
3. ArrayList 与 Vector 区别 (为什么要用ArrayList取代Vector呢?)
4. HashMap 和 Hashtable 的区别
5. HashSet 和 HashMap 区别
6. HashMap 和 ConcurrentHashMap 的区别
7. HashSet如何检查重复
8. comparable 和 comparator的区别
 - i. Comparator定制排序
 - ii. 重写compareTo方法实现按年龄来排序
9. 如何对Object的list排序?
10. 如何实现数组与List的相互转换?
11. 如何求ArrayList集合的交集 并集 差集 去重复并集
12. HashMap 的工作原理及代码实现
13. ConcurrentHashMap 的工作原理及代码实现
14. 集合框架底层数据结构总结
 - i. - Collection
 - i. 1. List
 - ii. 2. Set
 - ii. - Map
15. 集合的选用
16. 集合的常用方法

List, Set, Map三者的区别及总结

- **List:** 对付顺序的好帮手

List接口存储一组不唯一（可以有多个元素引用相同的对象），有序的对象

- **Set:**注重独一无二的性质

不允许重复的集合。不会有多个元素引用相同的对象。

- **Map:**用Key来搜索的专家

使用键值对存储。Map会维护与Key有关联的值。两个Key可以引用相同的对象，但Key不能重复，典型的Key是String类型，但也可以是任何对象。

ArrayList 与 LinkedList 区别

ArrayList底层使用的是数组（存读数据效率高，插入删除特定位置效率低），LinkedList底层使用的是双向循环链表数据结构（插入，删除效率特别高）。学过数据结构这门课后我们就知道采用链表存储，插入，删除元素时间复杂度不受元素位置的影响，都是近似O(1)而数组为近似O(n)，因此当数据特别多，而且经常需要插入删除元素时建议选用LinkedList.一般程序只用ArrayList就够了，因为一般数据量都不会蛮大，ArrayList是使用最多的集合类。

ArrayList 与 Vector 区别

Vector类的所有方法都是同步的。可以由两个线程安全地访问一个Vector对象、但是一个线程访问Vector，代码要在同步操作上耗费大量的时间。ArrayList不是同步的，所以在不需要同步时建议使用ArrayList。

HashMap 和 Hashtable 的区别

1. HashMap是非线程安全的，HashTable是线程安全的；HashTable内部的方法基本都经过synchronized修饰。
2. 因为线程安全的问题，HashMap要比HashTable效率高一点，HashTable基本被淘汰。
3. HashMap允许有null值的存在，而在HashTable中put进的键值只要有一个null，直接抛出NullPointerException。

Hashtable和HashMap有几个主要的不同：线程安全以及速度。仅在你需要完全的线程安全的时候使用Hashtable，而如果你使用Java5或以上的话，请使用ConcurrentHashMap吧

HashSet 和 HashMap 区别

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， 如果两个对象不同的话，那么返回false
HashMap相对于HashSet较快，因为它是使用唯一的键获取对象	HashSet较HashMap来说比较慢

HashMap 和 ConcurrentHashMap 的区别

HashMap与ConcurrentHashMap的区别

1. ConcurrentHashMap对整个桶数组进行了分割分段(Segment)，然后在每一个分段上都用lock锁进行保护，相对于HashTable的synchronized锁的粒度更精细了一些，并发性能更好，而HashMap没有锁机制，不是线程安全的。
(JDK1.8之后ConcurrentHashMap启用了一种全新的方式实现,利用CAS算法。)
2. HashMap的键值对允许有null，但是ConCurrentHashMap都不允许。

HashSet如何检查重复

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。（摘自我的Java启蒙书《Head fist java》第二版）

hashCode() 与 equals() 的相关规定：

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

==与equals的区别

1. ==是判断两个变量或实例是不是指向同一个内存空间 equals是判断两个变量或实例所指向的内存空间的值是不是相同
2. ==是指对内存地址进行比较 equals()是对字符串的内容进行比较3.==指引用是否相同 equals()指的是值是否相同

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

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

/**
 * TODO Collections类方法测试之排序
 * @author 寇爽
 * @date 2017年11月20日
 * @version 1.8
 */
public class CollectionsSort {

    public static void main(String[] args) {

        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 rotate(List list, int distance),旋转。
         * 当distance为正数时，将list后distance个元素整体移到前面。当distance为负数时，将
         * list的前distance个元素整体移到后面。
         */

        Collections.rotate(arrayList, 4);
        System.out.println("Collections.rotate(arrayList, 4):");
        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);

        // void shuffle(List list),随机排序
    }
}

```

```

Collections.shuffle(arrayList);
System.out.println("Collections.shuffle(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);
}
}

```

重写compareTo方法实现按年龄来排序

```

package map;

import java.util.Set;
import java.util.TreeMap;

public class TreeMap2 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        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());
        }
    }
}

// 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

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;
    }

    /**
     * TODO重写compareTo方法实现按年龄来排序
     */
    @Override
    public int compareTo(Person o) {
        // TODO Auto-generated method stub
        if (this.age > o.getAge()) {
            return 1;
        } else if (this.age < o.getAge()) {
            return -1;
        }
        return age;
    }
}

```

如何对Object的list排序

- 对objects数组进行排序，我们可以用Arrays.sort()方法
- 对objects的集合进行排序，需要使用Collections.sort()方法

如何实现数组与List的相互转换

List转数组： toArray(arraylist.size()方法； 数组转List:Arrays的asList(a)方法

```

List<String> arrayList = new ArrayList<String>();
arrayList.add("s");
arrayList.add("e");
arrayList.add("n");
/**
 * ArrayList转数组
 */
int size=arrayList.size();
String[] a = arrayList.toArray(new String[size]);
//输出第二个元素
System.out.println(a[1]);//结果: e
//输出整个数组
System.out.println(Arrays.toString(a));//结果: [s, e, n]
/**
 * 数组转list
 */
List<String> list=Arrays.asList(a);
/**
 * list转ArrayList
 */
List<String> arrayList2 = new ArrayList<String>();
arrayList2.addAll(list);
System.out.println(list);

```

如何求ArrayList集合的交集 并集 差集 去重复并集

需要用到List接口中定义的几个方法：

- addAll(Collection<? extends E> c) :按指定集合的Iterator返回的顺序将指定集合中的所有元素追加到此列表的末尾

实例代码：

- `retainAll(Collection<?> c)`: 仅保留此列表中包含在指定集合中的元素。
- `removeAll(Collection<?> c)` :从此列表中删除指定集合中包含的所有元素。 ``java package list;

```
import java.util.ArrayList; import java.util.List;

/* TODO 两个集合之间求交集 并集 差集 去重复并集

• @author 寇爽
• @date 2017年11月21日
• @version 1.8 */
public class MethodDemo {

    public static void main(String[] args) {

        // TODO Auto-generated method stub
        List<Integer> list1 = new ArrayList<Integer>();
        list1.add(1);
        list1.add(2);
        list1.add(3);
        list1.add(4);

        List<Integer> list2 = new ArrayList<Integer>();
        list2.add(2);
        list2.add(3);
        list2.add(4);
        list2.add(5);
        // 并集
        // list1.addAll(list2);
        // 交集
        // list1.retainAll(list2);
        // 差集
        // list1.removeAll(list2);
        // 无重复并集
        list2.removeAll(list1);
        list1.addAll(list2);
        for (Integer i : list1) {
            System.out.println(i);
        }
    }

}

...
}
```

HashMap 的工作原理及代码实现

[集合框架源码学习之HashMap\(JDK1.8\)](#)

ConcurrentHashMap 的工作原理及代码实现

[ConcurrentHashMap实现原理及源码分析](#)

集合框架底层数据结构总结

- Collection

1. List

- ArrayList: 数组 (查询快,增删慢 线程不安全,效率高)
- Vector: 数组 (查询快,增删慢 线程安全,效率低)
- LinkedList: 链表 (查询慢,增删快 线程不安全,效率高)

2. Set

- HashSet (无序, 唯一) :哈希表或者叫散列集(hash table)
- LinkedHashSet: 链表和哈希表组成。由链表保证元素的排序，由哈希表保证元素的唯一性
- TreeSet (有序, 唯一) : 红黑树(自平衡的排序二叉树。)

- Map

- HashMap: 基于哈希表的Map接口实现 (哈希表对键进行散列, Map结构即映射表存放键值对)
- LinkedHashMap:HashMap 的基础上加上了链表数据结构
- Hashtable:哈希表
- TreeMap:红黑树 (自平衡的排序二叉树)

集合的选用

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。

2018/3/11更新

集合的常用方法

今天下午无意看见一道某大厂的面试题，面试题的内容就是问你某一个集合常见的方法有哪些。虽然平时也经常见到这些集合，但是猛一下让我想某一个集合的常用的方法难免会有遗漏或者与其他集合搞混，所以建议大家还是照着API文档把常见的那几个集合的常用方法看一看。

会持续更新。。。

参考书籍：

《Head first java》第二版 推荐阅读真心不错 (适合基础较差的)

《Java核心技术卷1》推荐阅读真心不错 (适合基础较好的)

《算法》第四版 (适合想对数据结构的Java实现感兴趣的)

- [ArrayList简介](#)
- [ArrayList核心源码](#)
- [ArrayList源码分析](#)
 - [System.arraycopy\(\)和Arrays.copyOf\(\)方法](#)
 - [两者联系与区别](#)
 - [ArrayList核心扩容技术](#)
 - [内部类](#)
- [ArrayList经典Demo](#)

ArrayList简介

ArrayList 的底层是数组队列，相当于动态数组。与 Java 中的数组相比，它的容量能动态增长。在添加大量元素前，应用程序可以使用 `ensureCapacity` 操作来增加 ArrayList 实例的容量。这可以减少递增式再分配的数量。

它继承于 `AbstractList`，实现了 `List`, `RandomAccess`, `Cloneable`, `java.io.Serializable` 这些接口。

在我们学数据结构的时候就知道了线性表的顺序存储，插入删除元素的时间复杂度为 $O(n)$ ，求表长以及增加元素，取第 i 元素的时间复杂度为 $O(1)$

ArrayList 继承了 `AbstractList`，实现了 `List`。它是一个数组队列，提供了相关的添加、删除、修改、遍历等功能。

ArrayList 实现了 `RandomAccess` 接口，即提供了随机访问功能。`RandomAccess` 是 Java 中用来被 `List` 实现，为 `List` 提供快速访问功能的。在 ArrayList 中，我们即可以通过元素的序号快速获取元素对象，这就是快速随机访问。

ArrayList 实现了 `Cloneable` 接口，即覆盖了函数 `clone()`，能被克隆。

ArrayList 实现 `java.io.Serializable` 接口，这意味着 ArrayList 支持序列化，能通过序列化去传输。

和 Vector 不同，`ArrayList` 中的操作不是线程安全的！所以，建议在单线程中才使用 ArrayList，而在多线程中可以选择 Vector 或者 `CopyOnWriteArrayList`。

ArrayList核心源码

```
package java.util;

import java.util.function.Consumer;
import java.util.function.Predicate;
import java.util.function.UnaryOperator;

public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8683452581122892189L;

    /**
     * 默认初始容量大小
     */
    private static final int DEFAULT_CAPACITY = 10;

    /**
     * 空数组（用于空实例）。
     */
    private static final Object[] EMPTY_ELEMENTDATA = {};

    //用于默认大小空实例的共享空数组实例。
    //我们把它从EMPTY_ELEMENTDATA数组中区分出来，以知道在添加第一个元素时容量需要增加多少。
    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    /**
     * 保存ArrayList数据的数组
    
```

```

/*
transient Object[] elementData; // non-private to simplify nested class access

/**
 * ArrayList 所包含的元素个数
 */
private int size;

/**
 * 带初始容量参数的构造函数。 (用户自己指定容量)
 */
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        //创建initialCapacity大小的数组
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        //创建空数组
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    }
}

/**
 * 默认构造函数，其默认初始容量为10
 */
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}

/**
 * 构造一个包含指定集合的元素的列表，按照它们由集合的迭代器返回的顺序。
 */
public ArrayList(Collection<? extends E> c) {
    //
    elementData = c.toArray();
    //如果指定集合元素个数不为0
    if ((size = elementData.length) != 0) {
        // c.toArray 可能返回的不是Object类型的数组所以加上下面的语句用于判断,
        //这里用到了反射里面的getClass()方法
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    } else {
        // 用空数组代替
        this.elementData = EMPTY_ELEMENTDATA;
    }
}

/**
 * 修改这个ArrayList实例的容量是列表的当前大小。 应用程序可以使用此操作来最小化ArrayList实例的存储。
 */
public void trimToSize() {
    modCount++;
    if (size < elementData.length) {
        elementData = (size == 0)
            ? EMPTY_ELEMENTDATA
            : Arrays.copyOf(elementData, size);
    }
}

//下面是ArrayList的扩容机制
//ArrayList的扩容机制提高了性能，如果每次只扩充一个，
//那么频繁的插入会导致频繁的拷贝，降低性能，而ArrayList的扩容机制避免了这种情况。
/**
 * 如有必要，增加此ArrayList实例的容量，以确保它至少能容纳元素的数量
 * @param minCapacity 所需的最小容量
 */
public void ensureCapacity(int minCapacity) {
    int minExpand = (elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA)

```

```

        // any size if not default element table
        ? 0
        // larger than default for default empty table. It's already
        // supposed to be at default size.
        : DEFAULT_CAPACITY;

    if (minCapacity > minExpand) {
        ensureExplicitCapacity(minCapacity);
    }
}

//得到最小扩容量
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        // 获取默认的容量和传入参数的较大值
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }

    ensureExplicitCapacity(minCapacity);
}

//判断是否需要扩容
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        //调用grow方法进行扩容，调用此方法代表已经开始扩容了
        grow(minCapacity);
}

/**
 * 要分配的最大数组大小
 */
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

/**
 * ArrayList扩容的核心方法。
 */
private void grow(int minCapacity) {
    // oldCapacity为旧容量, newCapacity为新容量
    int oldCapacity = elementData.length;
    //将oldCapacity 右移一位, 其效果相当于oldCapacity / 2,
    //我们知道位运算的速度远远快于整除运算, 整句运算式的结果就是将新容量更新为旧容量的1.5倍,
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    //然后检查新容量是否大于最小需要容量, 若还是小于最小需要容量, 那么就把最小需要容量当作数组的新容量,
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    //再检查新容量是否超出了ArrayList所定义的最大容量,
    //若超出了, 则调用hugeCapacity()来比较minCapacity和 MAX_ARRAY_SIZE,
    //如果minCapacity大于最大容量, 则新容量则为ArrayList定义的最大容量, 否则, 新容量大小则为 minCapacity。
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}

//比较minCapacity和 MAX_ARRAY_SIZE
private static int hugeCapacity(int minCapacity) {
    if (minCapacity < 0) // overflow
        throw new OutOfMemoryError();
    return (minCapacity > MAX_ARRAY_SIZE) ?
        Integer.MAX_VALUE :
        MAX_ARRAY_SIZE;
}

/**
 * 返回此列表中的元素数。
 */
public int size() {
    return size;
}

```

```

    /**
     * 如果此列表不包含元素，则返回 true 。
     */
    public boolean isEmpty() {
        //注意=和==的区别
        return size == 0;
    }

    /**
     * 如果此列表包含指定的元素，则返回true 。
     */
    public boolean contains(Object o) {
        //indexOf()方法：返回此列表中指定元素的首次出现的索引，如果此列表不包含此元素，则为-1
        return indexOf(o) >= 0;
    }

    /**
     * 返回此列表中指定元素的首次出现的索引，如果此列表不包含此元素，则为-1
     */
    public int indexOf(Object o) {
        if (o == null) {
            for (int i = 0; i < size; i++)
                if (elementData[i]==null)
                    return i;
        } else {
            for (int i = 0; i < size; i++)
                //equals()方法比较
                if (o.equals(elementData[i]))
                    return i;
        }
        return -1;
    }

    /**
     * 返回此列表中指定元素的最后一次出现的索引，如果此列表不包含元素，则返回-1。.
     */
    public int lastIndexOf(Object o) {
        if (o == null) {
            for (int i = size-1; i >= 0; i--)
                if (elementData[i]==null)
                    return i;
        } else {
            for (int i = size-1; i >= 0; i--)
                if (o.equals(elementData[i]))
                    return i;
        }
        return -1;
    }

    /**
     * 返回此ArrayList实例的浅拷贝。（元素本身不被复制。）
     */
    public Object clone() {
        try {
            ArrayList<?> v = (ArrayList<?>) super.clone();
            //Arrays.copyOf功能是实现数组的复制，返回复制后的数组。参数是被复制的数组和复制的长度
            v.elementData = Arrays.copyOf(elementData, size);
            v.modCount = 0;
            return v;
        } catch (CloneNotSupportedException e) {
            // 这不应该发生，因为我们是可以克隆的
            throw new InternalError(e);
        }
    }

    /**
     * 以正确的顺序（从第一个到最后一个元素）返回一个包含此列表中所有元素的数组。
     * 返回的数组将是“安全的”，因为该列表不保留对它的引用。（换句话说，这个方法必须分配一个新的数组）。
     */

```

```

    *因此，调用者可以自由地修改返回的数组。此方法充当基于阵列和基于集合的API之间的桥梁。
    */
    public Object[] toArray() {
        return Arrays.copyOf(elementData, size);
    }

    /**
     * 以正确的顺序返回一个包含此列表中所有元素的数组（从第一个到最后一个元素）；
     * 返回的数组的运行时类型是指定数组的运行时类型。如果列表适合指定的数组，则返回其中。
     * 否则，将为指定数组的运行时类型和此列表的大小分配一个新数组。
     * 如果列表适用于指定的数组，其余空间（即数组的列表数量多于此元素），则紧跟在集合结束后的数组中的元素设置为null。
     * （这仅在调用者知道列表不包含任何空元素的情况下才能确定列表的长度。）
     */
    @SuppressWarnings("unchecked")
    public <T> T[] toArray(T[] a) {
        if (a.length < size)
            // 新建一个运行时类型的数组，但是ArrayList数组的内容
            return (T[]) Arrays.copyOf(elementData, size, a.getClass());
        //调用System提供的arraycopy()方法实现数组之间的复制
        System.arraycopy(elementData, 0, a, 0, size);
        if (a.length > size)
            a[size] = null;
        return a;
    }

    // Positional Access Operations

    @SuppressWarnings("unchecked")
    E elementData(int index) {
        return (E) elementData[index];
    }

    /**
     * 返回此列表中指定位置的元素。
     */
    public E get(int index) {
        rangeCheck(index);

        return elementData(index);
    }

    /**
     * 用指定的元素替换此列表中指定位置的元素。
     */
    public E set(int index, E element) {
        //对index进行界限检查
        rangeCheck(index);

        E oldValue = elementData(index);
        elementData[index] = element;
        //返回原来在这个位置的元素
        return oldValue;
    }

    /**
     * 将指定的元素追加到此列表的末尾。
     */
    public boolean add(E e) {
        ensureCapacityInternal(size + 1); // Increments modCount!!
        //这里看到ArrayList添加元素的实质就相当于为数组赋值
        elementData[size++] = e;
        return true;
    }

    /**
     * 在此列表中的指定位置插入指定的元素。
     * 先调用 rangeCheckForAdd 对index进行界限检查；然后调用 ensureCapacityInternal 方法保证capacity足够大；
     * 再将从index开始之后的所有成员后移一个位置；将element插入index位置；最后size加1。
     */

```

```

public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal(size + 1); // Increments modCount!!
    //arraycopy()这个实现数组之间复制的方法一定要看一下，下面就用到了arraycopy()方法实现数组自己复制自己
    System.arraycopy(elementData, index, elementData, index + 1,
                    size - index);
    elementData[index] = element;
    size++;
}

/**
 * 删除该列表中指定位置的元素。 将任何后续元素移动到左侧（从其索引中减去一个元素）。
 */
public E remove(int index) {
    rangeCheck(index);

    modCount++;
    E oldValue = elementData(index);

    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                        numMoved);
    elementData[--size] = null; // clear to let GC do its work
    //从列表中删除的元素
    return oldValue;
}

/**
 * 从列表中删除指定元素的第一个出现（如果存在）。 如果列表不包含该元素，则它不会更改。
 * 返回true, 如果此列表包含指定的元素
 */
public boolean remove(Object o) {
    if (o == null) {
        for (int index = 0; index < size; index++)
            if (elementData[index] == null)
                fastRemove(index);
        return true;
    } else {
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index]))
                fastRemove(index);
        return true;
    }
    return false;
}

/*
 * Private remove method that skips bounds checking and does not
 * return the value removed.
 */
private void fastRemove(int index) {
    modCount++;
    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                        numMoved);
    elementData[--size] = null; // clear to let GC do its work
}

/**
 * 从列表中删除所有元素。
 */
public void clear() {
    modCount++;
}

```

```

        // 把数组中所有的元素的值设为null
        for (int i = 0; i < size; i++)
            elementData[i] = null;

        size = 0;
    }

    /**
     * 按指定集合的Iterator返回的顺序将指定集合中的所有元素追加到此列表的末尾。
     */
    public boolean addAll(Collection<? extends E> c) {
        Object[] a = c.toArray();
        int numNew = a.length;
        ensureCapacityInternal(size + numNew); // Increments modCount
        System.arraycopy(a, 0, elementData, size, numNew);
        size += numNew;
        return numNew != 0;
    }

    /**
     * 将指定集合中的所有元素插入到此列表中，从指定的位置开始。
     */
    public boolean addAll(int index, Collection<? extends E> c) {
        rangeCheckForAdd(index);

        Object[] a = c.toArray();
        int numNew = a.length;
        ensureCapacityInternal(size + numNew); // Increments modCount

        int numMoved = size - index;
        if (numMoved > 0)
            System.arraycopy(elementData, index, elementData, index + numNew,
                            numMoved);

        System.arraycopy(a, 0, elementData, index, numNew);
        size += numNew;
        return numNew != 0;
    }

    /**
     * 从此列表中删除所有索引为fromIndex（含）和toIndex之间的元素。
     * 将任何后续元素移动到左侧（减少其索引）。
     */
    protected void removeRange(int fromIndex, int toIndex) {
        modCount++;
        int numMoved = size - toIndex;
        System.arraycopy(elementData, toIndex, elementData, fromIndex,
                        numMoved);

        // clear to let GC do its work
        int newSize = size - (toIndex-fromIndex);
        for (int i = newSize; i < size; i++) {
            elementData[i] = null;
        }
        size = newSize;
    }

    /**
     * 检查给定的索引是否在范围内。
     */
    private void rangeCheck(int index) {
        if (index >= size)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }

    /**
     * add和addAll使用的rangeCheck的一个版本
     */
    private void rangeCheckForAdd(int index) {

```

```

        if (index > size || index < 0)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }

    /**
     * 返回IndexOutOfBoundsException细节信息
     */
    private String outOfBoundsMsg(int index) {
        return "Index: "+index+", Size: "+size;
    }

    /**
     * 从此列表中删除指定集合中包含的所有元素。
     */
    public boolean removeAll(Collection<?> c) {
        Objects.requireNonNull(c);
        //如果此列表被修改则返回true
        return batchRemove(c, false);
    }

    /**
     * 仅保留此列表中包含在指定集合中的元素。
     *换句话说，从此列表中删除其中不包含在指定集合中的所有元素。
     */
    public boolean retainAll(Collection<?> c) {
        Objects.requireNonNull(c);
        return batchRemove(c, true);
    }

    /**
     * 从列表中的指定位置开始，返回列表中的元素（按正确顺序）的列表迭代器。
     *指定的索引表示初始调用将返回的第一个元素为next 。 初始调用previous将返回指定索引减1的元素。
     *返回的列表迭代器是fail-fast 。
     */
    public ListIterator<E> listIterator(int index) {
        if (index < 0 || index > size)
            throw new IndexOutOfBoundsException("Index: "+index);
        return new ListItr(index);
    }

    /**
     *返回列表中的列表迭代器（按适当的顺序）。
     *返回的列表迭代器是fail-fast 。
     */
    public ListIterator<E> listIterator() {
        return new ListItr(0);
    }

    /**
     *以正确的顺序返回该列表中的元素的迭代器。
     *返回的迭代器是fail-fast 。
     */
    public Iterator<E> iterator() {
        return new Itr();
    }
}

```

ArrayList源码分析

System.arraycopy()和Arrays.copyOf()方法

通过上面源码我们发现这两个实现数组复制的方法被广泛使用而且很多地方都特别巧妙。比如下面add(int index, E element)方法就很巧妙的用到了arraycopy()方法让数组自己复制自己实现让index开始之后的所有成员后移一个位置：

```
/**
```

```

    * 在此列表中的指定位置插入指定的元素。
    *先调用 rangeCheckForAdd 对index进行界限检查；然后调用 ensureCapacityInternal 方法保证capacity足够大；
    *再将从index开始之后的所有成员后移一个位置；将element插入index位置；最后size加1。
    */
    public void add(int index, E element) {
        rangeCheckForAdd(index);

        ensureCapacityInternal(size + 1); // Increments modCount!!
        //arraycopy()方法实现数组自己复制自己
        //elementData:源数组;index:源数组中的起始位置;elementData: 目标数组; index + 1: 目标数组中的起始位置; size - index: 要复制的数组元素的数量;
        System.arraycopy(elementData, index, elementData, index + 1, size - index);
        elementData[index] = element;
        size++;
    }
}

```

又如toArray()方法中用到了copyOf()方法

```

    /**
     *以正确的顺序（从第一个到最后一个元素）返回一个包含此列表中所有元素的数组。
     *返回的数组将是“安全的”，因为该列表不保留对它的引用。（换句话说，这个方法必须分配一个新的数组）。
     *因此，调用者可以自由地修改返回的数组。此方法充当基于阵列和基于集合的API之间的桥梁。
     */
    public Object[] toArray() {
        //elementData: 要复制的数组; size: 要复制的长度
        return Arrays.copyOf(elementData, size);
    }
}

```

两者联系与区别

联系：看两者源代码可以发现 copyOf() 内部调用了 System.arraycopy() 方法 区别：

1. arraycopy()需要目标数组，将原数组拷贝到你自己定义的数组里，而且可以选择拷贝的起点和长度以及放入新数组中的位置
2. copyOf()是系统自动在内部新建一个数组，并返回该数组。

ArrayList 核心扩容技术

```java //下面是ArrayList的扩容机制 //ArrayList的扩容机制提高了性能，如果每次只扩充一个， //那么频繁的插入会导致频繁的拷贝，降低性能，而ArrayList的扩容机制避免了这种情况。 /\*\*

- 如有必要，增加此ArrayList实例的容量，以确保它至少能容纳元素的数量
- @param minCapacity 所需的最小容量 \*/ public void ensureCapacity(int minCapacity) { int minExpand = (elementData != DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA)

```

 // any size if not default element table
 ? 0
 // larger than default for default empty table. It's already
 // supposed to be at default size.
 : DEFAULT_CAPACITY;
 }
}

```

```
if (minCapacity > minExpand) {
```

```
 ensureExplicitCapacity(minCapacity);
}
```

```
} //得到最小扩容量 private void ensureCapacityInternal(int minCapacity) { if (elementData ==
DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
```

```

 // 获取默认的容量和传入参数的较大值
 minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);

 }

 ensureExplicitCapacity(minCapacity); } //判断是否需要扩容,上面两个方法都要调用 private void
 ensureExplicitCapacity(int minCapacity) { modCount++;

 //如果说minCapacity也就是所需的最小容量大于保存ArrayList数据的数组的长度的话,就需要调用
 grow(minCapacity)方法扩容。//这个minCapacity到底为多少呢?举个例子在添加元素(add)方法中这个
 minCapacity的大小就为现在数组的长度加1 if (minCapacity - elementData.length > 0)

 //调用grow方法进行扩容, 调用此方法代表已经开始扩容了
 grow(minCapacity);

 }

````java
/**
 * ArrayList扩容的核心方法。
 */
private void grow(int minCapacity) {
    //elementData为保存ArrayList数据的数组
    //elementData.length求数组长度elementData.size是求数组中的元素个数
    // oldCapacity为旧容量, newCapacity为新容量
    int oldCapacity = elementData.length;
    //将oldCapacity 右移一位, 其效果相当于oldCapacity /2,
    //我们知道位运算的速度远远快于整除运算, 整句运算式的结果就是将新容量更新为旧容量的1.5倍,
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    //然后检查新容量是否大于最小需要容量, 若还是小于最小需要容量, 那么就把最小需要容量当作数组的新容量,
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    //再检查新容量是否超出了ArrayList所定义的最大容量,
    //若超出了, 则调用hugeCapacity()来比较minCapacity和 MAX_ARRAY_SIZE,
    //如果minCapacity大于最大容量, 则新容量则为ArrayList定义的最大容量, 否则, 新容量大小则为 minCapacity。
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

```

扩容机制代码已经做了详细的解释。另外值得注意的是大家很容易忽略的一个运算符：**移位运算符** 简介：移位运算符就是在二进制的基础上对数字进行平移。按照平移的方向和填充数字的规则分为三种：**<<(左移)**、**>>(带符号右移)**和**>>>(无符号右移)**。**作用：**对于大数据的2进制运算,位移运算符比那些普通运算符的运算要快很多,因为程序仅仅移动一下而已,不去计算,这样提高了效率,节省了资源。比如这里：int newCapacity = oldCapacity + (oldCapacity >> 1); 右移一位相当于除2, 右移n位相当于除以2的n次方。这里 oldCapacity 明显右移了1位所以相当于oldCapacity /2。

另外需要注意的是：

1. java 中的**length** 属性是针对数组说的,比如说你声明了一个数组,想知道这个数组的长度则用到了 length 这个属性.
2. java 中的**length()**方法是针对字符串String说的,如果想看这个字符串的长度则用到 length()这个方法.
3. java 中的**size()**方法是针对泛型集合说的,如果想看这个泛型有多少个元素,就调用此方法来查看!

## 内部类

```
(1)private class Itr implements Iterator<E>
```

```
(2)private class ListItr extends Itr implements ListIterator<E>
(3)private class SubList extends AbstractList<E> implements RandomAccess
(4)static final class ArrayListSpliterator<E> implements Spliterator<E>
```

ArrayList有四个内部类，其中的Itr是实现了Iterator接口，同时重写了里面的hasNext(), next(), remove()等方法；其中的ListItr继承Itr，实现了ListIterator接口，同时重写了hasPrevious(), nextIndex(), previousIndex(), previous(), set(E e), add(E e)等方法，所以这也看出了Iterator和ListIterator的区别：ListIterator在Iterator的基础上增加了添加对象，修改对象，逆向遍历等方法，这些都是Iterator不能实现的。

## ArrayList经典Demo

```
package list;
import java.util.ArrayList;
import java.util.Iterator;

public class ArrayListDemo {

 public static void main(String[] args){
 ArrayList<Integer> arrayList = new ArrayList<Integer>();

 System.out.printf("Before add:arrayList.size() = %d\n",arrayList.size());

 arrayList.add(1);
 arrayList.add(3);
 arrayList.add(5);
 arrayList.add(7);
 arrayList.add(9);
 System.out.printf("After add:arrayList.size() = %d\n",arrayList.size());

 System.out.println("Printing elements of arrayList");
 // 三种遍历方式打印元素
 // 第一种：通过迭代器遍历
 System.out.print("通过迭代器遍历:");
 Iterator<Integer> it = arrayList.iterator();
 while(it.hasNext()){
 System.out.print(it.next() + " ");
 }
 System.out.println();

 // 第二种：通过索引值遍历
 System.out.print("通过索引值遍历:");
 for(int i = 0; i < arrayList.size(); i++){
 System.out.print(arrayList.get(i) + " ");
 }
 System.out.println();

 // 第三种：for循环遍历
 System.out.print("for循环遍历:");
 for(Integer number : arrayList){
 System.out.print(number + " ");
 }

 // toArray用法
 // 第一种方式(最常用)
 Integer[] integer = arrayList.toArray(new Integer[0]);

 // 第二种方式(容易理解)
 Integer[] integer1 = new Integer[arrayList.size()];
 arrayList.toArray(integer1);

 // 抛出异常，java不支持向下转型
 // Integer[] integer2 = new Integer[arrayList.size()];
 // integer2 = arrayList.toArray();
 System.out.println();
 }
}
```

```
// 在指定位置添加元素
arrayList.add(2, 2);
// 删除指定位置上的元素
arrayList.remove(2);
// 判断指定元素
arrayList.remove((Object)3);
// 判断arrayList是否包含5
System.out.println("ArrayList contains 5 is: " + arrayList.contains(5));

// 清空ArrayList
arrayList.clear();
// 判断ArrayList是否为空
System.out.println("ArrayList is empty: " + arrayList.isEmpty());
}
}
```

## 一 先从 ArrayList 的构造函数说起

ArrayList有三种方式来初始化，构造方法源码如下：

```

 /**
 * 默认初始容量大小
 */
 private static final int DEFAULT_CAPACITY = 10;

 private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

 /**
 * 默认构造函数，使用初始容量10构造一个空列表(无参数构造)
 */
 public ArrayList() {
 this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
 }

 /**
 * 带初始容量参数的构造函数。 (用户自己指定容量)
 */
 public ArrayList(int initialCapacity) {
 if (initialCapacity > 0) { //初始容量大于0
 //创建initialCapacity大小的数组
 this.elementData = new Object[initialCapacity];
 } else if (initialCapacity == 0) { //初始容量等于0
 //创建空数组
 this.elementData = EMPTY_ELEMENTDATA;
 } else { //初始容量小于0, 抛出异常
 throw new IllegalArgumentException("Illegal Capacity: " +
 initialCapacity);
 }
 }

 /**
 * 构造包含指定collection元素的列表，这些元素利用该集合的迭代器按顺序返回
 * 如果指定的集合为null, throws NullPointerException。
 */
 public ArrayList(Collection<? extends E> c) {
 elementData = c.toArray();
 if ((size = elementData.length) != 0) {
 // c.toArray might (incorrectly) not return Object[] (see 6260652)
 if (elementData.getClass() != Object[].class)
 elementData = Arrays.copyOf(elementData, size, Object[].class);
 } else {
 // replace with empty array.
 this.elementData = EMPTY_ELEMENTDATA;
 }
 }
}

```

细心的同学一定会发现：以无参数构造方法创建 ArrayList 时，实际上初始化赋值的是一个空数组。当真正对数组进行添加元素操作时，才真正分配容量。即向数组中添加第一个元素时，数组容量扩为10。下面在我们分析 ArrayList 扩容时会降到这一点内容！

## 二 一步一步分析 ArrayList 扩容机制

这里以无参构造函数创建的 ArrayList 为例分析

## 1. 先来看 add 方法

```

 /**
 * 将指定的元素追加到此列表的末尾。
 */
 public boolean add(E e) {
 //添加元素之前，先调用ensureCapacityInternal方法
 ensureCapacityInternal(size + 1); // Increments modCount!!
 //这里看到ArrayList添加元素的实质就相当于为数组赋值
 elementData[size++] = e;
 return true;
 }

```

## 2. 再来看看 ensureCapacityInternal() 方法

可以看到 add 方法首先调用了 ensureCapacityInternal(size + 1)

```

//得到最小扩容量
private void ensureCapacityInternal(int minCapacity) {
 if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
 // 获取默认的容量和传入参数的较大值
 minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
 }

 ensureExplicitCapacity(minCapacity);
}

```

当要 add 进第1个元素时，minCapacity为1，在Math.max()方法比较后，minCapacity 为10。

## 3. ensureExplicitCapacity() 方法

如果调用 ensureCapacityInternal() 方法就一定会进过（执行）这个方法，下面我们来研究一下这个方法的源码！

```

//判断是否需要扩容
private void ensureExplicitCapacity(int minCapacity) {
 modCount++;

 // overflow-conscious code
 if (minCapacity - elementData.length > 0)
 //调用grow方法进行扩容，调用此方法代表已经开始扩容了
 grow(minCapacity);
}

```

我们来仔细分析一下：

- 当我们要 add 进第1个元素到 ArrayList 时，elementData.length 为0（因为还是一个空的 list），因为执行了 ensureCapacityInternal() 方法，所以 minCapacity 此时为10。此时，`minCapacity - elementData.length > 0` 成立，所以会进入 grow(minCapacity) 方法。
- 当add第2个元素时，minCapacity 为2，此时`elementData.length`(容量)在添加第一个元素后扩容成 10 了。此时，`minCapacity - elementData.length > 0` 不成立，所以不会进入（执行） grow(minCapacity) 方法。
- 添加第3、4…到第10个元素时，依然不会执行grow方法，数组容量都为10。

直到添加第11个元素，minCapacity(为11)比elementData.length (为10) 要大。进入grow方法进行扩容。

## 4. grow() 方法

```

 /**
 * 要分配的最大数组大小

```

```

/*
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

/**
 * ArrayList扩容的核心方法。
 */
private void grow(int minCapacity) {
 // oldCapacity为旧容量, newCapacity为新容量
 int oldCapacity = elementData.length;
 //将oldCapacity 右移一位, 其效果相当于oldCapacity /2,
 //我们知道位运算的速度远远快于整除运算, 整句运算式的结果就是将新容量更新为旧容量的1.5倍,
 int newCapacity = oldCapacity + (oldCapacity >> 1);
 //然后检查新容量是否大于最小需要容量, 若还是小于最小需要容量, 那么就把最小需要容量当作数组的新容量,
 if (newCapacity - minCapacity < 0)
 newCapacity = minCapacity;
 // 如果新容量大于 MAX_ARRAY_SIZE,进入(执行) `hugeCapacity()` 方法来比较 minCapacity 和 MAX_ARRAY_SIZE,
 //如果minCapacity大于最大容量, 则新容量则为`Integer.MAX_VALUE` , 否则, 新容量大小则为 MAX_ARRAY_SIZE 即为 `Integer.
 MAX_VALUE - 8`。
 if (newCapacity - MAX_ARRAY_SIZE > 0)
 newCapacity = hugeCapacity(minCapacity);
 // minCapacity is usually close to size, so this is a win:
 elementData = Arrays.copyOf(elementData, newCapacity);
}

```

**int newCapacity = oldCapacity + (oldCapacity >> 1),所以 ArrayList 每次扩容之后容量都会变为原来的 1.5 倍!** 记清楚了! 不是网上很多人说的 1.5 倍+1!

">>" (移位运算符) : >>1 右移一位相当于除2, 右移n位相当于除以 2 的 n 次方。这里 oldCapacity 明显右移了1位所以相当于oldCapacity /2。对于大数据的2进制运算,位移运算符比那些普通运算符的运算要快很多,因为程序仅仅移动一下而已,不去计算,这样提高了效率,节省了资源

我们再来通过例子探究一下 `grow()` 方法 :

- 当add第1个元素时, oldCapacity 为0, 经比较后第一个if判断成立, newCapacity = minCapacity(为10)。但是第二个if判断不会成立, 即newCapacity 不比 MAX\_ARRAY\_SIZE大, 则不会进入 `hugeCapacity` 方法。数组容量为10, add方法中 return true,size增为1。
- 当add第11个元素进入grow方法时, newCapacity为15, 比minCapacity (为11) 大, 第一个if判断不成立。新容量没有大于数组最大size, 不会进入hugeCapacity方法。数组容量扩为15, add方法中return true,size增为11。
- 以此类推……

这里补充一点比较重要, 但是容易被忽视掉的知识点:

- java 中的 `length` 属性是针对数组说的,比如说你声明了一个数组,想知道这个数组的长度则用到了 `length` 这个属性.
- java 中的 `length()` 方法是针对字符串说的,如果想看这个字符串的长度则用到 `length()` 这个方法.
- java 中的 `size()` 方法是针对泛型集合说的,如果想看这个泛型有多少个元素,就调用此方法来查看!

## 5. `hugeCapacity()` 方法。

从上面 `grow()` 方法源码我们知道: 如果新容量大于 MAX\_ARRAY\_SIZE,进入(执行) `hugeCapacity()` 方法来比较 `minCapacity` 和 `MAX_ARRAY_SIZE`, 如果`minCapacity`大于最大容量, 则新容量则为 `Integer.MAX_VALUE` , 否则, 新容量大小则为 `MAX_ARRAY_SIZE` 即为 `Integer.MAX_VALUE - 8` 。

```

private static int hugeCapacity(int minCapacity) {
 if (minCapacity < 0) // overflow
 throw new OutOfMemoryError();
 //对minCapacity和MAX_ARRAY_SIZE进行比较
 //若minCapacity大, 将Integer.MAX_VALUE作为新数组的大小
 //若MAX_ARRAY_SIZE大, 将MAX_ARRAY_SIZE作为新数组的大小
 //MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
 return (minCapacity > MAX_ARRAY_SIZE) ?
 Integer.MAX_VALUE :
 MAX_ARRAY_SIZE;
}

```

```
 }
```

### 三 System.arraycopy() 和 Arrays.copyOf() 方法

阅读源码的话，我们就会发现 ArrayList 中大量调用了这两个方法。比如：我们上面讲的扩容操作以及 `add(int index, E element)`、`toArray()` 等方法中都用到了该方法！

#### 3.1 System.arraycopy() 方法

```
/**
 * 在此列表中的指定位置插入指定的元素。
 * 先调用 rangeCheckForAdd 对index进行界限检查；然后调用 ensureCapacityInternal 方法保证capacity足够大；
 * 再将从index开始之后的所有成员后移一个位置；将element插入index位置；最后size加1。
 */
public void add(int index, E element) {
 rangeCheckForAdd(index);

 ensureCapacityInternal(size + 1); // Increments modCount!!
 //arraycopy()方法实现数组自己复制自己
 //elementData:源数组;index:源数组中的起始位置;elementData: 目标数组; index + 1: 目标数组中的起始位置; size - index: 要复制的数组元素的数量;
 System.arraycopy(elementData, index, elementData, index + 1, size - index);
 elementData[index] = element;
 size++;
}
```

我们写一个简单的方法测试以下：

```
public class ArraycopyTest {

 public static void main(String[] args) {
 // TODO Auto-generated method stub
 int[] a = new int[10];
 a[0] = 0;
 a[1] = 1;
 a[2] = 2;
 a[3] = 3;
 System.arraycopy(a, 2, a, 3, 3);
 a[2]=99;
 for (int i = 0; i < a.length; i++) {
 System.out.println(a[i]);
 }
 }
}
```

结果：

```
0 1 99 2 3 0 0 0 0 0
```

#### 3.2 Arrays.copyOf() 方法

```
/**
 * 以正确的顺序返回一个包含此列表中所有元素的数组（从第一个到最后一个元素）； 返回的数组的运行时类型是指定数组的运行时类型。
 */
public Object[] toArray() {
 //elementData: 要复制的数组; size: 要复制的长度
 return Arrays.copyOf(elementData, size);
```

```
 }
```

个人觉得使用 `Arrays.copyOf()` 方法主要是为了给原有数组扩容，测试代码如下：

```
public class ArraysCopyOfTest {

 public static void main(String[] args) {
 int[] a = new int[3];
 a[0] = 0;
 a[1] = 1;
 a[2] = 2;
 int[] b = Arrays.copyOf(a, 10);
 System.out.println("b.length" + b.length);
 }
}
```

结果：

```
10
```

### 3.3 两者联系和区别

联系：

看两者源代码可以发现 `copyOf()` 内部实际调用了 `System.arraycopy()` 方法

区别：

`arraycopy()` 需要目标数组，将原数组拷贝到你自己定义的数组里或者原数组，而且可以选择拷贝的起点和长度以及放入新数组中的位置 `copyOf()` 是系统自动在内部新建一个数组，并返回该数组。

## 四 ensureCapacity 方法

ArrayList 源码中有一个 `ensureCapacity` 方法不知道大家注意到没有，这个方法 ArrayList 内部没有被调用过，所以很显然是提供给用户调用的，那么这个方法有什么作用呢？

```
/**
 * 如有必要，增加此 ArrayList 实例的容量，以确保它至少可以容纳由minimum capacity参数指定的元素数。
 *
 * @param minCapacity 所需的最小容量
 */
public void ensureCapacity(int minCapacity) {
 int minExpand = (elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA)
 // any size if not default element table
 ? 0
 // larger than default for default empty table. It's already
 // supposed to be at default size.
 : DEFAULT_CAPACITY;

 if (minCapacity > minExpand) {
 ensureExplicitCapacity(minCapacity);
 }
}
```

最好在 `add` 大量元素之前用 `ensureCapacity` 方法，以减少增量从新分配的次数

我们通过下面的代码实际测试以下这个方法的效果：

```
public class EnsureCapacityTest {
```

```
public static void main(String[] args) {
 ArrayList<Object> list = new ArrayList<Object>();
 final int N = 10000000;
 long startTime = System.currentTimeMillis();
 for (int i = 0; i < N; i++) {
 list.add(i);
 }
 long endTime = System.currentTimeMillis();
 System.out.println("使用ensureCapacity方法前: "+(endTime - startTime));

 list = new ArrayList<Object>();
 long startTime1 = System.currentTimeMillis();
 list.ensureCapacity(N);
 for (int i = 0; i < N; i++) {
 list.add(i);
 }
 long endTime1 = System.currentTimeMillis();
 System.out.println("使用ensureCapacity方法后: "+(endTime1 - startTime1));
}
```

运行结果：

```
使用ensureCapacity方法前: 4637
使用ensureCapacity方法前: 241
```

通过运行结果，我们可以很明显的看出向 ArrayList 添加大量元素之前最好先使用 ensureCapacity 方法，以减少增量从新分配的次数

- 简介
- 内部结构分析
- LinkedList源码分析
  - 构造方法
  - 添加 (add) 方法
  - 根据位置取数据的方法
  - 根据对象得到索引的方法
  - 检查链表是否包含某对象的方法:
  - 删除 (remove/pop) 方法
- LinkedList类常用方法测试:

## 简介

LinkedList是一个实现了List接口和Deque接口的双端链表。 LinkedList底层的链表结构使它支持高效的插入和删除操作，另外它实现了Deque接口，使得LinkedList类也具有队列的特性； LinkedList不是线程安全的，如果想使LinkedList变成线程安全的，可以调用静态类Collections类中的synchronizedList方法： ``java List list=Collections.synchronizedList(new LinkedList(...)); ``## 内部结构分析 \*\*如下图所示： \*\* ! [LinkedList内部结构] (<https://user-gold-cdn.xitu.io/2018/3/19/1623e363fe0450b0?w=600&h=481&f=jpeg&s=18502>) 看完了图之后，我们再看LinkedList类中的一个\*\*内部私有类Node\*\*就很好理解了： ``java private static class Node { E item;//节点值 Node next;//前驱节点 Node prev;//后继节点 Node prev, E element, Node next) { this.item = element; this.next = next; this.prev = prev; } } `` 这个类就代表双端链表的节点Node。这个类有三个属性，分别是前驱节点，本节点的值，后继节点。 ## LinkedList源码分析 #### 构造方法 \*\*空构造方法：\*\* ``java public LinkedList() {} `` \*\*用已有的集合创建链表的构造方法：\*\* ``java public LinkedList(Collection c) { this(); addAll(c); } `` #### add方法 \*\*add(E e)\*\* 方法： 将元素添加到链表尾部 ``java public boolean add(E e) { linkLast(e); //这里就只调用了这一个方法 return true; } `` \*\*链接使e作为最后一个元素。\*\* ``void linkLast(E e) { final Node l = last; final Node newNode = new Node<>(l, e, null); last = newNode; //新建节点 if (l == null) first = newNode; else l.next = newNode; //指向后继元素也就是指向下一个元素 size++; modCount++; } `` \*\*add(int index,E e)\*\*： 在指定位置添加元素 ``java public void add(int index, E element) { checkPositionIndex(index); //检查索引是否处于[0-size]之间 if (index == size) //添加在链表尾部 linkLast(element); else //添加在链表中间 linkBefore(element, node(index)); } `` \*\*linkBefore方法\*\*需要给定两个参数，一个插入节点的值，一个指定的node，所以我们又调用了Node(index)去找到index对应的node

**addAll(Collection c)**: 将集合插入到链表尾部

```
public boolean addAll(Collection<? extends E> c) {
 return addAll(size, c);
}
```

**addAll(int index, Collection c)**: 将集合从指定位置开始插入

```
public boolean addAll(int index, Collection<? extends E> c) {
 //1:检查index范围是否在size之内
 checkPositionIndex(index);

 //2:toArray()方法把集合的数据存到对象数组中
 Object[] a = c.toArray();
 int numNew = a.length;
 if (numNew == 0)
 return false;

 //3: 得到插入位置的前驱节点和后继节点
 Node<E> pred, succ;
 //如果插入位置为尾部，前驱节点为last，后继节点为null
 if (index == size) {
 succ = null;
 pred = last;
 }
```

```

 //否则，调用node()方法得到后继节点，再得到前驱节点
 else {
 succ = node(index);
 pred = succ.prev;
 }

 // 4: 遍历数据将数据插入
 for (Object o : a) {
 @SuppressWarnings("unchecked") E e = (E) o;
 //创建新节点
 Node<E> newNode = new Node<>(pred, e, null);
 //如果插入位置在链表头部
 if (pred == null)
 first = newNode;
 else
 pred.next = newNode;
 pred = newNode;
 }

 //如果插入位置在尾部，重置last节点
 if (succ == null) {
 last = pred;
 }
 //否则，将插入的链表与先前链表连接起来
 else {
 pred.next = succ;
 succ.prev = pred;
 }

 size += numNew;
 modCount++;
 return true;
}

```

上面可以看出addAll方法通常包括下面四个步骤：

1. 检查index范围是否在size之内
2. toArray()方法把集合的数据存到对象数组中
3. 得到插入位置的前驱和后继节点
4. 遍历数据，将数据插入到指定位置

**addFirst(E e):** 将元素添加到链表头部

```

public void addFirst(E e) {
 linkFirst(e);
}

private void linkFirst(E e) {
 final Node<E> f = first;
 final Node<E> newNode = new Node<>(null, e, f); //新建节点，以头节点为后继节点
 first = newNode;
 //如果链表为空，last节点也指向该节点
 if (f == null)
 last = newNode;
 //否则，将头节点的前驱指针指向新节点，也就是指向前一个元素
 else
 f.prev = newNode;
 size++;
 modCount++;
}

```

**addLast(E e):** 将元素添加到链表尾部，与 **add(E e)** 方法一样

```
public void addLast(E e) {
 linkLast(e);
}
```

## 根据位置取数据的方法

**get(int index):** 根据指定索引返回数据

```
public E get(int index) {
 //检查index范围是否在size之内
 checkElementIndex(index);
 //调用Node(index)去找到index对应的node然后返回它的值
 return node(index).item;
}
```

获取头节点 (**index=0**) 数据方法:

```
public E getFirst() {
 final Node<E> f = first;
 if (f == null)
 throw new NoSuchElementException();
 return f.item;
}
public E element() {
 return getFirst();
}
public E peek() {
 final Node<E> f = first;
 return (f == null) ? null : f.item;
}

public E peekFirst() {
 final Node<E> f = first;
 return (f == null) ? null : f.item;
}
```

区别: **getFirst(),element(),peek(),peekFirst()** 这四个获取头结点方法的区别在于对链表为空时的处理, 是抛出异常还是返回null, 其中**getFirst()** 和**element()** 方法将会在链表为空时, 抛出异常

**element()**方法的内部就是使用**getFirst()**实现的。它们会在链表为空时, 抛出**NoSuchElementException** 获取尾节点 (**index=-1**) 数据方法:

```
public E getLast() {
 final Node<E> l = last;
 if (l == null)
 throw new NoSuchElementException();
 return l.item;
}
public E peekLast() {
 final Node<E> l = last;
 return (l == null) ? null : l.item;
}
```

两者区别: **getLast()** 方法在链表为空时, 会抛出**NoSuchElementException**, 而**peekLast()** 则不会, 只是会返回null。

## 根据对象得到索引的方法

**int indexOf(Object o):** 从头遍历找

```

public int indexOf(Object o) {
 int index = 0;
 if (o == null) {
 //从头遍历
 for (Node<E> x = first; x != null; x = x.next) {
 if (x.item == null)
 return index;
 index++;
 }
 } else {
 //从头遍历
 for (Node<E> x = first; x != null; x = x.next) {
 if (o.equals(x.item))
 return index;
 index++;
 }
 }
 return -1;
}

```

**int lastIndexOf(Object o):** 从尾遍历找

```

public int lastIndexOf(Object o) {
 int index = size;
 if (o == null) {
 //从尾遍历
 for (Node<E> x = last; x != null; x = x.prev) {
 index--;
 if (x.item == null)
 return index;
 }
 } else {
 //从尾遍历
 for (Node<E> x = last; x != null; x = x.prev) {
 index--;
 if (o.equals(x.item))
 return index;
 }
 }
 return -1;
}

```

**检查链表是否包含某对象的方法:**

**contains(Object o):** 检查对象o是否存在于链表中

```

public boolean contains(Object o) {
 return indexOf(o) != -1;
}

```

## 删除方法

**remove() ,removeFirst(),pop():** 删除头节点

```

public E pop() {
 return removeFirst();
}
public E remove() {
 return removeFirst();
}
public E removeFirst() {
 final Node<E> f = first;
}

```

```

 if (f == null)
 throw new NoSuchElementException();
 return unlinkFirst(f);
 }
}

```

**removeLast(),pollLast(): 删除尾节点**

```

public E removeLast() {
 final Node<E> l = last;
 if (l == null)
 throw new NoSuchElementException();
 return unlinkLast(l);
}
public E pollLast() {
 final Node<E> l = last;
 return (l == null) ? null : unlinkLast(l);
}
}

```

**区别：**removeLast()在链表为空时将抛出NoSuchElementException，而pollLast()方法返回null。

**remove(Object o): 删除指定元素**

```

public boolean remove(Object o) {
 //如果删除对象为null
 if (o == null) {
 //从头开始遍历
 for (Node<E> x = first; x != null; x = x.next) {
 //找到元素
 if (x.item == null) {
 //从链表中移除找到的元素
 unlink(x);
 return true;
 }
 }
 } else {
 //从头开始遍历
 for (Node<E> x = first; x != null; x = x.next) {
 //找到元素
 if (o.equals(x.item)) {
 //从链表中移除找到的元素
 unlink(x);
 return true;
 }
 }
 }
 return false;
}

```

当删除指定对象时，只需调用remove(Object o)即可，不过该方法一次只会删除一个匹配的对象，如果删除了匹配对象，返回true，否则false。

**unlink(Node x) 方法：**

```

E unlink(Node<E> x) {
 // assert x != null;
 final E element = x.item;
 final Node<E> next = x.next;//得到后继节点
 final Node<E> prev = x.prev;//得到前驱节点

 //删除前驱指针
 if (prev == null) {
 first = next;如果删除的节点是头节点，令头节点指向该节点的后继节点
 } else {
 prev.next = next;//将前驱节点的后继节点指向后继节点
 }
}

```

```

 x.prev = null;
 }

 //删除后继指针
 if (next == null) {
 last = prev; //如果删除的节点是尾节点,令尾节点指向该节点的前驱节点
 } else {
 next.prev = prev;
 x.next = null;
 }

 x.item = null;
 size--;
 modCount++;
 return element;
}

```

**remove(int index):** 删除指定位置的元素

```

public E remove(int index) {
 //检查index范围
 checkElementIndex(index);
 //将节点删除
 return unlink(node(index));
}

```

## LinkedList类常用方法测试

```

package list;

import java.util.Iterator;
import java.util.LinkedList;

public class LinkedListDemo {
 public static void main(String[] args) {
 //创建存放int类型的linkedList
 LinkedList<Integer> linkedList = new LinkedList<>();
 /***** linkedList的基本操作 *****/
 linkedList.addFirst(0); // 添加元素到列表开头
 linkedList.add(1); // 在列表结尾添加元素
 linkedList.add(2, 2); // 在指定位置添加元素
 linkedList.addLast(3); // 添加元素到列表结尾

 System.out.println("linkedList (直接输出) : " + linkedList);

 System.out.println("getFirst()获得第一个元素: " + linkedList.getFirst()); // 返回此列表的第一个元素
 System.out.println("getLast()获得最后一个元素: " + linkedList.getLast()); // 返回此列表的最后一个元素
 System.out.println("removeFirst()删除第一个元素并返回: " + linkedList.removeFirst()); // 移除并返回此列表的第一个元素
 System.out.println("removeLast()删除最后一个元素并返回: " + linkedList.removeLast()); // 移除并返回此列表的最后一个元素
 System.out.println("After remove:" + linkedList);
 System.out.println("contains()方法判断列表是否包含1这个元素: " + linkedList.contains(1)); // 判断此列表包含指定元素,如果是,则返回true
 System.out.println("该linkedList的大小 : " + linkedList.size()); // 返回此列表的元素个数

 /***** 位置访问操作 *****/
 System.out.println("-----");
 linkedList.set(1, 3); // 将此列表中指定位置的元素替换为指定的元素
 System.out.println("After set(1, 3):" + linkedList);
 System.out.println("get(1)获得指定位置 (这里为1) 的元素: " + linkedList.get(1)); // 返回此列表中指定位置处的元素

 /***** Search操作 *****/
 System.out.println("-----");

```

```

linkedList.add(3);
System.out.println("indexOf(3): " + linkedList.indexOf(3)); // 返回此列表中首次出现的指定元素的索引
System.out.println("lastIndexOf(3): " + linkedList.lastIndexOf(3)); // 返回此列表中最后出现的指定元素的索引

/**************** Queue操作 *****/
System.out.println("-----");
System.out.println("peek(): " + linkedList.peek()); // 获取但不移除此列表的头
System.out.println("element(): " + linkedList.element()); // 获取但不移除此列表的头
linkedList.poll(); // 获取并移除此列表的头
System.out.println("After poll(): " + linkedList);
linkedList.remove();
System.out.println("After remove(): " + linkedList); // 获取并移除此列表的头
linkedList.offer(4);
System.out.println("After offer(4): " + linkedList); // 将指定元素添加到此列表的末尾

/**************** Deque操作 *****/
System.out.println("-----");
linkedList.offerFirst(2); // 在此列表的开头插入指定的元素
System.out.println("After offerFirst(2): " + linkedList);
linkedList.offerLast(5); // 在此列表末尾插入指定的元素
System.out.println("After offerLast(5): " + linkedList);
System.out.println("peekFirst(): " + linkedList.peekFirst()); // 获取但不移除此列表的第一个元素
System.out.println("peekLast(): " + linkedList.peekLast()); // 获取但不移除此列表的第一个元素
linkedList.pollFirst(); // 获取并移除此列表的第一个元素
System.out.println("After pollFirst(): " + linkedList);
linkedList.pollLast(); // 获取并移除此列表的最后一个元素
System.out.println("After pollLast(): " + linkedList);
linkedList.push(2); // 将元素推入此列表所表示的堆栈 (插入到列表的头)
System.out.println("After push(2): " + linkedList);
linkedList.pop(); // 从此列表所表示的堆栈处弹出一个元素 (获取并移除列表第一个元素)
System.out.println("After pop(): " + linkedList);
linkedList.add(3);
linkedList.removeFirstOccurrence(3); // 从此列表中移除第一次出现的指定元素 (从头部到尾部遍历列表)
System.out.println("After removeFirstOccurrence(3): " + linkedList);
linkedList.removeLastOccurrence(3); // 从此列表中移除最后一次出现的指定元素 (从头部到尾部遍历列表)
System.out.println("After removeLastOccurrence(3): " + linkedList);

/**************** 遍历操作 *****/
System.out.println("-----");
linkedList.clear();
for (int i = 0; i < 100000; i++) {
 linkedList.add(i);
}
// 迭代器遍历
long start = System.currentTimeMillis();
Iterator<Integer> iterator = linkedList.iterator();
while (iterator.hasNext()) {
 iterator.next();
}
long end = System.currentTimeMillis();
System.out.println("Iterator: " + (end - start) + " ms");

// 顺序遍历(随机遍历)
start = System.currentTimeMillis();
for (int i = 0; i < linkedList.size(); i++) {
 linkedList.get(i);
}
end = System.currentTimeMillis();
System.out.println("for: " + (end - start) + " ms");

// 另一种for循环遍历
start = System.currentTimeMillis();
for (Integer i : linkedList)
;
end = System.currentTimeMillis();
System.out.println("for2: " + (end - start) + " ms");

// 通过pollFirst()或pollLast()来遍历LinkedList
LinkedList<Integer> temp1 = new LinkedList<>();

```

```
temp1.addAll(linkedList);
start = System.currentTimeMillis();
while (temp1.size() != 0) {
 temp1.pollFirst();
}
end = System.currentTimeMillis();
System.out.println("pollFirst()或pollLast(): " + (end - start) + " ms");

// 通过removeFirst()或removeLast()来遍历LinkedList
LinkedList<Integer> temp2 = new LinkedList<>();
temp2.addAll(linkedList);
start = System.currentTimeMillis();
while (temp2.size() != 0) {
 temp2.removeFirst();
}
end = System.currentTimeMillis();
System.out.println("removeFirst()或removeLast(): " + (end - start) + " ms");
}

}
```

- [HashMap 简介](#)
- [底层数据结构分析](#)
  - [JDK1.8之前](#)
  - [JDK1.8之后](#)
- [HashMap源码分析](#)
  - [构造方法](#)
  - [put方法](#)
  - [get方法](#)
  - [resize方法](#)
- [HashMap常用方法测试](#)

感谢 [changfubai](#) 对本文的改进做出的贡献！

## HashMap 简介

HashMap 主要用来存放键值对，它基于哈希表的Map接口实现，是常用的Java集合之一。

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）时，将链表转化为红黑树，以减少搜索时间。

## 底层数据结构分析

### 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);
}
```

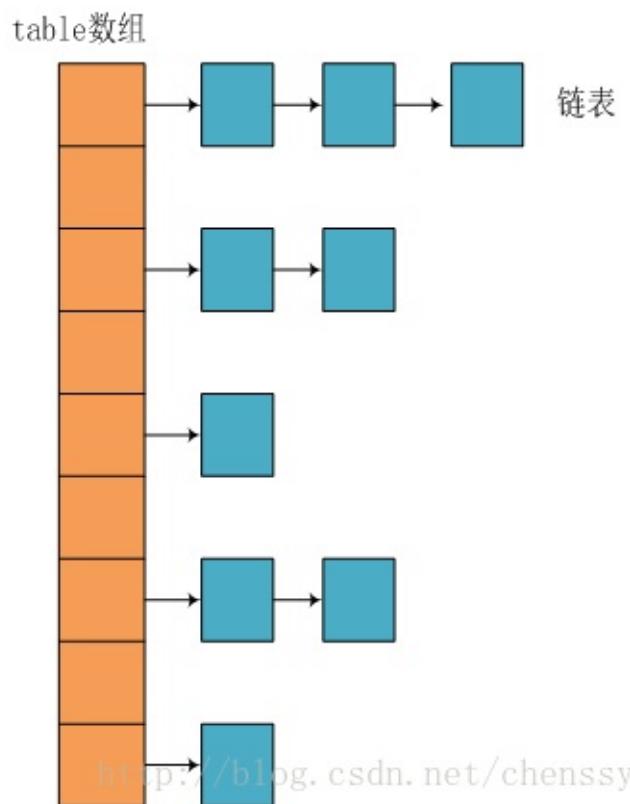
对比一下 JDK1.7 的 HashMap 的 hash 方法源码.

```
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);
}
```

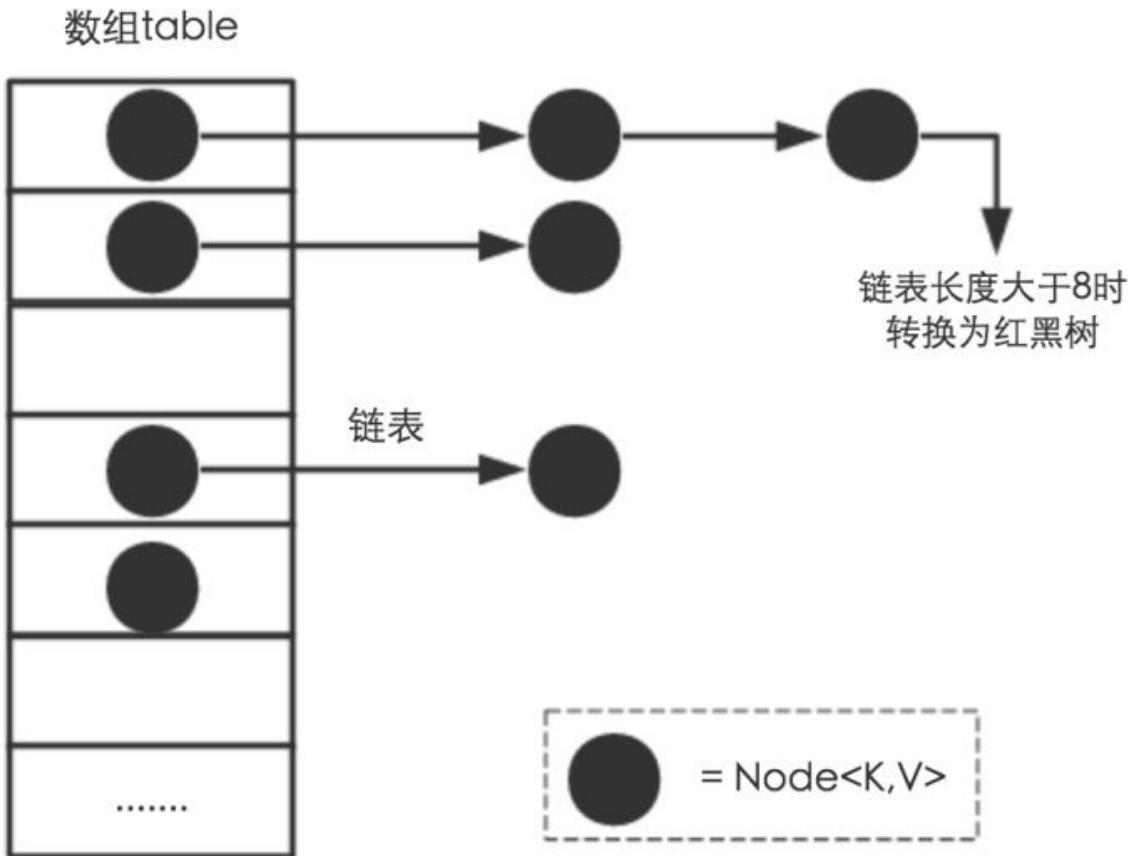
相比于 JDK1.8 的 hash 方法，JDK 1.7 的 hash 方法的性能会稍差一点点，因为毕竟扰动了 4 次。

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，以减少搜索时间。



## 类的属性:

```

public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable {
 // 序列号
 private static final long serialVersionUID = 362498820763181265L;
 // 默认的初始容量是16
 static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;
 // 最大容量
 static final int MAXIMUM_CAPACITY = 1 << 30;
 // 默认的填充因子
 static final float DEFAULT_LOAD_FACTOR = 0.75f;
 // 当桶(bucket)上的结点数大于这个值时会转成红黑树
 static final int TREEIFY_THRESHOLD = 8;
 // 当桶(bucket)上的结点数小于这个值时树转链表
 static final int UNTREEIFY_THRESHOLD = 6;
 // 桶中结构转化为红黑树对应的table的最小大小
 static final int MIN_TREEIFY_CAPACITY = 64;
 // 存放元素的数组，总是2的幂次倍
 transient Node<K,V>[] table;
 // 存放具体元素的集
 transient Set<map.entry<K,V>> entrySet;
 // 存放元素的个数，注意这个不等于数组的长度。
 transient int size;
 // 每次扩容和更改map结构的计数器
 transient int modCount;
 // 临界值 当实际大小(容量*填充因子)超过临界值时，会进行扩容
 int threshold;
 // 填充因子
 final float loadFactor;
}

```

- **loadFactor**加载因子

loadFactor加载因子是控制数组存放数据的疏密程度，loadFactor越趋近于1，那么数组中存放的数据(entry)也就越多，也就越密，也就是会让链表的长度增加，load Factor越小，也就是趋近于0，

**loadFactor**太大导致查找元素效率低，太小导致数组的利用率低，存放的数据会很分散。**loadFactor**的默认值为0.75f是官方给出的一个比较好的临界值。

- **threshold**

**threshold = capacity \* loadFactor**, 当**Size>=threshold**的时候，那么就要考虑对数组的扩增了，也就是说，这个的意思就是衡量数组是否需要扩增的一个标准。

### Node节点类源码:

```
// 继承自 Map.Entry<K,V>
static class Node<K,V> implements Map.Entry<K,V> {
 final int hash;// 哈希值，存放元素到hashmap中时用来与其他元素hash值比较
 final K key;//键
 V value;//值
 // 指向下一个节点
 Node<K,V> next;
 Node(int hash, K key, V value, Node<K,V> next) {
 this.hash = hash;
 this.key = key;
 this.value = value;
 this.next = next;
 }
 public final K getKey() { return key; }
 public final V getValue() { return value; }
 public final String toString() { return key + "=" + value; }
 // 重写hashCode()方法
 public final int hashCode() {
 return Objects.hashCode(key) ^ Objects.hashCode(value);
 }

 public final V setValue(V newValue) {
 V oldValue = value;
 value = newValue;
 return oldValue;
 }
 // 重写 equals() 方法
 public final boolean equals(Object o) {
 if (o == this)
 return true;
 if (o instanceof Map.Entry) {
 Map.Entry<?,?> e = (Map.Entry<?,?>)o;
 if (Objects.equals(key, e.getKey()) &&
 Objects.equals(value, e.getValue()))
 return true;
 }
 return false;
 }
}
```

### 树节点类源码:

```
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
 TreeNode<K,V> parent; // 父
 TreeNode<K,V> left; // 左
 TreeNode<K,V> right; // 右
 TreeNode<K,V> prev; // needed to unlink next upon deletion
 boolean red; // 判断颜色
 TreeNode(int hash, K key, V val, Node<K,V> next) {
 super(hash, key, val, next);
 }
 // 返回根节点
}
```

```

final TreeNode<K,V> root() {
 for (TreeNode<K,V> r = this, p;;) {
 if ((p = r.parent) == null)
 return r;
 r = p;
 }
}

```

## HashMap源码分析

### 构造方法

```

+-- HashMap
 C HashMap()
 C HashMap(int)
 C HashMap(int, float)
 C HashMap(Map<? extends K, ? extends V>)

```

```

// 默认构造函数。
public Map<? extends K, ? extends V>() {
 this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}

// 包含另一个“Map”的构造函数
public Map<? extends K, ? extends V>(Map<? extends K, ? extends V> m) {
 this.loadFactor = DEFAULT_LOAD_FACTOR;
 putMapEntries(m, false); //下面会分析到这个方法
}

// 指定“容量大小”的构造函数
public Map<? extends K, ? extends V>(int initialCapacity) {
 this(initialCapacity, DEFAULT_LOAD_FACTOR);
}

// 指定“容量大小”和“加载因子”的构造函数
public Map<? extends K, ? extends V>(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);
}

```

### putMapEntries方法：

```

final void putMapEntries(Map<? extends K, ? extends V> m, boolean evict) {
 int s = m.size();
 if (s > 0) {
 // 判断table是否已经初始化
 if (table == null) { // pre-size
 // 未初始化，s为m的实际元素个数
 float ft = ((float)s / loadFactor) + 1.0F;
 int t = ((ft < (float)MAXIMUM_CAPACITY) ?
 (int)ft : MAXIMUM_CAPACITY);
 // 计算得到的t大于阈值，则初始化阈值
 if (t > threshold)
 threshold = tableSizeFor(t);
 }
 // 已初始化，并且m元素个数大于阈值，进行扩容处理
 else if (s > threshold)

```

```

 resize();
 // 将m中的所有元素添加至HashMap中
 for (Map.Entry<? extends K, ? extends V> e : m.entrySet()) {
 K key = e.getKey();
 V value = e.getValue();
 putVal(hash(key), key, value, false, evict);
 }
}
}
}

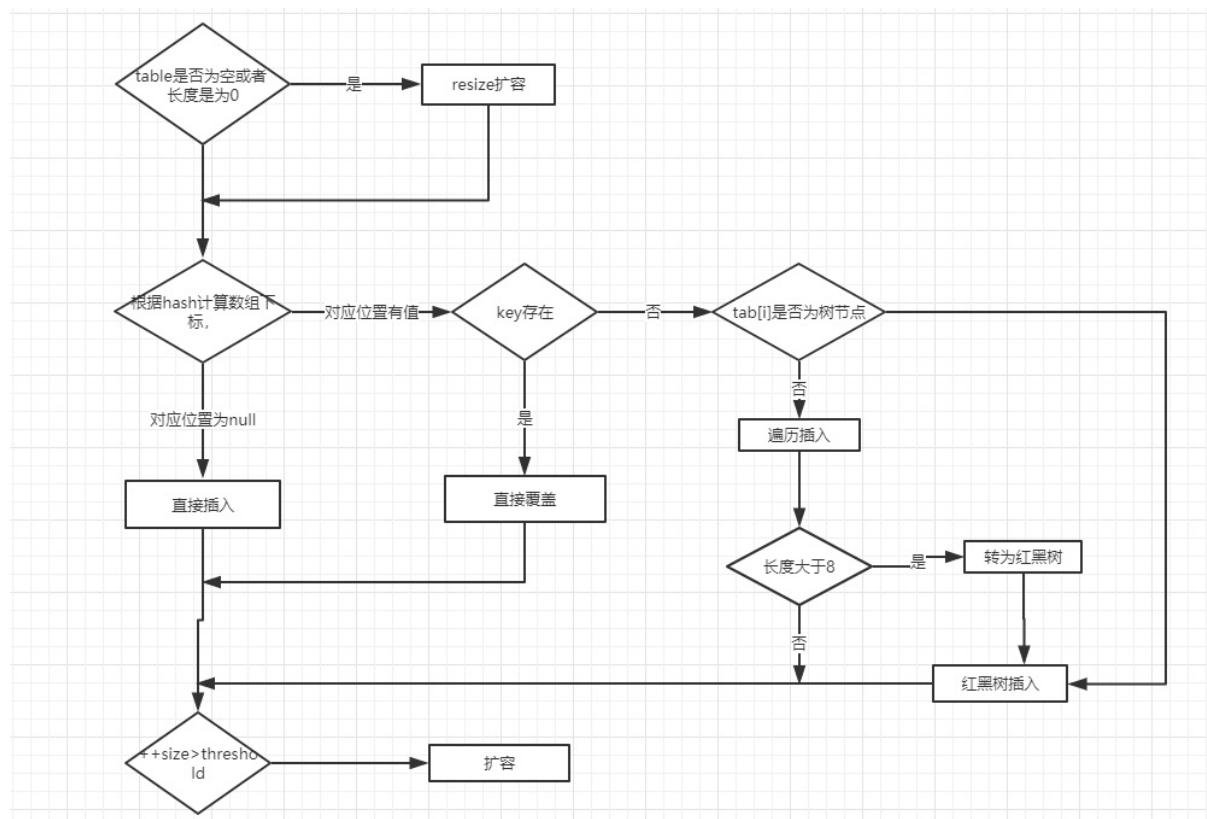
```

## put方法

HashMap只提供了put用于添加元素，putVal方法只是给put方法调用的一个方法，并没有提供给用户使用。

对putVal方法添加元素的分析如下：

- ①如果定位到的数组位置没有元素就直接插入。
- ②如果定位到的数组位置有元素就和要插入的key比较，如果key相同就直接覆盖，如果key不相同，就判断p是否是一个树节点，如果是就调用 `e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value)` 将元素添加进入。如果不是就遍历链表插入。



```

public V put(K key, V value) {
 return putVal(hash(key), key, value, false, true);
}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
 boolean evict) {
 Node<K,V>[] tab; Node<K,V> p; int n, i;
 // table未初始化或者长度为0, 进行扩容
 if ((tab = table) == null || (n = tab.length) == 0)
 n = (tab = resize()).length;
 // (n - 1) & hash 确定元素存放在哪个桶中, 桶为空, 新生成结点放入桶中(此时, 这个结点是放在数组中)
 if ((p = tab[i = (n - 1) & hash]) == null)
 tab[i] = newNode(hash, key, value, null);
}

```

```

// 桶中已经存在元素
else {
 Node<K,V> e; K k;
 // 比较桶中第一个元素(数组中的结点)的hash值相等, key相等
 if (p.hash == hash &&
 ((k = p.key) == key || (key != null && key.equals(k))))
 // 将第一个元素赋值给e, 用e来记录
 e = p;
 // hash值不相等, 即key不相等; 为红黑树结点
 else if (p instanceof TreeNode)
 // 放入树中
 e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
 // 为链表结点
 else {
 // 在链表最末插入结点
 for (int binCount = 0; ; ++binCount) {
 // 到达链表的尾部
 if ((e = p.next) == null) {
 // 在尾部插入新结点
 p.next = newNode(hash, key, value, null);
 // 结点数量达到阈值, 转化为红黑树
 if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
 treeifyBin(tab, hash);
 // 跳出循环
 break;
 }
 // 判断链表中结点的key值与插入的元素的key值是否相等
 if (e.hash == hash &&
 ((k = e.key) == key || (key != null && key.equals(k))))
 // 相等, 跳出循环
 break;
 // 用于遍历桶中的链表, 与前面的e = p.next组合, 可以遍历链表
 p = e;
 }
 }
 // 表示在桶中找到key值、hash值与插入元素相等的结点
 if (e != null) {
 // 记录e.value
 V oldValue = e.value;
 // onlyIfAbsent为false或者旧值为null
 if (!onlyIfAbsent || oldValue == null)
 // 用新值替换旧值
 e.value = value;
 // 访问后回调
 afterNodeAccess(e);
 // 返回旧值
 return oldValue;
 }
}
// 结构性修改
++modCount;
// 实际大小大于阈值则扩容
if (++size > threshold)
 resize();
// 插入后回调
afterNodeInsertion(evict);
return null;
}

```

我们再来对比一下 **JDK1.7 put方法的代码**

对于**put方法的分析如下:**

- ①如果定位到的数组位置没有元素就直接插入。
- ②如果定位到的数组位置有元素, 遍历以这个元素为头结点的链表, 依次和插入的key比较, 如果key相同就直接覆盖, 不同就采用头插法插入元素。

```

public V put(K key, V value) {
 if (table == EMPTY_TABLE) {
 inflateTable(threshold);
 }
 if (key == null)
 return putForNullKey(value);
 int hash = hash(key);
 int i = indexFor(hash, table.length);
 for (Entry<K,V> e = table[i]; e != null; e = e.next) { // 先遍历
 Object k;
 if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
 V oldValue = e.value;
 e.value = value;
 e.recordAccess(this);
 return oldValue;
 }
 }
 modCount++;
 addEntry(hash, key, value, i); // 再插入
 return null;
}

```

## get方法

```

public V get(Object key) {
 Node<K,V> e;
 return (e = getNode(hash(key), key)) == null ? null : e.value;
}

final Node<K,V> getNode(int hash, Object key) {
 Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
 if ((tab = table) != null && (n = tab.length) > 0 &&
 (first = tab[(n - 1) & hash]) != null) {
 // 数组元素相等
 if (first.hash == hash && // always check first node
 ((k = first.key) == key || (key != null && key.equals(k))))
 return first;
 // 桶中不止一个节点
 if ((e = first.next) != null) {
 // 在树中get
 if (first instanceof TreeNode)
 return ((TreeNode<K,V>)first).getTreeNode(hash, key);
 // 在链表中get
 do {
 if (e.hash == hash &&
 ((k = e.key) == key || (key != null && key.equals(k))))
 return e;
 } while ((e = e.next) != null);
 }
 }
 return null;
}

```

## resize方法

进行扩容，会伴随着一次重新hash分配，并且会遍历hash表中所有的元素，是非常耗时的。在编写程序中，要尽量避免resize。

```

final Node<K,V>[] resize() {
 Node<K,V>[] oldTab = table;
 int oldCap = (oldTab == null) ? 0 : oldTab.length;
 int oldThr = threshold;

```

```

int newCap, newThr = 0;
if (oldCap > 0) {
 // 超过最大值就不再扩充了，就只好随你碰撞去吧
 if (oldCap >= MAXIMUM_CAPACITY) {
 threshold = Integer.MAX_VALUE;
 return oldTab;
 }
 // 没超过最大值，就扩充为原来的2倍
 else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY && oldCap >= DEFAULT_INITIAL_CAPACITY)
 newThr = oldThr << 1; // double threshold
 }
 else if (oldThr > 0) // initial capacity was placed in threshold
 newCap = oldThr;
 else {
 signifies using defaults
 newCap = DEFAULT_INITIAL_CAPACITY;
 newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
 }
 // 计算新的resize上限
 if (newThr == 0) {
 float ft = (float)newCap * loadFactor;
 newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ? (int)ft : Integer.MAX_VALUE);
 }
 threshold = newThr;
 @SuppressWarnings({"rawtypes", "unchecked"})
 Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
 table = newTab;
 if (oldTab != null) {
 // 把每个bucket都移动到新的buckets中
 for (int j = 0; j < oldCap; ++j) {
 Node<K,V> e;
 if ((e = oldTab[j]) != null) {
 oldTab[j] = null;
 if (e.next == null)
 newTab[e.hash & (newCap - 1)] = e;
 else if (e instanceof TreeNode)
 ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
 else {
 Node<K,V> loHead = null, loTail = null;
 Node<K,V> hiHead = null, hiTail = null;
 Node<K,V> next;
 do {
 next = e.next;
 // 原索引
 if ((e.hash & oldCap) == 0) {
 if (loTail == null)
 loHead = e;
 else
 loTail.next = e;
 loTail = e;
 }
 // 原索引+oldCap
 else {
 if (hiTail == null)
 hiHead = e;
 else
 hiTail.next = e;
 hiTail = e;
 }
 } while ((e = next) != null);
 // 原索引放到bucket里
 if (loTail != null) {
 loTail.next = null;
 newTab[j] = loHead;
 }
 // 原索引+oldCap放到bucket里
 if (hiTail != null) {
 hiTail.next = null;
 newTab[j + oldCap] = hiHead;
 }
 }
 }
 }
 }
}

```

```

 }
 }
}
return newTab;
}

```

## HashMap常用方法测试

```

package map;

import java.util.Collection;
import java.util.HashMap;
import java.util.Set;

public class HashMapDemo {

 public static void main(String[] args) {
 HashMap<String, String> map = new HashMap<String, String>();
 // 键不能重复，值可以重复
 map.put("san", "张三");
 map.put("si", "李四");
 map.put("wu", "王五");
 map.put("wang", "老王");
 map.put("wang", "老王2");// 老王被覆盖
 map.put("lao", "老王");
 System.out.println("-----直接输出 hashmap:-----");
 System.out.println(map);
 /**
 * 遍历HashMap
 */
 // 1. 获取Map中的所有键
 System.out.println("-----foreach获取Map中所有的键:-----");
 Set<String> keys = map.keySet();
 for (String key : keys) {
 System.out.print(key + " ");
 }
 System.out.println();// 换行
 // 2. 获取Map中所有值
 System.out.println("-----foreach获取Map中所有的值:-----");
 Collection<String> values = map.values();
 for (String value : values) {
 System.out.print(value + " ");
 }
 System.out.println();// 换行
 // 3. 得到key的值的同时得到key所对应的值
 System.out.println("-----得到key的值的同时得到key所对应的值:-----");
 Set<String> keys2 = map.keySet();
 for (String key : keys2) {
 System.out.print(key + ":" + map.get(key) + " ");
 }
 /**
 * 另外一种不常用的遍历方式
 */
 // 当我调用put(key,value)方法的时候，首先会把key和value封装到
 // Entry这个静态内部类对象中，把Entry对象再添加到数组中，所以我们想获取
 // map中的所有键值对，我们只要获取数组中的所有Entry对象，接下来
 // 调用Entry对象中的getKey()和getValue()方法就能获取键值对了
 Set<java.util.Map.Entry<String, String>> entrys = map.entrySet();
 for (java.util.Map.Entry<String, String> entry : entrys) {
 System.out.println(entry.getKey() + "--" + entry.getValue());
 }
 }
}

```

```
 /**
 * HashMap其他常用方法
 */
 System.out.println("after map.size(): "+map.size());
 System.out.println("after map.isEmpty(): "+map.isEmpty());
 System.out.println(map.remove("san"));
 System.out.println("after map.remove(): "+map);
 System.out.println("after map.get(si): "+map.get("si"));
 System.out.println("after map.containsKey(si): "+map.containsKey("si"));
 System.out.println("after containsValue(李四): "+map.containsValue("李四"));
 System.out.println(map.replace("si", "李四2"));
 System.out.println("after map.replace(si, 李四2): "+map);
}

}
```

## 多线程系列文章

下列文章，我都更新在了我的博客专栏：[Java并发编程指南](#)。

1. Java多线程学习（一）Java多线程入门
2. Java多线程学习（二）synchronized关键字（1）
3. Java多线程学习（二）synchronized关键字（2）
4. Java多线程学习（三）volatile关键字
5. Java多线程学习（四）等待/通知（wait/notify）机制
6. Java多线程学习（五）线程间通信知识点补充
7. Java多线程学习（六）Lock锁的使用
8. Java多线程学习（七）并发编程中一些问题
9. Java多线程学习（八）线程池与Executor框架

## 多线程系列文章重要知识点与思维导图

### Java多线程学习（一）Java多线程入门

## 第一章 Java多线程入门

### 四 一些常用方法

- currentThread()
- getId()
- getName()
- getPriority()
- isAlive()
- sleep(long millis)
- interrupt()
- interrupted和isInterrupted
- setName(String name)
- isDaemon()
- setDaemon(boolean on)
- join()
- yield()
- setPriority(int newPriority)

### 五 如何停止一个线程呢

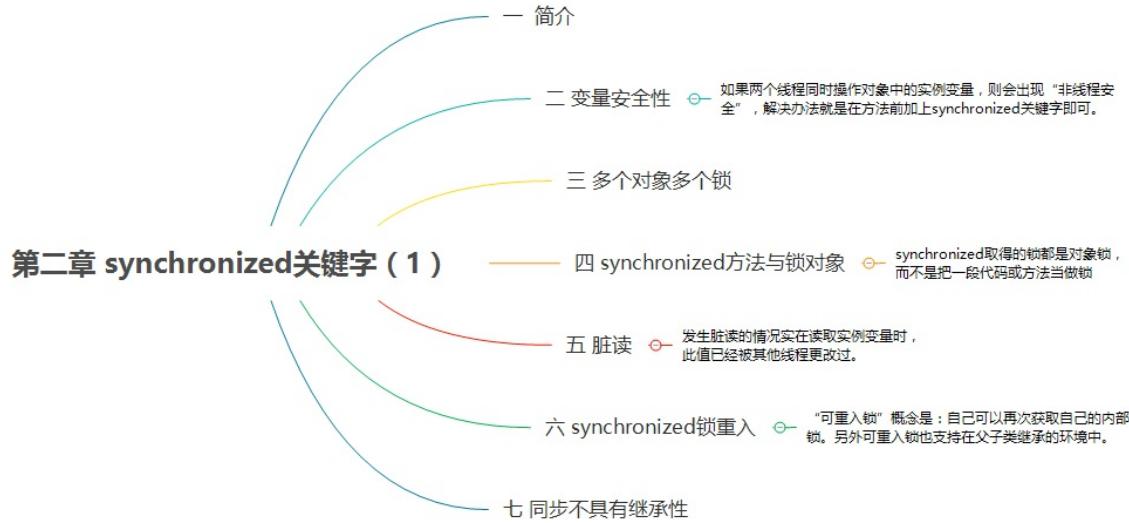
- 5.1 使用interrupt方法
- 5.2 使用return停止线程

### 六 线程的优先级

### 七 Java多线程分类

- 7.1 多线程分类
- 7.2 如何设置守护

## Java多线程学习（二）synchronized关键字（1）



注意：可重入锁的概念。

另外要注意：**synchronized**取得的锁都是对象锁，而不是把一段代码或方法当做锁。如果多个线程访问的是同一个对象，哪个线程先执行带synchronized关键字的方法，则哪个线程就持有该方法，那么其他线程只能呈等待状态。如果多个线程访问的是多个对象则不一定，因为多个对象会产生多个锁。

## Java多线程学习（二）synchronized关键字（2）



注意：

- 其他线程执行对象中**synchronized**同步方法（上一节我们介绍过，需要回顾的可以看上一节的文章）和**synchronized(this)**代码块时呈现同步效果；
- 如果两个线程使用了同一个“对象监视器”(**synchronized(object)**)，运行结果同步，否则不同步。

**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，而**synchronized**关键字加到非**static**静态方法上是给对象上锁。

数据类型String的常量池属性:在JVM中具有String常量池缓存的功能

## Java多线程学习（三）volatile关键字

### 第三章 volatile关键字

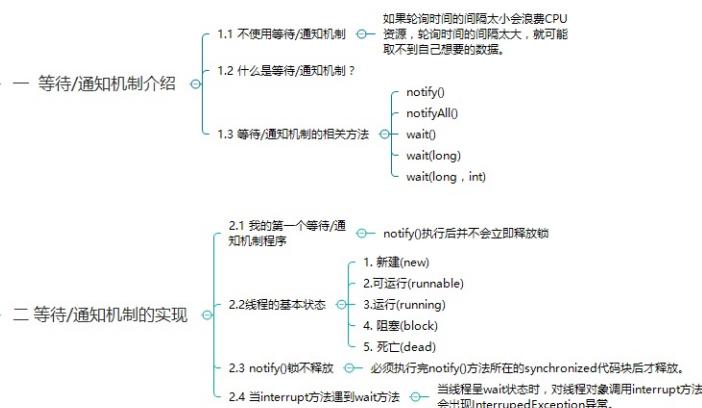


注意：

**synchronized关键字和volatile关键字比较**

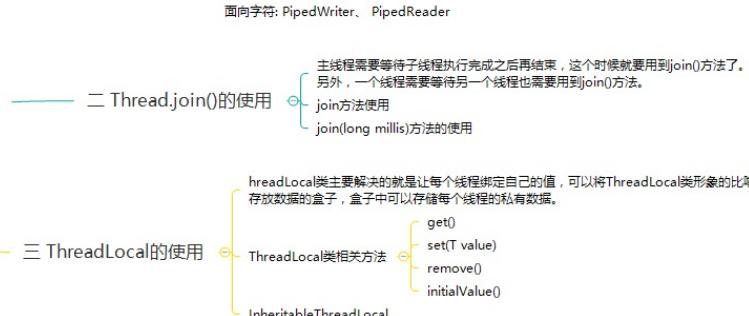
### Java多线程学习（四）等待/通知（wait/notify）机制

#### 第四章 等待/通知（wait/notify）机制



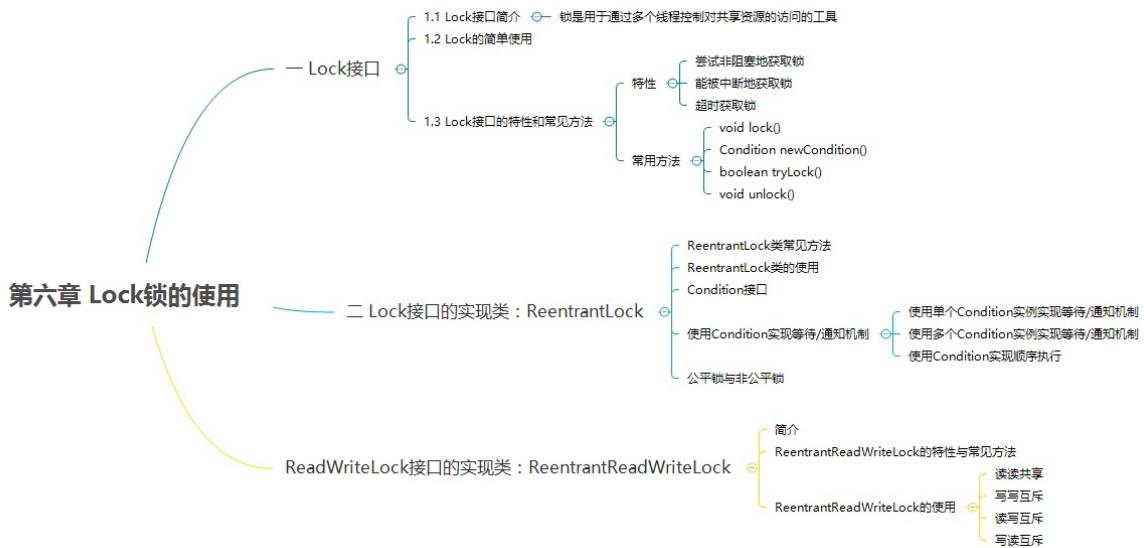
### Java多线程学习（五）线程间通信知识点补充

#### 第五章 线程间通信知识点补充



注意： 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。

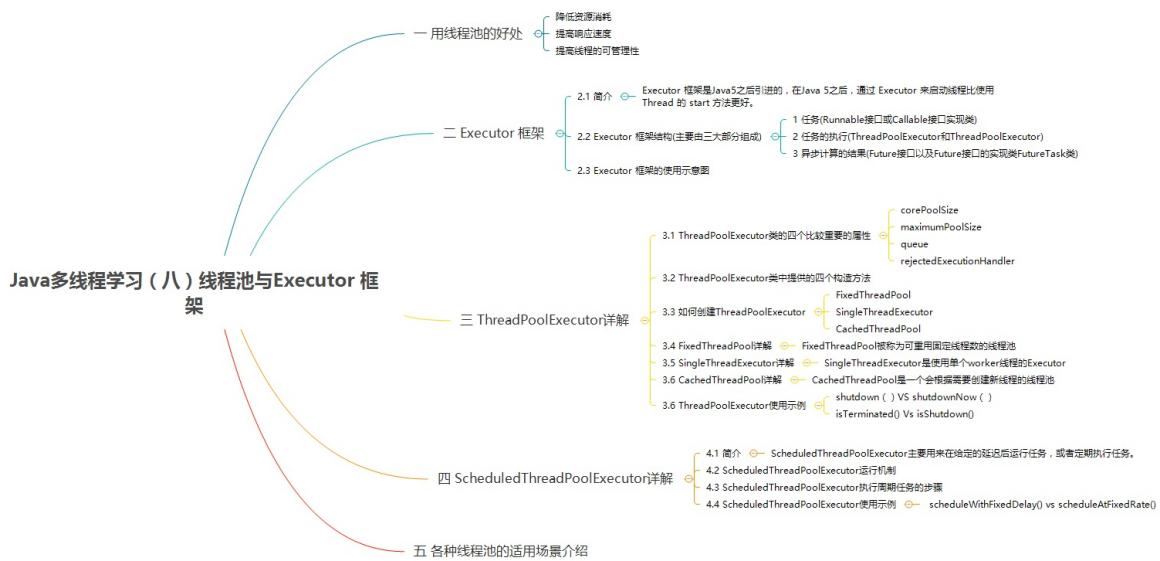
### Java多线程学习（六）Lock锁的使用



## Java多线程学习（七）并发编程中一些问题



## Java多线程学习（八）线程池与Executor 框架



本文是对 synchronized 关键字使用、底层原理、JDK1.6之后的底层优化以及和ReenTrantLock对比做的总结。如果没有学过 synchronized 关键字使用的话，阅读起来可能比较费力。两篇比较基础的讲解 synchronized 关键字的文章：

- [《Java多线程学习（二）synchronized关键字（1）》](#)
- [《Java多线程学习（二）synchronized关键字（2）》](#)

## synchronized 关键字的总结

### synchronized关键字最主要的三种使用方式的总结

- 修饰实例方法，作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁
- 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁。也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（static 表明这是该类的一个静态资源，不管new了多少个对象，只有一份，所以对该类的所有对象都加了锁）。所以如果一个线程A调用一个实例对象的非静态synchronized 方法，而线程B需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。
- 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。和 synchronized 方法一样，synchronized(this)代码块也是锁定当前对象的。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。这里再提一下：synchronized关键字加到非 static 静态方法上是给对象实例上锁。另外需要注意的是：尽量不要使用 synchronized(String a) 因为JVM中，字符串常量池具有缓冲功能！

### synchronized 关键字底层实现原理总结

- 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。当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor 对象存在于每个 Java 对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么 Java 中任意对象可以作为锁的原因) 的所有权。当计数器为 0 时可以成功获取，获取后将锁计数器设为 1 也就是加 1。相应的在执行 monitorexit 指令后，将锁计数器设为 0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。
- 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。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，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对 synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6 对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

所有用户程序都是运行在用户态的，但是有时候程序确实需要做一些内核态的事情，例如从硬盘读取数据，或者从键盘获取输入等。而唯一可以做这些事情的就是操作系统。

### synchronized关键字底层优化总结

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。

锁主要存在四中状态，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。

## 偏向锁

引入偏向锁的目的和引入轻量级锁的目的很像，他们都是为了没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。但是不同是：轻量级锁在无竞争的情况下使用 CAS 操作去代替使用互斥量。而偏向锁在无竞争的情况下会把整个同步都消除掉。

偏向锁的“偏”就是偏心的偏，它的意思是会偏向于第一个获得它的线程，如果在接下来的执行中，该锁没有被其他线程获取，那么持有偏向锁的线程就不需要进行同步！关于偏向锁的原理可以查看《深入理解Java虚拟机：JVM高级特性与最佳实践》第二版的13章第三节锁优化。

但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。

## 轻量级锁

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(1.6之后加入的)。轻量级锁不是为了代替重量级锁，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗，因为使用轻量级锁时，不需要申请互斥量。另外，轻量级锁的加锁和解锁都用到了CAS操作。关于轻量级锁的加锁和解锁的原理可以查看《深入理解Java虚拟机：JVM高级特性与最佳实践》第二版的13章第三节锁优化。

轻量级锁能够提升程序同步性能的依据是“对于绝大部分锁，在整个同步周期内都是不存在竞争的”，这是一个经验数据。如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥操作的开销。但如果存在锁竞争，除了互斥量开销外，还会额外发生CAS操作，因此在有锁竞争的情况下，轻量级锁比传统的重量级锁更慢！如果锁竞争激烈，那么轻量级将很快膨胀为重量级锁！

## 自旋锁和自适应自旋

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。

互斥同步对性能最大的影响就是阻塞的实现，因为挂起线程/恢复线程的操作都需要转入内核态中完成（用户态转换到内核态会耗费时间）。

一般线程持有锁的时间都不是太长，所以仅仅为了这一点时间去挂起线程/恢复线程是得不偿失的。所以，虚拟机的开发团队就这样去考虑：“我们能不能让后面来的请求获取锁的线程等待一会而不被挂起呢？看看持有锁的线程是否很快就会释放锁”。为了让一个线程等待，我们只需要让线程执行一个忙循环（自旋），这项技术就叫做自旋。

百度百科对自旋锁的解释：

何谓自旋锁？它是为实现保护共享资源而提出一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时候最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，“自旋”一词就是因此而得名。

自旋锁在 JDK1.6 之前其实就已经引入了，不过是默认关闭的，需要通过 `--xx:+UseSpinning` 参数来开启。JDK1.6 及 1.6 之后，就改为默认开启的了。需要注意的是：自旋等待不能完全替代阻塞，因为它还是要占用处理器时间。如果锁被占用的时间短，那么效果当然就很好了！反之，相反！自旋等待的时间必须要有限度。如果自旋超过了限定次数仍然没有获得锁，就应该挂起线程。自旋次数的默认值是10次，用户可以修改 `--XX:PreBlockSpin` 来更改。

另外，在 JDK1.6 中引入了自适应的自旋锁。自适应的自旋锁带来的改进就是：自旋的时间不在固定了，而是和前一次同一个锁上的自旋时间以及锁的拥有者的状态来决定，虚拟机变得越来越“聪明”了。

## 锁消除

锁消除理解起来很简单，它指的就是虚拟机即使编译器在运行时，如果检测到那些共享数据不可能存在竞争，那么就执行锁消除。锁消除可以节省毫无意义的请求锁的时间。

## 锁粗化

原则上，我们再编写代码的时候，总是推荐将同步快的作用范围限制得尽量小——只在共享数据的实际作用域才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待线程也能尽快拿到锁。

大部分情况下，上面的原则都是没有问题的，但是如果一系列的连续操作都对同一个对象反复加锁和解锁，那么会带来很多不必要的性能消耗。

# ReenTrantLock 和 synchronized 关键字的总结

推荐一篇讲解 ReenTrantLock 的使用比较基础的文章： [《Java多线程学习（六）Lock锁的使用》](#)

## 两者都是可重入锁

两者都是可重入锁。“可重入锁”概念是：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，如果不可锁重入的话，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），所以我们可以通过查看它的源代码，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：①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）

- 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。
- 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。
- 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

在JDK1.6之前，`synchronized` 的性能是比 `ReenTrantLock` 差很多。具体表示为：`synchronized` 关键字吞吐量岁线程数的增加，下降得非常严重。而`ReenTrantLock` 基本保持一个比较稳定的水平。我觉得这也侧面反映了，`synchronized` 关键字还有非常大的优化余地。后续的技术发展也证明了这一点，我们上面也讲了在 JDK1.6 之后 JVM 团队对 `synchronized` 关键字做了很多优化。JDK1.6 之后，`synchronized` 和 `ReenTrantLock` 的性能基本是持平了。所以网上那些说因为性能才选择 `ReenTrantLock` 的文章都是错的！JDK1.6之后，性能已经不是选择`synchronized`和`ReenTrantLock`的影响因素了！而且虚拟机在未来的性能改进中会更偏向于原生的`synchronized`，所以还是提倡在`synchronized`能满足你的需求的情况下，优先考虑使用`synchronized`关键字来进行同步！优化后的`synchronized`和`ReenTrantLock`一样，在很多地方都是用到了CAS操作。

## 参考

- 《深入理解Java虚拟机：JVM高级特性与最佳实践》第二版第13章
- 《实战Java虚拟机》
- <https://blog.csdn.net/javazejian/article/details/72828483#commentBox>
- <https://blog.csdn.net/qq838642798/article/details/65441415>
- <http://cmsblogs.com/?p=2071>

## 写在前面（常见面试题）

### 基本问题：

- 介绍下 Java 内存区域（运行时数据区）
- Java 对象的创建过程（五步，建议能默写出来并且要知道每一步虚拟机做了什么）
- 对象的访问定位的两种方式（句柄和直接指针两种方式）

### 拓展问题：

- String类和常量池
- 8种基本类型的包装类和常量池

## 1 概述

对于 Java 程序员来说，在虚拟机自动内存管理机制下，不再需要像C/C++程序开发程序员这样为每一个 new 操作去写对应的 delete/free 操作，不容易出现内存泄漏和内存溢出问题。正是因为 Java 程序员把内存控制权利交给 Java 虚拟机，一旦出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那么排查错误将会是一个非常艰巨的任务。

## 2 运行时数据区域

Java 虚拟机在执行 Java 程序的过程中会把它管理的内存划分成若干个不同的数据区域。

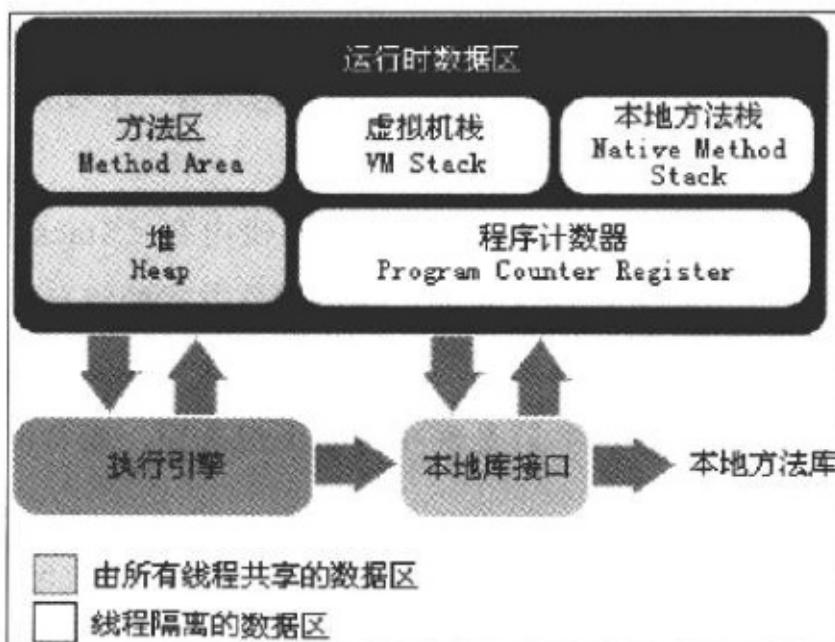


图 2-1 Java 虚拟机运行时数据区

线程私有的，其他的则是线程共享的。

线程私有的：

这些组成部分一些事

- 程序计数器
- 虚拟机栈
- 本地方法栈

线程共享的：

- 堆
- 方法区
- 直接内存

## 2.1 程序计数器

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

另外，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

从上面的介绍中我们知道程序计数器主要有两个作用：

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

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

## 2.2 Java 虚拟机栈

与程序计数器一样，Java虚拟机栈也是线程私有的，它的生命周期和线程相同，描述的是Java方法执行的内存模型。

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

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

Java虚拟机栈会出现两种异常：`StackOverflowError` 和 `OutOfMemoryError`。

- **StackOverflowError**：若Java虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前Java虚拟机栈的最大深度的时候，就抛出`StackOverflowError`异常。
- **OutOfMemoryError**：若Java虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了，无法再动态扩展了，此时抛出`OutOfMemoryError`异常。

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

## 2.3 本地方法栈

和虚拟机栈所发挥的作用非常相似，区别是：虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的Native方法服务。在HotSpot虚拟机中和Java虚拟机栈合二为一。

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

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

## 2.4 堆

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

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



在 JDK 1.8 中移除整个永久代，取而代之的是一个叫元空间（Metaspace）的区域（永久代使用的是JVM的堆内存空间，而元空间使用的是物理内存，直接受到本机的物理内存限制）。

推荐阅读：

- 《Java8内存模型—永久代(PermGen)和元空间(Metaspace)》：<http://www.cnblogs.com/paddix/p/5309550.html>

## 2.5 方法区

方法区与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 **Non-Heap**（非堆），目的应该是与 Java 堆区分开来。

HotSpot 虚拟机中方法区也常被称为“**永久代**”，本质上两者并不等价。仅仅是因为 HotSpot 虚拟机设计团队用永久代来实现方法区而已，这样 HotSpot 虚拟机的垃圾收集器就可以像管理 Java 堆一样管理这部分内存了。但是这并不是一个好主意，因为这样更容易遇到内存溢出问题。

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

## 2.6 运行时常量池

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

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

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



——图片来源：<https://blog.csdn.net/wangbiao007/article/details/78545189>

推荐阅读：

- 《Java 中几种常量池的区分》：[https://blog.csdn.net/qq\\_26222859/article/details/73135660](https://blog.csdn.net/qq_26222859/article/details/73135660)

## 2.7 直接内存

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

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

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

## 3 HotSpot 虚拟机对象探秘

通过上面的介绍我们大概知道了虚拟机的内存情况，下面我们来详细的了解一下 HotSpot 虚拟机在 Java 堆中对象分配、布局和访问的全过程。

### 3.1 对象的创建

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

## Java 创建对象的过程



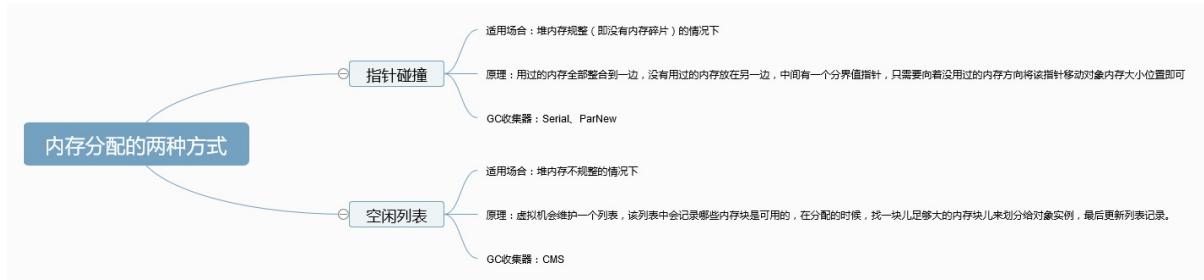
微信公众号：Java面试通关手册

**①类加载检查：** 虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

**②分配内存：** 在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。分配方式有“指针碰撞”和“空闲列表”两种，选择那种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

内存分配的两种方式：（补充内容，需要掌握）

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



内存分配并发问题（补充内容，需要掌握）

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

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

**③初始化零值：** 内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

**④设置对象头：** 初始化零值完成之后，虚拟机要对对象进行必要的设置，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。这些信息存放在对象头中。另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

**⑤执行 init 方法：** 在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚刚开始，`<init>` 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令之后会接着执行 `<init>` 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

## 3.2 对象的内存布局

在 Hotspot 虚拟机中，对象在内存中的布局可以分为3快区域：对象头、实例数据和对齐填充。

Hotspot虚拟机的对象头包括两部分信息，第一部分用于存储对象自身的自身运行时数据（哈希码、GC分代年龄、锁状态标志等等），另一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是那个类的实例。

实例数据部分是对象真正存储的有效信息，也是在程序中所定义的各种类型的字段内容。

对齐填充部分不是必然存在的，也没有什么特别的含义，仅仅起占位作用。因为Hotspot虚拟机的自动内存管理系统要求对象起始地址必须是8字节的整数倍，换句话说就是对象的大小必须是8字节的整数倍。而对象头部分正好是8字节的倍数（1倍或2倍），因此，当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

### 3.3 对象的访问定位

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

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

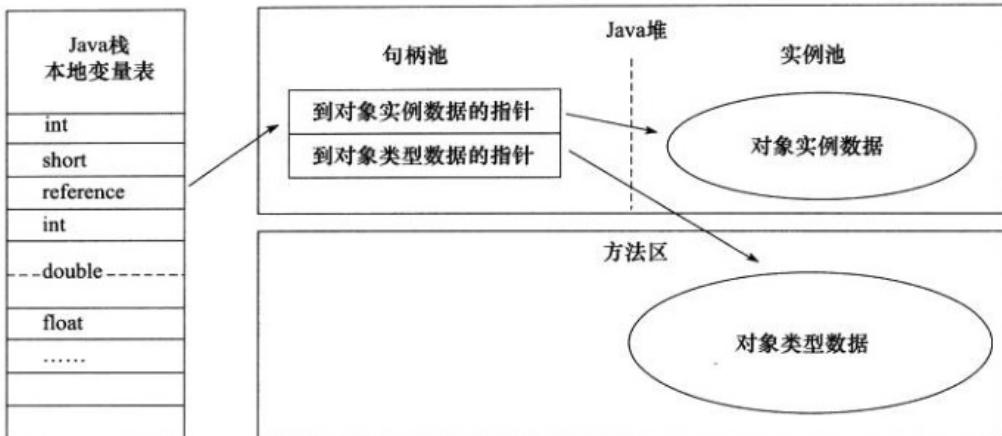


图 2-2 通过句柄访问对象

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

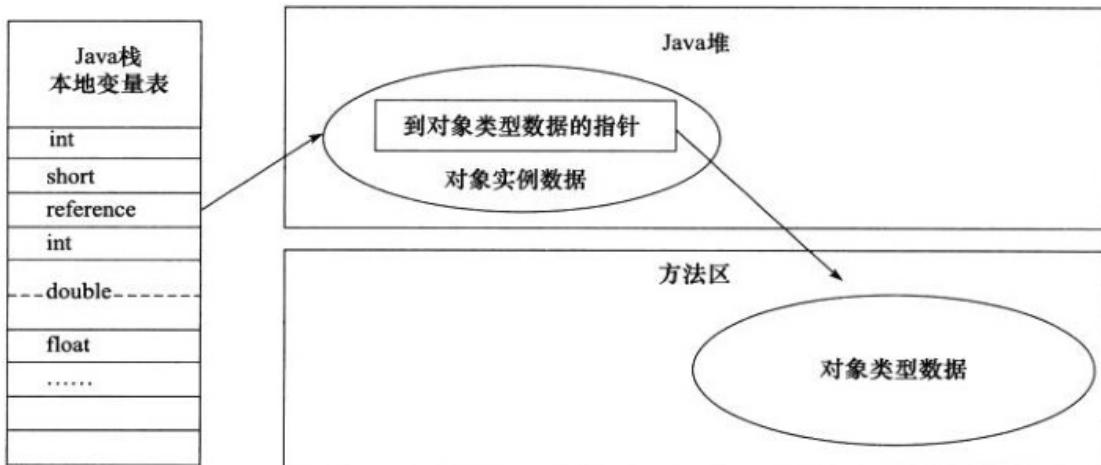


图 2-3 通过直接指针访问对象

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

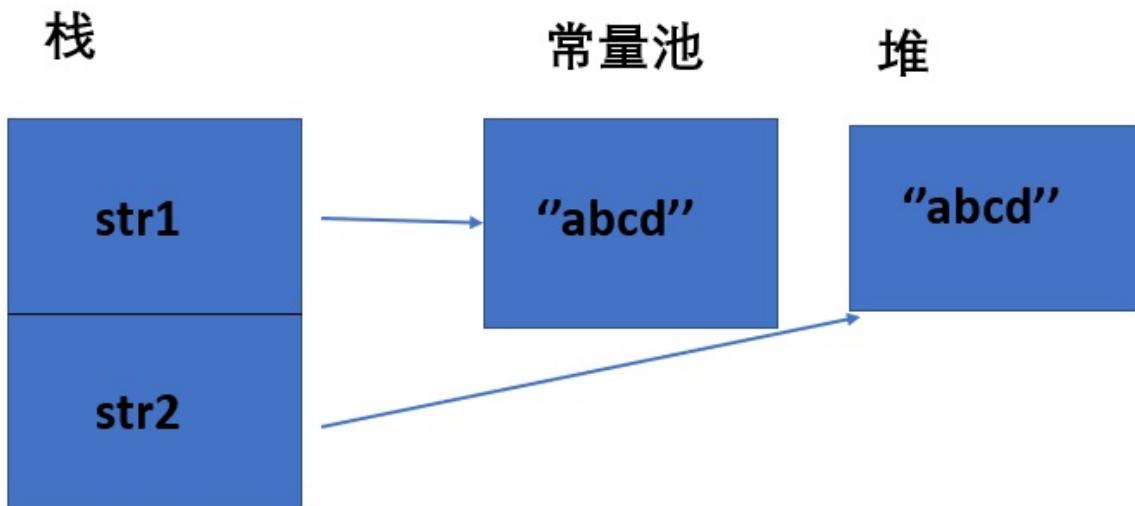
## 四 重点补充内容

### String 类和常量池

1 String 对象的两种创建方式：

```
String str1 = "abcd";
String str2 = new String("abcd");
System.out.println(str1==str2); //false
```

这两种不同的创建方法是有差别的，第一种方式是在常量池中拿对象，第二种方式是直接在堆内存空间创建一个新的对象。



记住：只要使用new方法，便需要创建新的对象。

2 String 类型的常量池比较特殊。它的主要使用方法有两种：

- 直接使用双引号声明出来的 String 对象会直接存储在常量池中。
- 如果不是用双引号声明的 String 对象，可以使用 String 提供的 `intern` 方法。`String.intern()` 是一个 Native 方法，它的作用是：如果运行时常量池中已经包含一个等于此 String 对象内容的字符串，则返回常量池中该字符串的引用；如果没有，则在常量池中创建与此 String 内容相同的字符串，并返回常量池中创建的字符串的引用。

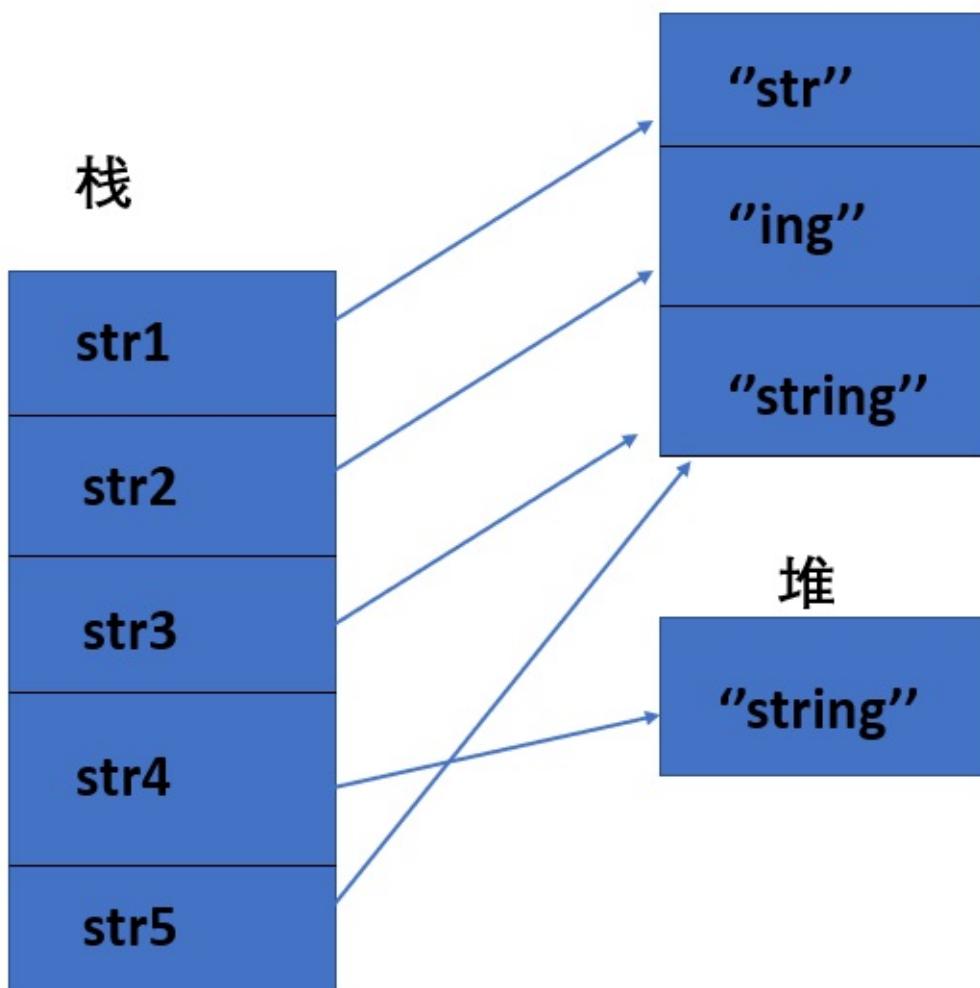
```
String s1 = new String("计算机");
String s2 = s1.intern();
String s3 = "计算机";
System.out.println(s1==s2); //false, 因为一个是堆内存中的String对象一个是常量池中的String对象,
System.out.println(s3==s2); //true, 因为两个都是常量池中的String对
```

3 String 字符串拼接

```
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
```

## 常量池



尽量避免多个字符串拼接，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。

**String s1 = new String("abc");这句话创建了几个对象？**

创建了两个对象。

验证：

```
String s1 = new String("abc");// 堆内存的地址值
String s2 = "abc";
System.out.println(s1 == s2); // 输出false, 因为一个是堆内存，一个是常量池的内存，故两者是不同的。
System.out.println(s1.equals(s2)); // 输出true
```

结果：

```
false
true
```

解释：

先有字符串"abc"放入常量池，然后 new 了一份字符串"abc"放入Java堆(字符串常量"abc"在编译期就已经确定放入常量池，而 Java 堆上的"abc"是在运行期初始化阶段才确定)，然后 Java 栈的 str1 指向Java堆上的"abc"。

## 8种基本类型的包装类和常量池

- Java 基本类型的包装类的大部分都实现了常量池技术，即Byte,Short,Integer,Long,Character,Boolean；这5种包装类默认创建了数值[-128, 127]的相应类型的缓存数据，但是超出此范围仍然会去创建新的对象。
- 两种浮点数类型的包装类 Float,Double 并没有实现常量池技术。

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

Integer 缓存源代码：

```
/**
 * 此方法将始终缓存-128到127（包括端点）范围内的值，并可以缓存此范围之外的其他值。
 */
public static Integer valueOf(int i) {
 if (i >= IntegerCache.low && i <= IntegerCache.high)
 return IntegerCache.cache[i + (-IntegerCache.low)];
 return new Integer(i);
}
```

应用场景：

1. Integer i1=40; Java 在编译的时候会直接将代码封装成Integer i1=Integer.valueOf(40);，从而使用常量池中的对象。
2. Integer i1 = new Integer(40);这种情况下会创建新的对象。

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

Integer比较更丰富的一个例子：

```
Integer i1 = 40;
Integer i2 = 40;
Integer i3 = 0;
Integer i4 = new Integer(40);
Integer i5 = new Integer(40);
Integer i6 = new Integer(0);

System.out.println("i1=i2 " + (i1 == i2));
System.out.println("i1=i2+i3 " + (i1 == i2 + i3));
System.out.println("i1=i4 " + (i1 == i4));
```

```
System.out.println("i4=i5 " + (i4 == i5));
System.out.println("i4=i5+i6 " + (i4 == i5 + i6));
System.out.println("40=i5+i6 " + (40 == i5 + i6));
```

结果：

```
i1=i2 true
i1=i2+i3 true
i1=i4 false
i4=i5 false
i4=i5+i6 true
40=i5+i6 true
```

解释：

语句*i4 == i5 + i6*，因为+这个操作符不适用于Integer对象，首先i5和i6进行自动拆箱操作，进行数值相加，即*i4 == 40*。然后Integer对象无法与数值进行直接比较，所以i4自动拆箱转为int值40，最终这条语句转为*40 == 40*进行数值比较。

参考：

- 《深入理解Java虚拟机：JVM高级特性与最佳实践（第二版）》
- 《实战java虚拟机》
- <https://www.cnblogs.com/CZDblog/p/5589379.html>
- <https://www.cnblogs.com/java-zhao/p/5180492.html>
- [https://blog.csdn.net/qq\\_26222859/article/details/73135660](https://blog.csdn.net/qq_26222859/article/details/73135660)
- <https://blog.csdn.net/cugwuhan2014/article/details/78038254>

上文回顾： [《可能是把Java内存区域讲的最清楚的一篇文章》](#)

## 写在前面

### 本节常见面试题：

问题答案在文中都有提到

- 如何判断对象是否死亡（两种方法）。
- 简单的介绍一下强引用、软引用、弱引用、虚引用（虚引用与软引用和弱引用的区别、使用软引用能带来的好处）。
- 如何判断一个常量是废弃常量
- 如何判断一个类是无用的类
- 垃圾收集有哪些算法，各自的特点？
- HotSpot为什么要分为新生代和老年代？
- 常见的垃圾回收器有那些？
- 介绍一下CMS,G1收集器。
- Minor Gc和Full GC有什么不同呢？

### 本文导火索



当需要排查各种 内存溢出问题、当垃圾收集称为系统达到更高并发的瓶颈时，我们就需要对这些“自动化”的技术实施必要的监控和调节。

## 1 揭开JVM内存分配与回收的神秘面纱

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

**JDK1.8之前的堆内存示意图：**



从上图可以看出堆内存的分为新生代、老年代和永久代。新生代又被进一步分为：Eden 区 + Survivor1 区 + Survivor2 区。值得注意的是，在 JDK 1.8 中移除整个永久代，取而代之的是一个叫元空间（Metaspace）的区域（永久代使用的是JVM的堆内存空间，而元空间使用的是物理内存，直接受到本机的物理内存限制）。



公众号：Java面试通关手册

## 1.1 对象优先在eden区分配

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

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

在测试之前我们先来看看 **Minor Gc** 和 **Full GC** 有什么不同呢？

- **新生代GC (Minor GC)** :指发生新生代的垃圾收集动作，Minor GC非常频繁，回收速度一般也比较快。
- **老年代GC (Major GC/Full GC)** :指发生在老年代的GC，出现了Major GC经常会伴随至少一次的Minor GC（并非绝对），Major GC的速度一般会比Minor GC的慢10倍以上。

测试：

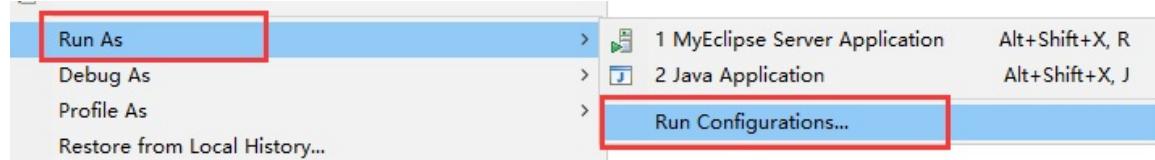
```
public class GCTest {
```

```

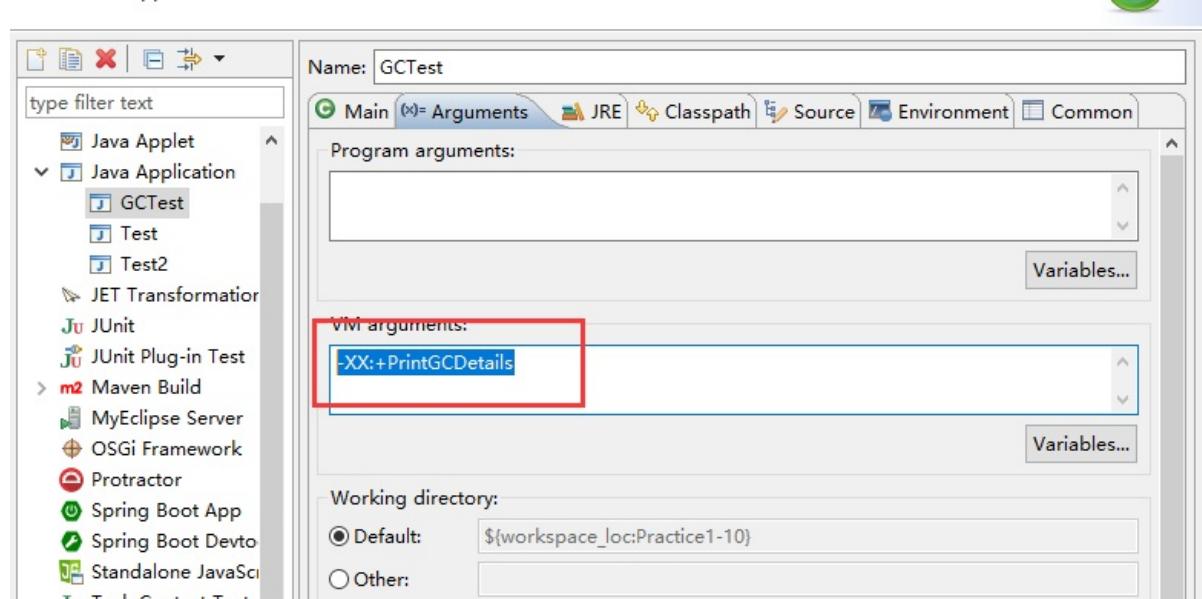
public static void main(String[] args) {
 byte[] allocation1, allocation2;
 allocation1 = new byte[30900*1024];
 //allocation2 = new byte[900*1024];
}
}

```

通过以下方式运行：



添加的参数： -XX:+PrintGCDetails



运行结果：

```

Heap 新生代
PSYoungGen total 38400K, used 33280K [0x00000000d5d00000, 0x00000000d8780000, 0x0000000100000000)
eden space 33280K, 100% used [0x00000000d5d00000, 0x00000000d7d80000, 0x00000000d7d80000)
from space 5120K, 0% used [0x00000000d8280000, 0x00000000d8280000, 0x00000000d8780000)
to space 5120K, 0% used [0x00000000d7d80000, 0x00000000d7d80000, 0x00000000d8280000)
ParOldGen 老年代
total 87552K, used 0K [0x0000000081600000, 0x0000000086b80000, 0x00000000d5d00000)
object space 87552K, 0% used [0x0000000081600000, 0x0000000081600000, 0x0000000086b80000)
Metaspace
used 2621K, capacity 4486K, committed 4864K, reserved 1056768K
class space used 283K, capacity 386K, committed 512K, reserved 1048576K
元空间对应于JDK1.8的永久代

```

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

```

allocation2 = new byte[900*1024];

```

```

[GC (Allocation Failure) [PSYoungGen: 32897K->768K(38400K)] 32897K->31676K(125952K), 0.0229658 secs] [Times: us
Heap
PSYoungGen total 38400K, used 2001K [0x00000000d5d00000, 0x00000000da800000, 0x0000000100000000)
eden space 33280K, 3% used [0x00000000d5d00000, 0x00000000d5e344b8, 0x00000000d7d80000)
from space 5120K, 15% used [0x00000000d7d80000, 0x00000000d7e40030, 0x00000000d8280000)
to space 5120K, 0% used [0x00000000da300000, 0x00000000da300000, 0x00000000da800000)
ParOldGen total 87552K, used 30908K [0x0000000081600000, 0x0000000086b80000, 0x00000000d5d00000)
object space 87552K, 35% used [0x0000000081600000, 0x000000008342f010, 0x0000000086b80000)
Metaspace
used 2621K, capacity 4486K, committed 4864K, reserved 1056768K
class space used 283K, capacity 386K, committed 512K, reserved 1048576K

```

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

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

## 1.2 大对象直接进入老年代

大对象就是需要大量连续内存空间的对象（比如：字符串、数组）。

为什么要这样呢？

为了避免为大对象分配内存时由于分配担保机制带来的复制而降低效率。

## 1.3 长期存活的对象将进入老年代

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

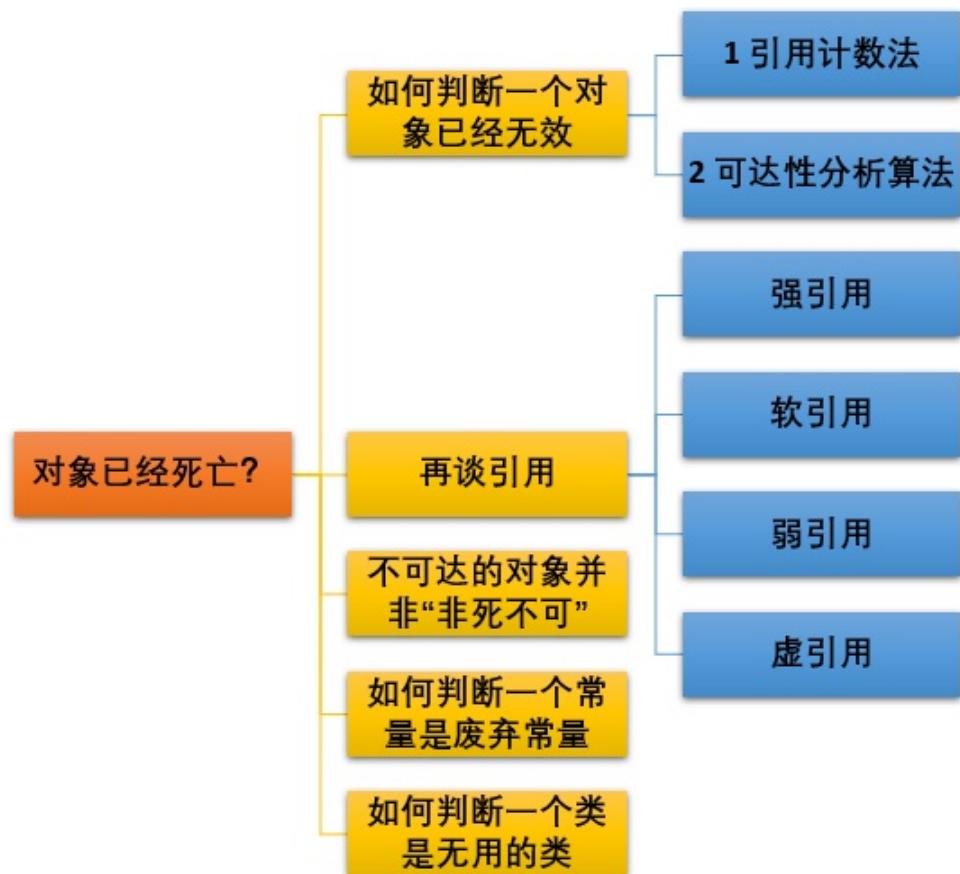
如果对象在 Eden 出生并经过第一次 Minor GC 后仍然能够存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为1.对象在 Survivor 中每熬过一次 MinorGC,年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

## 1.4 动态对象年龄判定

为了更好的适应不同程序的内存情况，虚拟机不是永远要求对象年龄必须达到了某个值才能进入老年代，如果 Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无需达到要求的年龄。

## 2 对象已经死亡？

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



## 2.1 引用计数法

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

这个方法实现简单，效率高，但是目前主流的虚拟机中并没有选择这个算法来管理内存，其最主要的原因是它很难解决对象之间相互循环引用的问题。所谓对象之间的相互引用问题，如下面代码所示：除了对象objA 和 objB 相互引用着对方之外，这两个对象之间再无任何引用。但是他们因为互相引用对方，导致它们的引用计数器都不为0，于是引用计数算法无法通知 GC 回收器回收他们。

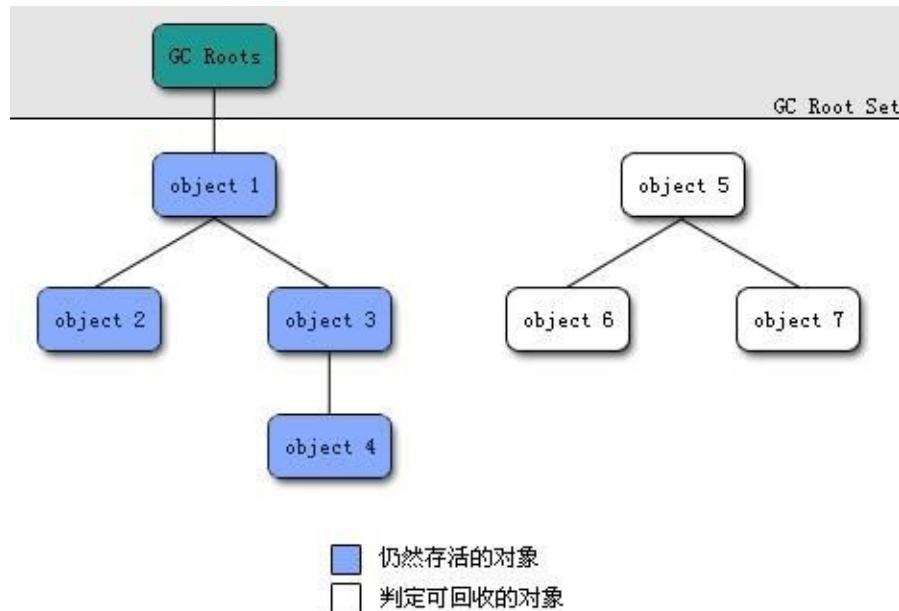
```

public class ReferenceCountingGc {
 Object instance = null;
 public static void main(String[] args) {
 ReferenceCountingGc objA = new ReferenceCountingGc();
 ReferenceCountingGc objB = new ReferenceCountingGc();
 objA.instance = objB;
 objB.instance = objA;
 objA = null;
 objB = null;
 }
}

```

## 2.2 可达性分析算法

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



## 2.3 再谈引用

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

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

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

### 1. 强引用

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

### 2. 软引用 (SoftReference)

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

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

### 3. 弱引用 (WeakReference)

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

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

### 4. 虚引用 (PhantomReference)

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

虚引用主要用来跟踪对象被垃圾回收的活动。

虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

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

## 2.4 不可达的对象并非“非死不可”

即使在可达性分析法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑阶段”，要真正宣告一个对象死亡，至少要经历两次标记过程；可达性分析法中不可达的对象被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize 方法。当对象没有覆盖 finalize 方法，或 finalize 方法已经被虚拟机调用过时，虚拟机将这两种情况视为没有必要执行。

被判定为需要执行的对象将会被放在一个队列中进行第二次标记，除非这个对象与引用链上的任何一个对象建立关联，否则就会被真的回收。

## 2.5 如何判断一个常量是废弃常量

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

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

注意：我们在[可能是把Java内存区域讲的最清楚的一篇文章](#)也讲了JDK1.7及之后版本的JVM已经将运行时常量池从方法区中移了出来，在Java堆（Heap）中开辟了一块区域存放运行时常量池。

## 2.6 如何判断一个类是无用的类

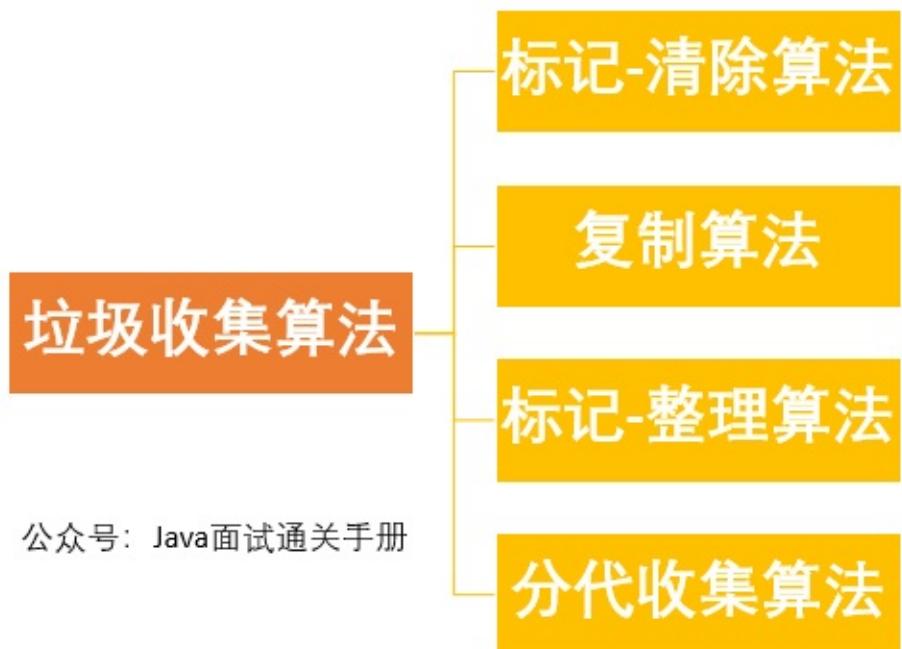
方法区主要回收的是无用的类，那么如何判断一个类是无用的类的呢？

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

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

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

# 3 垃圾收集算法

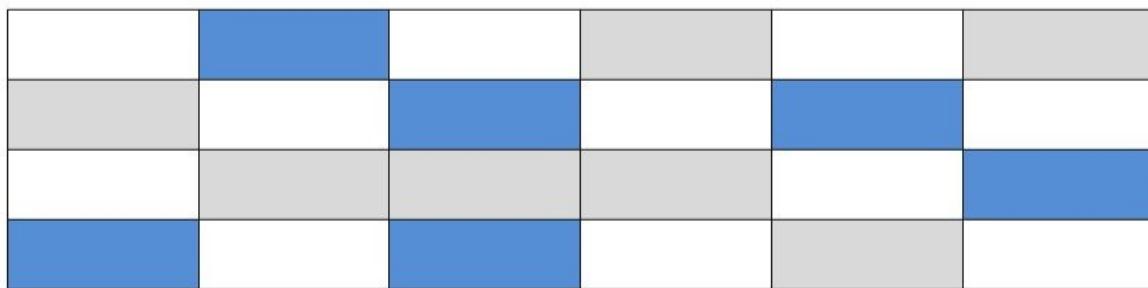


### 3.1 标记-清除算法

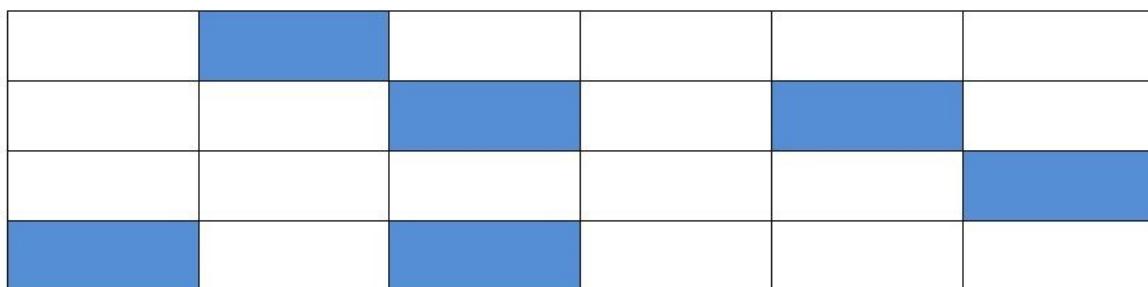
算法分为“标记”和“清除”阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。它是最基础的收集算法，效率也很高，但是会带来两个明显的问题：

1. 效率问题
2. 空间问题（标记清除后会产生大量不连续的碎片）

## 内存整理前



## 内存整理后



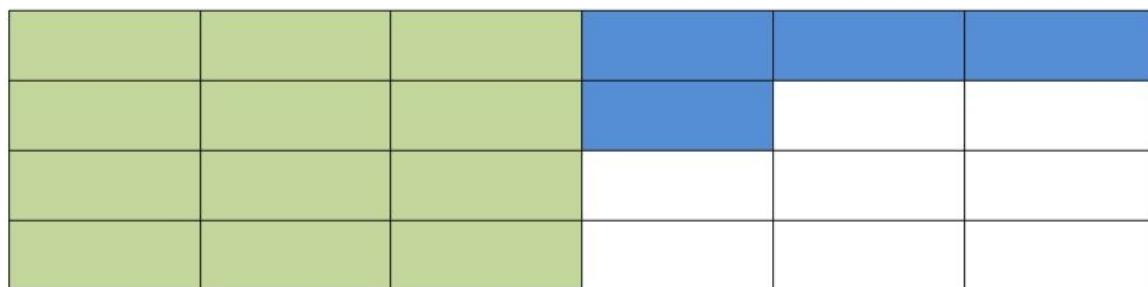
### 3.2 复制算法

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

## 内存整理前



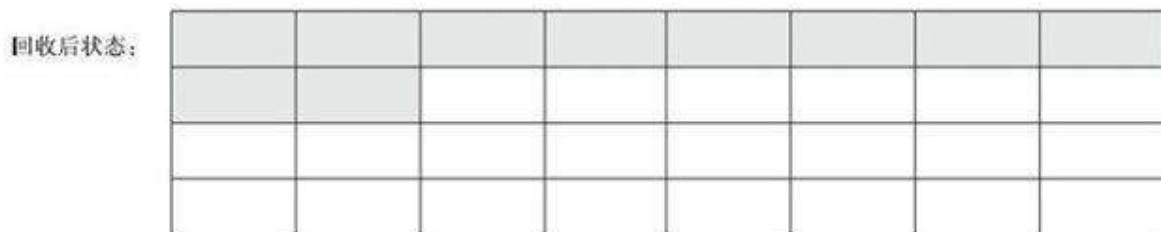
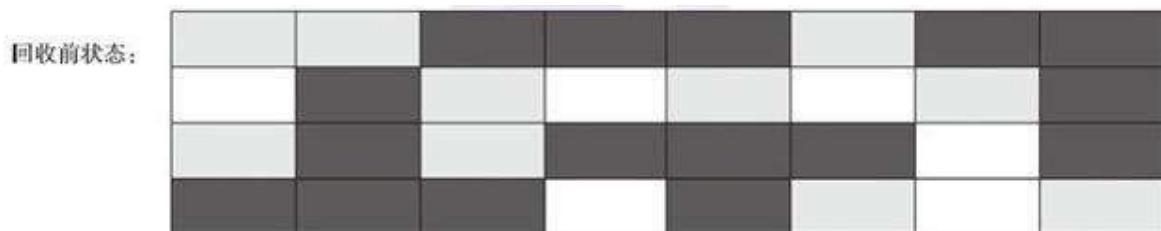
## 内存整理后



| 可用内存 | 可回收内存 | 存活对象 | 保留内存 |
|------|-------|------|------|
|------|-------|------|------|

### 3.3 标记-整理算法

根据老年代的特点特出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一段移动，然后直接清理掉端边界以外的内存。



存活对象

可回收

未使用

## 3.4 分代收集算法

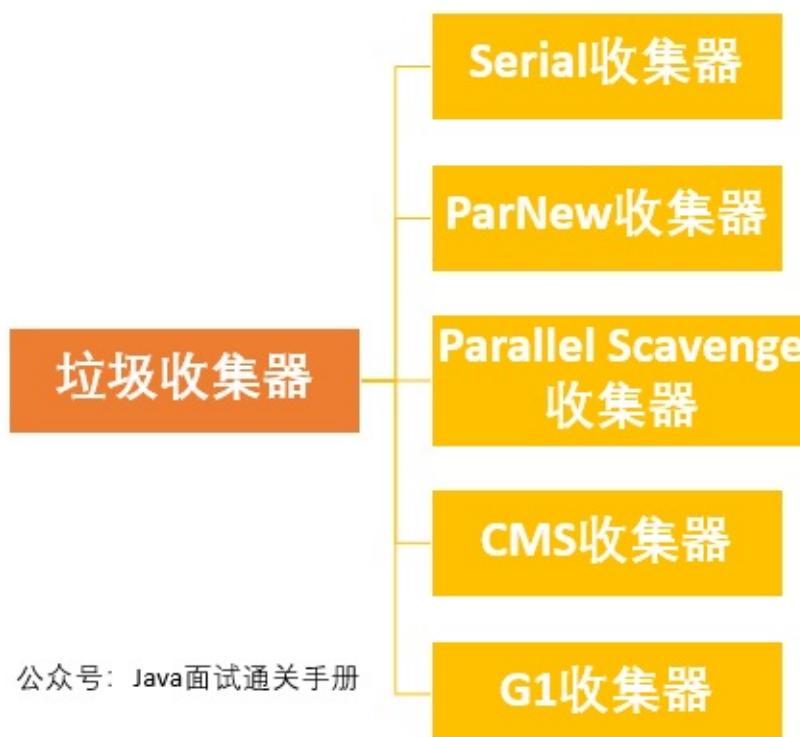
当前虚拟机的垃圾回收都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将Java堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

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

延伸面试问题：HotSpot为什么要分为新生代和老年代？

根据上面的对分代收集算法的介绍回答。

## 4 垃圾收集器



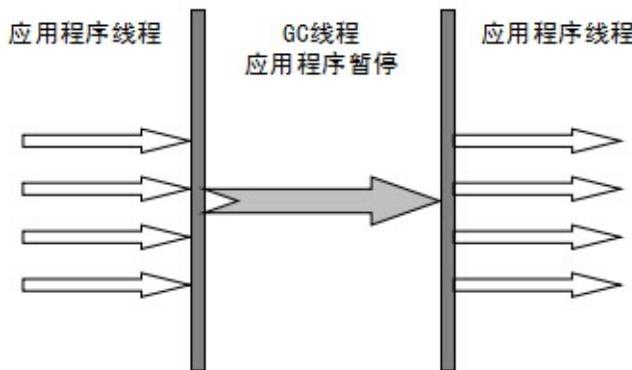
如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

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

### 4.1 Serial收集器

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

新生代采用复制算法，老年代采用标记-整理算法。



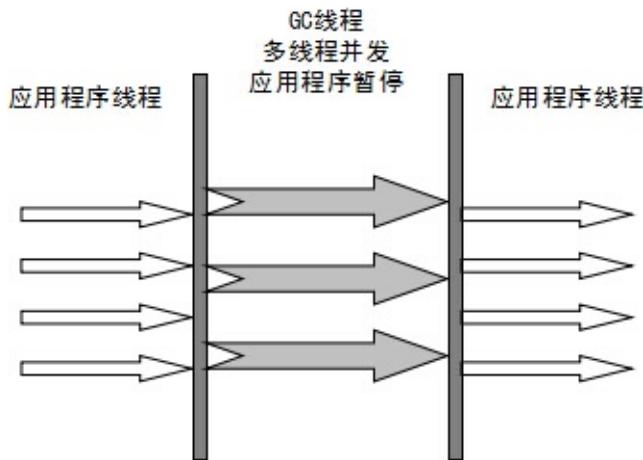
虚拟机的设计者们当然知道Stop The World带来的不良用户体验，所以在后续的垃圾收集器设计中停顿时间在不断缩短（仍然还有停顿，寻找最优秀的垃圾收集器的过程仍然在继续）。

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

## 4.2 ParNew收集器

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

新生代采用复制算法，老年代采用标记-整理算法。



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

并行和并发概念补充：

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

## 4.3 Parallel Scavenge收集器

Parallel Scavenge 收集器类似于ParNew 收集器。那么它有什么特别之处呢？

```
-XX:+UseParallelGC
```

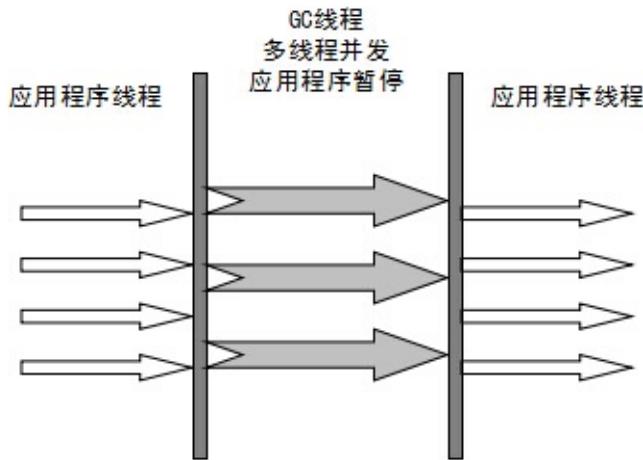
使用Parallel收集器+ 老年代串行

```
-XX:+UseParallelOldGC
```

使用Parallel收集器+ 老年代并行

**Parallel Scavenge**收集器关注点是吞吐量（高效率的利用CPU）。CMS等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）。所谓吞吐量就是CPU中用于运行用户代码的时间与CPU总消耗时间的比值。Parallel Scavenge收集器提供了很多参数供用户找到最合适的停顿时间或最大吞吐量，如果对于收集器运作不太了解的话，手工优化存在的话可以选择把内存管理优化交给虚拟机去完成也是一个不错的选择。

新生代采用复制算法，老年代采用标记-整理算法。



#### 4.4. Serial Old收集器

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

#### 4.5 Parallel Old收集器

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

#### 4.6 CMS收集器

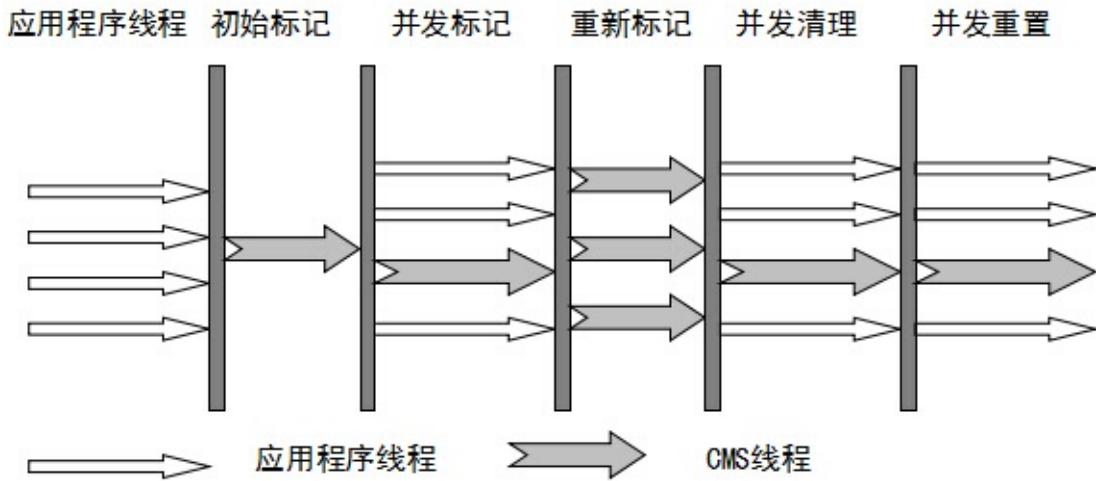
**CMS (Concurrent Mark Sweep)** 收集器是一种以获取最短回收停顿时间为为目标的收集器。它而非常符合在注重用户体验的应用上使用。

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

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

- **初始标记：**暂停所有的其他线程，并记录下直接与root相连的对象，速度很快；
- **并发标记：**同时开启GC和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以GC线程无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方。

- **重新标记：**重新标记阶段就是为了修正并发标记期间因为用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短
- **并发清除：**开启用户线程，同时GC线程开始对为标记的区域做清扫。



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

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

## 4.7 G1收集器

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

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

- **并行与并发：**G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短Stop-The-World停顿时间。部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让java程序继续执行。
- **分代收集：**虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但是还是保留了分代的概念。
- **空间整合：**与CMS的“标记--清理”算法不同，G1从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。
- **可预测的停顿：**这是G1相对于CMS的另一个大优势，降低停顿时间是G1 和 CMS 共同的关注点，但G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内。

G1收集器的运作大致分为以下几个步骤：

- 初始标记
- 并发标记
- 最终标记
- 筛选回收

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

参考：

- 《深入理解Java虚拟机：JVM高级特性与最佳实践（第二版）》

- <https://my.oschina.net/hosee/blog/644618>

Java面试通关手册 (Java学习指南) github地址 (欢迎star和pull) : [https://github.com/Snailclimb/Java\\_Guide](https://github.com/Snailclimb/Java_Guide)

下面是按jvm虚拟机知识点分章节总结的一些jvm学习与面试相关的一些东西。一般作为Java程序员在面试的时候一般会问的大多就是**Java内存区域**、**虚拟机垃圾算法**、**虚拟垃圾收集器**、**JVM内存管理**这些问题了。这些内容参考周的《深入理解Java虚拟机》中第二章和第三章就足够了对应下面的[深入理解虚拟机之Java内存区域](#) 和 [深入理解虚拟机之垃圾回收](#)这两篇文章。

## 常见面试题

[深入理解虚拟机之Java内存区域](#):

1. 介绍下Java内存区域（运行时数据区）。
2. 对象的访问定位的两种方式。

[深入理解虚拟机之垃圾回收](#)

1. 如何判断对象是否死亡（两种方法）。
2. 简单的介绍一下强引用、软引用、弱引用、虚引用（虚引用与软引用和弱引用的区别、使用软引用能带来的好处）。
3. 垃圾收集有哪些算法，各自的特点？
4. HotSpot为什么要分为新生代和老年代？
5. 常见的垃圾回收器有那些？
6. 介绍一下CMS,G1收集器。
7. Minor Gc和Full GC有什么不同呢？

[虚拟机性能监控和故障处理工具](#)

1. JVM调优的常见命令行工具有哪些？

[深入理解虚拟机之类文件结构](#)

1. 简单介绍一下Class类文件结构（常量池主要存放的是那两大常量？Class文件的继承关系是如何确定的？字段表、方法表、属性表主要包含那些信息？）

[深入理解虚拟机之虚拟机类加载机制](#)

1. 简单说说类加载过程，里面执行了哪些操作？
2. 对类加载器有了解吗？
3. 什么是双亲委派模型？
4. 双亲委派模型的工作过程以及使用它的好处。

## 推荐阅读

[深入理解虚拟机之虚拟机字节码执行引擎](#)

[《深入理解 Java 内存模型》读书笔记](#)（非常不错的文章）

[全面理解Java内存模型\(JMM\)及volatile关键字](#)

欢迎关注我的微信公众号：“Java面试通关手册”（一个有温度的微信公众号，期待与你共同进步~~~坚持原创，分享美文，分享各种Java学习资源）：



下面是自己学习设计模式的时候做的总结，有些是自己的原创文章，有些是网上写的比较好的文章，保存下来细细消化吧！

## 创建型模式：

### 创建型模式概述：

- 创建型模式(Creational Pattern)对类的实例化过程进行了抽象，能够将软件模块中对象的创建和对象的使用分离。为了使软件的结构更加清晰，外界对于这些对象只需要知道它们共同的接口，而不清楚其具体的实现细节，使整个系统的设计更加符合单一职责原则。
- 创建型模式在创建什么(What)，由谁创建(Who)，何时创建(When)等方面都为软件设计者提供了尽可能大的灵活性。创建型模式隐藏了类的实例的创建细节，通过隐藏对象如何被创建和组合在一起达到使整个系统独立的目的。

- ✓ **简单工厂模式 ( Simple Factory )** 
- ✓ **工厂方法模式 ( Factory Method )** 
- ✓ **抽象工厂模式 ( Abstract Factory )** 
- ✓ **建造者模式 ( Builder )** 
- ✓ **原型模式 ( Prototype )** 
- ✓ **单例模式 ( Singleton )** 

### 创建型模式系列文章推荐：

- 单例模式：

[深入理解单例模式——只有一个实例](#)

- 工厂模式：

[深入理解工厂模式——由对象工厂生成对象](#)

- 建造者模式：

[深入理解建造者模式——组装复杂的实例](#)

- 原型模式：

[深入理解原型模式——通过复制生成实例](#)

## 结构型模式：

### 结构型模式概述：

- **结构型模式(Structural Pattern)**：描述如何将类或者对象结合在一起形成更大的结构，就像搭积木，可以通过简单积木的组合形成复杂的、功能更为强大的结构



### 结构型模式系列文章推荐：

- 适配器模式：

[深入理解适配器模式——加个“适配器”以便于复用](#)

[适配器模式原理及实例介绍-IBM](#)

- 桥接模式：

[设计模式笔记16：桥接模式\(Bridge Pattern\)](#)

- 组合模式：

[大话设计模式—组合模式](#)

- 装饰模式：

[java模式—装饰者模式](#)

[Java设计模式-装饰者模式](#)

- 外观模式：

[java设计模式之外观模式（门面模式）](#)

- [享元模式](#):

[享元模式](#)

- [代理模式](#):

[代理模式原理及实例讲解（IBM出品，很不错）](#)

[轻松学，Java 中的代理模式及动态代理](#)

[Java代理模式及其应用](#)

## 行为型模式

### 行为型模式概述：

- 行为型模式(Behavioral Pattern)是对在不同的对象之间划分责任和算法的抽象化。
- 行为型模式不仅仅关注类和对象的结构，而且重点关注它们之间的相互作用。
- 通过行为型模式，可以更加清晰地划分类与对象的职责，并研究系统在运行时实例对象之间的交互。在系统运行时，对象并不是孤立的，它们可以通过相互通信与协作完成某些复杂功能，一个对象在运行时也将影响到其他对象的运行。

行为型模式分为类行为型模式和对象行为型模式两种：

- **类行为型模式：**类的行为型模式使用继承关系在几个类之间分配行为，类行为型模式主要通过多态等方式来分配父类与子类的职责。
- **对象行为型模式：**对象的行为型模式则使用对象的聚合关联关系来分配行为，对象行为型模式主要是通过对象关联等方式来分配两个或多个类的职责。根据“合成复用原则”，系统中要尽量使用关联关系来取代继承关系，因此大部分行为型设计模式都属于对象行为型设计模式。

### ✓ 职责链模式(Chain of Responsibility)



### ✓ 命令模式(Command)



### ✓ 解释器模式(Interpreter)



### ✓ 迭代器模式(Iterator)



### ✓ 中介者模式(Mediator)



### ✓ 备忘录模式(Memento)



### ✓ 观察者模式(Observer)



### ✓ 状态模式(State)



### ✓ 策略模式(Strategy)



### ✓ 模板方法模式(Template Method)



### ✓ 访问者模式(Visitor)



- [职责链模式](#):

[Java设计模式之责任链模式、职责链模式](#)

[责任链模式实现的三种方式](#)

- [命令模式](#):

- [解释器模式](#):

- [迭代器模式](#):

- 中介者模式：
- 备忘录模式：
- 观察者模式：
- 状态模式：
- 策略模式：
- 模板方法模式：
- 访问者模式：

- Queue
  - 什么是队列
  - 队列的种类
  - Java 集合框架中的队列 Queue
  - 推荐文章
- Set
  - 什么是 Set
  - 补充：有序集合与无序集合说明
  - HashSet 和 TreeSet 底层数据结构
  - 推荐文章
- List
  - 什么是 List
  - List 的常见实现类
  - ArrayList 和 LinkedList 源码学习
  - 推荐阅读
- Map
- 树

## Queue

### 什么是队列

队列是数据结构中比较重要的一种类型，它支持 FIFO，尾部添加、头部删除（先进队列的元素先出队列），跟我们生活中的排队类似。

### 队列的种类

- 单队列（单队列就是常见的队列，每次添加元素时，都是添加到队尾，存在“假溢出”的问题也就是明明有位置却不能添加的情况）
- 循环队列（避免了“假溢出”的问题）

### Java 集合框架中的队列 Queue

Java 集合中的 Queue 继承自 Collection 接口，Deque, LinkedList, PriorityQueue, BlockingQueue 等类都实现了它。Queue 用来存放等待处理元素的集合，这种场景一般用于缓冲、并发访问。除了继承 Collection 接口的一些方法，Queue 还添加了额外的添加、删除、查询操作。

### 推荐文章

- Java 集合深入理解 (9) : Queue 队列

## Set

### 什么是 Set

Set 继承于 Collection 接口，是一个不允许出现重复元素，并且无序的集合，主要 HashSet 和 TreeSet 两大实现类。在判断重复元素的时候，Set 集合会调用 hashCode() 和 equal() 方法来实现。

### 补充：有序集合与无序集合说明

- 有序集合：集合里的元素可以根据 key 或 index 访问 (List、Map)
- 无序集合：集合里的元素只能遍历。 (Set)

## HashSet 和 TreeSet 底层数据结构

**HashSet** 是哈希表结构，主要利用 HashMap 的 key 来存储元素，计算插入元素的 hashCode 来获取元素在集合中的位置；

**TreeSet** 是红黑树结构，每一个元素都是树中的一个节点，插入的元素都会进行排序；

## 推荐文章

- [Java集合--Set\(基础\)](#)

# List

## 什么是List

在 List 中，用户可以精确控制列表中每个元素的插入位置，另外用户可以通过整数索引（列表中的位置）访问元素，并搜索列表中的元素。与 Set 不同，List 通常允许重复的元素。另外 List 是有序集合而 Set 是无序集合。

## List的常见实现类

**ArrayList** 是一个数组队列，相当于动态数组。它由数组实现，随机访问效率高，随机插入、随机删除效率低。

**LinkedList** 是一个双向链表。它也可以被当作堆栈、队列或双端队列进行操作。LinkedList随机访问效率低，但随机插入、随机删除效率高。

**Vector** 是矢量队列，和ArrayList一样，它也是一个动态数组，由数组实现。但是ArrayList是非线程安全的，而Vector是线程安全的。

**Stack** 是栈，它继承于Vector。它的特性是：先进后出(FILO, First In Last Out)。相关阅读：[java数据结构与算法之栈\(Stack\) 设计与实现](#)

## ArrayList 和 LinkedList 源码学习

- [ArrayList 源码学习](#)
- [LinkedList 源码学习](#)

## 推荐阅读

- [java 数据结构与算法之顺序表与链表深入分析](#)

# Map

- [集合框架源码学习之 HashMap\(JDK1.8\)](#)
- [ConcurrentHashMap 实现原理及源码分析](#)

# 树

- [1 二叉树](#)

## [二叉树](#) (百度百科)

(1)[完全二叉树](#)——若设二叉树的高度为 $h$ , 除第 $h$ 层外, 其它各层( $1 \sim h-1$ )的结点数都达到最大个数, 第 $h$ 层有叶子结点, 并且叶子结点都是从左到右依次排布, 这就是完全二叉树。

(2)[满二叉树](#)——除了叶结点外每一个结点都有左右子叶且叶子结点都处在最底层的二叉树。

(3)[平衡二叉树](#)——平衡二叉树又被称为AVL树(区别于AVL算法), 它是一棵二叉排序树, 且具有以下性质: 它是一棵空树或它的左右两个子树的高度差的绝对值不超过1, 并且左右两个子树都是一棵平衡二叉树。

## • 2 完全二叉树

### [完全二叉树](#) (百度百科)

完全二叉树: 叶节点只能出现在最下层和次下层, 并且最下面一层的结点都集中在该层最左边的若干位置的二叉树

## • 3 满二叉树

### [满二叉树](#) (百度百科, 国内外的定义不同)

国内教程定义: 一个二叉树, 如果每一个层的结点数都达到最大值, 则这个二叉树就是满二叉树。也就是说, 如果一个二叉树的层数为 $K$ , 且结点总数是 $(2^k) - 1$ , 则它就是满二叉树。

## • 堆

### [数据结构之堆的定义](#)

堆是具有以下性质的完全二叉树: 每个结点的值都大于或等于其左右孩子结点的值, 称为大顶堆; 或者每个结点的值都小于或等于其左右孩子结点的值, 称为小顶堆

## • 4 二叉查找树 (BST)

### [浅谈算法和数据结构: 七 二叉查找树](#)

二叉查找树的特点:

1. 若任意节点的左子树不空, 则左子树上所有结点的值均小于它的根结点的值;
2. 若任意节点的右子树不空, 则右子树上所有结点的值均大于它的根结点的值;
3. 任意节点的左、右子树也分别为二叉查找树。
4. 没有键值相等的节点 (no duplicate nodes)。

## • 5 平衡二叉树 (Self-balancing binary search tree)

### [平衡二叉树](#) (百度百科, 平衡二叉树的常用实现方法有红黑树、AVL、替罪羊树、Treap、伸展树等)

## • 6 红黑树

- 红黑树特点:

1. 每个节点非红即黑;
2. 根节点总是黑色的;
3. 每个叶子节点都是黑色的空节点 (NIL节点);
4. 如果节点是红色的, 则它的子节点必须是黑色的 (反之不一定);
5. 从根节点到叶节点或空子节点的每条路径, 必须包含相同数目的黑色节点 (即相同的黑色高度)

- 红黑树的应用:

TreeMap、TreeSet以及JDK1.8之后的HashMap底层都用到了红黑树。

- 为什么要用红黑树

简单来说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。详细了解可以查看 [漫画：什么是红黑树？（也介绍到了二叉查找树，非常推荐）](#)

- 推荐文章：

- [漫画：什么是红黑树？（也介绍到了二叉查找树，非常推荐）](#)
- [寻找红黑树的操作手册（文章排版以及思路真的不错）](#)
- [红黑树深入剖析及Java实现（美团点评技术团队）](#)

## • 7 B-, B+, B\*树

[二叉树学习笔记之B树、B+树、B\\*树](#)

[《B-树，B+树，B\\*树详解》](#)

[《B-树，B+树与B\\*树的优缺点比较》](#)

B-树（或B树）是一种平衡的多路查找(又称排序)树，在文件系统中有所应用。主要用作文件的索引。其中的B就表示平衡(Balance)

1. B+ 树的叶子节点链表结构相比于 B- 树便于扫库，和范围检索。
2. B+树支持range-query(区间查询)非常方便，而B树不支持。这是数据库选用B+树的最主要原因。
3. B树是B+树的变体，B树分配新结点的概率比B+树要低，空间使用率更高；

## • 8 LSM 树

[\[HBase\] LSM树 VS B+树](#)

B+树最大的性能问题是会产生大量的随机IO

为了克服B+树的弱点，HBase引入了LSM树的概念，即Log-Structured Merge-Trees。

[LSM树由来、设计思想以及应用到HBase的索引](#)

## LeetCode

[LeetCode（中国）官网](#)

[如何高效地使用 LeetCode](#)

## 牛客网：

[牛客网首页](#)

### 剑指offer编程题

分类解析：

- (1) 斐波那契数列问题和跳台阶问题
- (2) 二维数组查找和替换空格问题
- (3) 数值的整数次方和调整数组元素顺序
- (4) 链表相关编程题
- (5) 栈变队列和栈的压入、弹出序列

### 2017校招真题

### 华为机试题

## 公司真题

### 网易2018校园招聘编程题真题集合

解析：

- 网易2018校招编程题1-3

[网易2018校招内推编程题集合](#)

[2017年校招全国统一模拟笔试\(第五场\)编程题集合](#)

[2017年校招全国统一模拟笔试\(第四场\)编程题集合](#)

[2017年校招全国统一模拟笔试\(第三场\)编程题集合](#)

[2017年校招全国统一模拟笔试\(第二场\)编程题集合](#)

[2017年校招全国统一模拟笔试\(第一场\)编程题集合](#)

[百度2017春招笔试真题编程题集合](#)

[网易2017春招笔试真题编程题集合](#)

[网易2017秋招编程题集合](#)

[网易有道2017内推编程题](#)

[滴滴出行2017秋招笔试真题-编程题汇总](#)

[腾讯2017暑期实习生编程题](#)

[今日头条2017客户端工程师实习生笔试题](#)

[今日头条2017后端工程师实习生笔试题](#)

## 排序算法：

[图解排序算法\(一\)之3种简单排序\(选择, 冒泡, 直接插入\)](#)

[图解排序算法\(二\)之希尔排序](#)

[图解排序算法\(三\)之堆排序](#)

[图解排序算法\(四\)之归并排序](#)

[图解排序算法\(五\)之快速排序——三数取中法](#)

本文主要对消息摘要算法和加密算法做了整理，包括MD5、SHA、DES、AES、RSA等，并且提供了相应算法的Java实现和测试。

## 一 消息摘要算法

### 1. 简介：

- 消息摘要算法的主要特征是加密过程不需要密钥，并且经过加密的数据无法被解密
- 只有输入相同的明文数据经过相同的消息摘要算法才能得到相同的密文。
- 消息摘要算法主要应用在“数字签名”领域，作为对明文的摘要算法。
- 著名的摘要算法有RSA公司的MD5算法和SHA-1算法及其大量的变体。

### 2. 特点：

1. 无论输入的消息有多长，计算出来的消息摘要的长度总是固定的。
2. 消息摘要看起来是“伪随机的”。也就是说对相同的信息求摘要结果相同。
3. 消息轻微改变生成的摘要变化会很大
4. 只能进行正向的信息摘要，而无法从摘要中恢复出任何的消息，甚至根本就找不到任何与原信息相关的信息

### 3. 应用：

消息摘要算法最常用的场景就是数字签名以及数据（密码）加密了。（一般平时做项目用的比较多的就是使用MD5对用户密码进行加密）

### 4. 何谓数字签名：

数字签名主要用到了非对称密钥加密技术与数字摘要技术。数字签名技术是将摘要信息用发送者的私钥加密，与原文一起传送给接收者。接收者只有用发送者的公钥才能解密被加密的摘要信息，然后用HASH函数对收到的原文产生一个摘要信息，与解密的摘要信息对比。如果相同，则说明收到的信息是完整的，在传输过程中没有被修改，否则说明信息被修改过。

因此数字签名能够验证信息的完整性。数字签名是个加密的过程，数字签名验证是个解密的过程。

### 5. 常见消息/数字摘要算法：

#### MD5：

##### 简介：

MD5的作用是让大容量信息在用数字签名软件签署私人密钥前被“压缩”成一种保密的格式（也就是把一个任意长度的字节串转换成一定长的十六进制数字串）。

##### 特点：

1. 压缩性：任意长度的数据，算出的MD5值长度都是固定的。
2. 容易计算：从原数据计算出MD5值很容易。
3. 抗修改性：对原数据进行任何改动，哪怕只修改1个字节，所得到的MD5值都有很大区别。

4. 强抗碰撞：已知原数据和其MD5值，想找到一个具有相同MD5值的数据（即伪造数据）是非常困难的。

## 代码实现：

利用JDK提供`java.security.MessageDigest`类实现MD5算法：

```
package com.snailclimb.ks.securityAlgorithm;

import java.security.MessageDigest;

public class MD5Demo {

 // test
 public static void main(String[] args) {
 System.out.println(getMD5Code("你若安好，便是晴天"));
 }

 private MD5Demo() {
 }

 // md5加密
 public static String getMD5Code(String message) {
 String md5Str = "";
 try {
 //创建MD5算法消息摘要
 MessageDigest md = MessageDigest.getInstance("MD5");
 //生成的哈希值的字节数组
 byte[] md5Bytes = md.digest(message.getBytes());
 md5Str = bytes2Hex(md5Bytes);
 } catch (Exception e) {
 e.printStackTrace();
 }
 return md5Str;
 }

 // 2进制转16进制
 public static String bytes2Hex(byte[] bytes) {
 StringBuffer result = new StringBuffer();
 int temp;
 try {
 for (int i = 0; i < bytes.length; i++) {
 temp = bytes[i];
 if(temp < 0) {
 temp += 256;
 }
 if (temp < 16) {
 result.append("0");
 }
 result.append(Integer.toHexString(temp));
 }
 } catch (Exception e) {
 e.printStackTrace();
 }
 return result.toString();
 }
}
```

结果：

```
6bab82679914f7cb480a120b532ffa80
```

注意`MessageDigest`类的几个方法：

```
static MessageDigest getInstance(String algorithm)//返回实现指定摘要算法的MessageDigest对象
```

```
byte[] digest(byte[] input)//使用指定的字节数组对摘要执行最终更新，然后完成摘要计算。
```

## 不利用Java提供的java.security.MessageDigest类实现MD5算法：

```
package com.snailclimb.ks.securityAlgorithm;

public class MD5{
 /*
 *四个链接变量
 */
 private final int A=0x67452301;
 private final int B=0xefcdab89;
 private final int C=0x98badcfe;
 private final int D=0x10325476;
 /*
 *ABCD的临时变量
 */
 private int Attemp,Btemp,Ctemp,Dtemp;

 /*
 *常量ti
 *公式:floor(abs(sin(i+1))×(2pow32))
 */
 private final int K[]={

 0xd76aa478,0xe8c7b756,0x242070db,0xc1bdceee,
 0xf57c0faf,0x4787c62a,0xa8304613,0xfd469501,0x698098d8,
 0x8b44f7af,0xfffff5bb1,0x895cd7be,0x6b901122,0xfd987193,
 0xa679438e,0x49b40821,0xf61e2562,0xc040b340,0x265e5a51,
 0xe9b6c7aa,0xd62f105d,0x02441453,0xd8a1e681,0xe7d3fbc8,
 0x21e1cde6,0xc33707d6,0xf4d50d87,0x455a14ed,0xa9e3e905,
 0xfcfa3f8,0x676f02d9,0x8d2a4c8a,0xffffa3942,0x8771f681,
 0x6d9d6122,0xfde5380c,0xa4beea44,0x4bdecfa9,0xf6bb4b60,
 0xebefbc70,0x289b7ec6,0xea127fa,0xd4ef3085,0x04881d05,
 0xd9d4d039,0xe6db99e5,0x1fa27cf8,0xc4ac5665,0xf4292244,
 0x432aff97,0xab9423a7,0xfc93a039,0x655b59c3,0x8f0ccc92,
 0xffeff47d,0x85845dd1,0x6fa87e4f,0xfe2ce6e0,0xa3014314,
 0x4e0811a1,0xf7537e82,0xbd3af235,0x2ad7d2bb,0xeb86d391};

 /*
 *向左位移数,计算方法未知
 */
 private final int s[]={7,12,17,22,7,12,17,22,7,12,17,22,7,
 12,17,22,5,9,14,20,5,9,14,20,5,9,14,20,
 4,11,16,23,4,11,16,23,4,11,16,23,4,11,16,23,6,10,
 15,21,6,10,15,21,6,10,15,21,6,10,15,21};

 /*
 *初始化函数
 */
 private void init(){
 Attemp=A;
 Btemp=B;
 Ctemp=C;
 Dtemp=D;
 }
 /*
 *移动一定位数
 */
 private int shift(int a,int s){
 return(a<<s)|(a>>(32-s));//右移的时候，高位一定要补零，而不是补充符号位
 }
 /*
 *主循环
 */
}
```

```

private void MainLoop(int M[]){
 int F,g;
 int a=Atemp;
 int b=Btemp;
 int c=Ctemp;
 int d=Dtemp;
 for(int i = 0; i < 64; i ++){
 if(i<16){
 F=(b&c)|((~b)&d);
 g=i;
 }else if(i<32){
 F=(d&b)|((~d)&c);
 g=(5*i+1)%16;
 }else if(i<48){
 F=b^c^d;
 g=(3*i+5)%16;
 }else{
 F=c^(b|(~d));
 g=(7*i)%16;
 }
 int tmp=d;
 d=c;
 c=b;
 b=b+shift(a+F+K[i]+M[g],s[i]);
 a=tmp;
 }
 Atemp=a+Atemp;
 Btemp=b+Btemp;
 Ctemp=c+Ctemp;
 Dtemp=d+Dtemp;
}

/*
*填充函数
*处理后应满足bits=448(mod512),字节就是bytes=56 (mode64)
*填充方式为先加一个0,其它位补零
*最后加上64位的原来长度
*/
private int[] add(String str){
 int num=((str.length()>8)?8:1); //以512位, 64个字节为一组
 int strByte[]=new int[num*16]; //64/4=16, 所以有16个整数
 for(int i=0;i<num*16;i++){ //全部初始化0
 strByte[i]=0;
 }
 int i;
 for(i=0;i<str.length();i++){
 strByte[i>>2]|=str.charAt(i)<<((i%4)*8); //一个整数存储四个字节, 小端序
 }
 strByte[i>>2]|=0x80<<((i%4)*8); //尾部添加1
 /*
 *添加原长度, 长度指位的长度, 所以要乘8, 然后是小端序, 所以放在倒数第二个, 这里长度只用了32位
 */
 strByte[num*16-2]=str.length()*8;
 return strByte;
}
/*
*调用函数
*/
public String getMD5(String source){
 init();
 int strByte[]=add(source);
 for(int i=0;i<strByte.length/16;i++){
 int num[]=new int[16];
 for(int j=0;j<16;j++){
 num[j]=strByte[i*16+j];
 }
 MainLoop(num);
 }
 return changeHex(Atemp)+changeHex(Btemp)+changeHex(Ctemp)+changeHex(Dtemp);
}

```

```

 }
 /*
 *整数变成16进制字符串
 */
 private String changeHex(int a){
 String str="";
 for(int i=0;i<4;i++){
 str+=String.format("%2s", Integer.toHexString(((a>>i*8)%10000)&0xffff)).replace(' ', '0');
 }
 return str;
 }
 /*
 *单例
 */
 private static MD5 instance;
 public static MD5 getInstance(){
 if(instance==null){
 instance=new MD5();
 }
 return instance;
 }
 private MD5(){}
 public static void main(String[] args){
 String str=MD5.getInstance().getMD5("你若安好，便是晴天");
 System.out.println(str);
 }
}

```

## SHA1:

对于长度小于 $2^{64}$ 位的消息，SHA1会产生一个160位(40个字符)的消息摘要。当接收到消息的时候，这个消息摘要可以用来验证数据的完整性。在传输的过程中，数据很可能发生变化，那么这时候就会产生不同的消息摘要。

SHA1有如下特性：

- 不可以从消息摘要中复原信息；
- 两个不同的消息不会产生同样的消息摘要,(但会有 $1 \times 10^{-48}$ 分之一的机率出现相同的消息摘要,一般使用时忽略)。

## 代码实现：

\*利用JDK提供java.security.MessageDigest类实现SHA1算法：

```

package com.snailclimb.ks.securityAlgorithm;

import java.io.UnsupportedEncodingException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class SHA1Demo {

 public static void main(String[] args) {
 // TODO Auto-generated method stub
 System.out.println(getSha1("你若安好，便是晴天"));
 }

 public static String getSha1(String str) {
 if (null == str || 0 == str.length()) {
 return null;
 }
 }
}

```

```

char[] hexDigits = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f' };
try {
 //创建SHA1算法消息摘要对象
 MessageDigest mdTemp = MessageDigest.getInstance("SHA1");
 //使用指定的字节数组更新摘要。
 mdTemp.update(str.getBytes("UTF-8"));
 //生成的哈希值的字节数组
 byte[] md = mdTemp.digest();
 //SHA1算法生成信息摘要关键过程
 int j = md.length;
 char[] buf = new char[j * 2];
 int k = 0;
 for (int i = 0; i < j; i++) {
 byte byte0 = md[i];
 buf[k++] = hexDigits[byte0 >>> 4 & 0xf];
 buf[k++] = hexDigits[byte0 & 0xf];
 }
 return new String(buf);
} catch (NoSuchAlgorithmException e) {
 e.printStackTrace();
} catch (UnsupportedEncodingException e) {
 e.printStackTrace();
}
return "0";
}
}

```

结果：

```
8ce764110a42da9b08504b20e26b19c9e3382414
```

## 二 加密算法

### 1. 简介：

- 加密技术包括两个元素：加密算法和密钥。
- 加密算法是将普通的文本（或者可以理解的信息）与一串数字（密钥）的结合，产生不可理解的密文的步骤。
- 密钥是用来对数据进行编码和解码的一种算法。
- 在安全保密中，可通过适当的密钥加密技术和管理机制来保证网络的信息通讯安全。

### 2. 分类：

密钥加密技术的密码体制分为对称密钥体制和非对称密钥体制两种。相应地，对数据加密的技术分为两类，即对称加密（私人密钥加密）和非对称加密（公开密钥加密）。

对称加密以数据加密标准（DES，Data Encryption Standard）算法为典型代表，非对称加密通常以RSA（Rivest Shamir Adleman）算法为代表。

对称加密的加密密钥和解密密钥相同。非对称加密的加密密钥和解密密钥不同，加密密钥可以公开而解密密钥需要保密。

### 3. 应用：

常被用在电子商务或者其他需要保证网络传输安全的范围。

## 4. 对称加密：

加密密钥和解密密钥相同的加密算法。

对称加密算法使用起来简单快捷，密钥较短，且破译困难，除了数据加密标准（DES），另一个对称密钥加密系统是国际数据加密算法（IDEA），它比DES的加密性好，而且对计算机功能要求也没有那么高。IDEA加密标准由PGP（Pretty Good Privacy）系统使用。

### DES:

DES全称为Data Encryption Standard，即数据加密标准，是一种使用密钥加密的块算法，现在已经过时。

### 代码实现：

DES算法实现：

```
package com.snailclimb.ks.securityAlgorithm;

import java.io.UnsupportedEncodingException;
import java.security.SecureRandom;
import javax.crypto.spec.DESKeySpec;
import javax.crypto.SecretKeyFactory;
import javax.crypto.SecretKey;
import javax.crypto.Cipher;

/**
 * DES加密介绍 DES是一种对称加密算法，所谓对称加密算法即：加密和解密使用相同密钥的算法。DES加密算法出自IBM的研究，

 * 后来被美国政府正式采用，之后开始广泛流传，但是近些年使用越来越少，因为DES使用56位密钥，以现代计算能力，

 * 24小时内即可被破解。虽然如此，在某些简单应用中，我们还是可以使用DES加密算法，本文简单讲解DES的JAVA实现。

 * 注意：DES加密和解密过程中，密钥长度都必须是8的倍数
 */
public class DesDemo {
 public DesDemo() {
 }

 // 测试
 public static void main(String args[]) {
 // 待加密内容
 String str = "cryptology";
 // 密码，长度要是8的倍数
 String password = "95880288";

 byte[] result;
 try {
 result = DesDemo.encrypt(str.getBytes(), password);
 System.out.println("加密后: " + result);
 byte[] decryResult = DesDemo.decrypt(result, password);
 System.out.println("解密后: " + new String(decryResult));
 } catch (UnsupportedEncodingException e2) {
 // TODO Auto-generated catch block
 e2.printStackTrace();
 } catch (Exception e1) {
 e1.printStackTrace();
 }
 }

 // 直接将如上内容解密

 /**
 * 加密
 *
 * @param datasource
 * byte[]
 * @param password
 */
 public static byte[] encrypt(byte[] datasource, String password) throws UnsupportedEncodingException {
 DESKeySpec desKeySpec = new DESKeySpec(password);
 SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
 SecretKey secretKey = keyFactory.generateSecret(desKeySpec);
 Cipher cipher = Cipher.getInstance("DES");
 cipher.init(Cipher.ENCRYPT_MODE, secretKey);
 byte[] encryptedData = cipher.doFinal(datasource);
 return encryptedData;
 }

 /**
 * 解密
 *
 * @param encryptedData
 * byte[]
 * @param password
 */
 public static byte[] decrypt(byte[] encryptedData, String password) throws UnsupportedEncodingException {
 DESKeySpec desKeySpec = new DESKeySpec(password);
 SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
 SecretKey secretKey = keyFactory.generateSecret(desKeySpec);
 Cipher cipher = Cipher.getInstance("DES");
 cipher.init(Cipher.DECRYPT_MODE, secretKey);
 byte[] decryptedData = cipher.doFinal(encryptedData);
 return decryptedData;
 }
}
```

```

 * String
 * @return byte[]
 */
public static byte[] encrypt(byte[] datasource, String password) {
 try {
 SecureRandom random = new SecureRandom();
 DESKeySpec desKey = new DESKeySpec(password.getBytes());
 // 创建一个密匙工厂, 然后用它把DESKeySpec转换成
 SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
 SecretKey securekey = keyFactory.generateSecret(desKey);
 // Cipher对象实际完成加密操作
 Cipher cipher = Cipher.getInstance("DES");
 // 用密匙初始化Cipher对象,ENCRYPT_MODE用于将 Cipher 初始化为加密模式的常量
 cipher.init(Cipher.ENCRYPT_MODE, securekey, random);
 // 现在, 获取数据并加密
 // 正式执行加密操作
 return cipher.doFinal(datasource); // 按单部分操作加密或解密数据, 或者结束一个多部分操作
 } catch (Throwable e) {
 e.printStackTrace();
 }
 return null;
}

/**
 * 解密
 *
 * @param src
 * byte[]
 * @param password
 * String
 * @return byte[]
 * @throws Exception
 */
public static byte[] decrypt(byte[] src, String password) throws Exception {
 // DES算法要求有一个可信任的随机数源
 SecureRandom random = new SecureRandom();
 // 创建一个DESKeySpec对象
 DESKeySpec desKey = new DESKeySpec(password.getBytes());
 // 创建一个密匙工厂
 SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");// 返回实现指定转换的
 // Cipher
 // 对象
 // 将DESKeySpec对象转换成SecretKey对象
 SecretKey securekey = keyFactory.generateSecret(desKey);
 // Cipher对象实际完成解密操作
 Cipher cipher = Cipher.getInstance("DES");
 // 用密匙初始化Cipher对象
 cipher.init(Cipher.DECRYPT_MODE, securekey, random);
 // 真正开始解密操作
 return cipher.doFinal(src);
}
}

```

结果：

```

加密后: [B@50cbc42f
解密后: cryptology

```

## IDEA:

- 这种算法是在DES算法的基础上发展出来的，类似于三重DES。
- 发展IDEA也是因为感到DES具有密钥太短等缺点。
- IDEA的密钥为128位，这么长的密钥在今后若干年内应该是安全的。
- 在实际项目中用到的很少了解即可。

## 代码实现：

IDEA算法实现

```

package com.snailclimb.ks.securityAlgorithm;

import java.security.Key;
import java.security.SecureRandom;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;

import org.apache.commons.codec.binary.Base64;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class IDEADemo {
 public static void main(String args[]) {
 bcIDEA();
 }
 public static void bcIDEA() {
 String src = "www.xttblog.com security idea";
 try {
 Security.addProvider(new BouncyCastleProvider());

 //生成key
 KeyGenerator keyGenerator = KeyGenerator.getInstance("IDEA");
 keyGenerator.init(128);
 SecretKey secretKey = keyGenerator.generateKey();
 byte[] keyBytes = secretKey.getEncoded();

 //转换密钥
 Key key = new SecretKeySpec(keyBytes, "IDEA");

 //加密
 Cipher cipher = Cipher.getInstance("IDEA/ECB/ISO10126Padding");
 cipher.init(Cipher.ENCRYPT_MODE, key);
 byte[] result = cipher.doFinal(src.getBytes());
 System.out.println("bc idea encrypt : " + Base64.encodeBase64String(result));

 //解密
 cipher.init(Cipher.DECRYPT_MODE, key);
 result = cipher.doFinal(result);
 System.out.println("bc idea decrypt : " + new String(result));
 } catch (Exception e) {
 e.printStackTrace();
 }
 }
}

```

## 5. 非对称加密：

- 与对称加密算法不同，非对称加密算法需要两个密钥：公开密钥（publickey）和私有密钥（privatekey）。
- 公开密钥与私有密钥是一对，如果用公开密钥对数据进行加密，只有用对应的私有密钥才能解密；
- 如果用私有密钥对数据进行加密，那么只有用对应的公开密钥才能解密。
- 因为加密和解密使用的是两个不同的密钥，所以这种算法叫作非对称加密算法。

### RAS:

RSA是目前最有影响力和最常用的公钥加密算法。它能够抵抗到目前为止已知的绝大多数密码攻击，已被ISO推荐为公钥数据加密标准。

## 代码实现：

RAS算法实现：

```

package com.snailclimb.ks.securityAlgorithm;

import org.apache.commons.codec.binary.Base64;

import java.security.*;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.HashMap;
import java.util.Map;

import javax.crypto.Cipher;

/**
 * Created by humf.需要依赖 commons-codec 包
 */
public class RSADemo {

 public static void main(String[] args) throws Exception {
 Map<String, Key> keyMap = initKey();
 String publicKey = getPublicKey(keyMap);
 String privateKey = getPrivateKey(keyMap);

 System.out.println(keyMap);
 System.out.println("-----");
 System.out.println(publicKey);
 System.out.println("-----");
 System.out.println(privateKey);
 System.out.println("-----");
 byte[] encryptByPrivateKey = encryptByPrivateKey("123456".getBytes(), privateKey);
 byte[] encryptByPublicKey = encryptByPublicKey("123456", publicKey);
 System.out.println(encryptByPrivateKey);
 System.out.println("-----");
 System.out.println(encryptByPublicKey);
 System.out.println("-----");
 String sign = sign(encryptByPrivateKey, privateKey);
 System.out.println(sign);
 System.out.println("-----");
 boolean verify = verify(encryptByPrivateKey, publicKey, sign);
 System.out.println(verify);
 System.out.println("-----");
 byte[] decryptByPublicKey = decryptByPublicKey(encryptByPrivateKey, publicKey);
 byte[] decryptByPrivateKey = decryptByPrivateKey(encryptByPublicKey, privateKey);
 System.out.println(decryptByPublicKey);
 System.out.println("-----");
 System.out.println(decryptByPrivateKey);
 }

 public static final String KEY_ALGORITHM = "RSA";
 public static final String SIGNATURE_ALGORITHM = "MD5withRSA";

 private static final String PUBLIC_KEY = "RSAPublicKey";
 private static final String PRIVATE_KEY = "RSAPrivateKey";

 public static byte[] decryptBASE64(String key) {
 return Base64.decodeBase64(key);
 }

 public static String encryptBASE64(byte[] bytes) {
 return Base64.encodeBase64String(bytes);
 }

 /**
 * 用私钥对信息生成数字签名

```

```

/*
 * @param data
 * 加密数据
 * @param privateKey
 * 私钥
 * @return
 * @throws Exception
 */
public static String sign(byte[] data, String privateKey) throws Exception {
 // 解密由base64编码的私钥
 byte[] keyBytes = decryptBASE64(privateKey);
 // 构造PKCS8EncodedKeySpec对象
 PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(keyBytes);
 // KEY_ALGORITHM 指定的加密算法
 KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
 // 取私钥匙对象
 PrivateKey priKey = keyFactory.generatePrivate(pkcs8KeySpec);
 // 用私钥对信息生成数字签名
 Signature signature = Signature.getInstance(SIGNATURE_ALGORITHM);
 signature.initSign(priKey);
 signature.update(data);
 return encryptBASE64(signature.sign());
}

/**
 * 校验数字签名
 *
 * @param data
 * 加密数据
 * @param publicKey
 * 公钥
 * @param sign
 * 数字签名
 * @return 校验成功返回true 失败返回false
 * @throws Exception
 */
public static boolean verify(byte[] data, String publicKey, String sign) throws Exception {
 // 解密由base64编码的公钥
 byte[] keyBytes = decryptBASE64(publicKey);
 // 构造X509EncodedKeySpec对象
 X509EncodedKeySpec keySpec = new X509EncodedKeySpec(keyBytes);
 // KEY_ALGORITHM 指定的加密算法
 KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
 // 取公钥匙对象
 PublicKey pubKey = keyFactory.generatePublic(keySpec);
 Signature signature = Signature.getInstance(SIGNATURE_ALGORITHM);
 signature.initVerify(pubKey);
 signature.update(data);
 // 验证签名是否正常
 return signature.verify(decryptBASE64(sign));
}

public static byte[] decryptByPrivateKey(byte[] data, String key) throws Exception {
 // 对密钥解密
 byte[] keyBytes = decryptBASE64(key);
 // 取得私钥
 PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(keyBytes);
 KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
 Key privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
 // 对数据解密
 Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
 cipher.init(Cipher.DECRYPT_MODE, privateKey);
 return cipher.doFinal(data);
}

/**
 * 解密

 * 用私钥解密
 *

```

```

 * @param data
 * @param key
 * @return
 * @throws Exception
 */
 public static byte[] decryptByPrivateKey(String data, String key) throws Exception {
 return decryptByKey(decryptBASE64(data), key);
 }

 /**
 * 解密

 * 用公钥解密
 *
 * @param data
 * @param key
 * @return
 * @throws Exception
 */
 public static byte[] decryptByPublicKey(byte[] data, String key) throws Exception {
 // 对密钥解密
 byte[] keyBytes = decryptBASE64(key);
 // 取得公钥
 X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(keyBytes);
 KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
 Key publicKey = keyFactory.generatePublic(x509KeySpec);
 // 对数据解密
 Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
 cipher.init(Cipher.DECRYPT_MODE, publicKey);
 return cipher.doFinal(data);
 }

 /**
 * 加密

 * 用公钥加密
 *
 * @param data
 * @param key
 * @return
 * @throws Exception
 */
 public static byte[] encryptByPublicKey(String data, String key) throws Exception {
 // 对公钥解密
 byte[] keyBytes = decryptBASE64(key);
 // 取得公钥
 X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(keyBytes);
 KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
 Key publicKey = keyFactory.generatePublic(x509KeySpec);
 // 对数据加密
 Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
 cipher.init(Cipher.ENCRYPT_MODE, publicKey);
 return cipher.doFinal(data.getBytes());
 }

 /**
 * 加密

 * 用私钥加密
 *
 * @param data
 * @param key
 * @return
 * @throws Exception
 */
 public static byte[] encryptByPrivateKey(byte[] data, String key) throws Exception {
 // 对密钥解密
 byte[] keyBytes = decryptBASE64(key);
 // 取得私钥
 PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(keyBytes);
 KeyFactory keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
 Key privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
 }
}

```

```

 // 对数据加密
 Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
 cipher.init(Cipher.ENCRYPT_MODE, privateKey);
 return cipher.doFinal(data);
}

/**
 * 取得私钥
 *
 * @param keyMap
 * @return
 * @throws Exception
 */
public static String getPrivateKey(Map<String, Key> keyMap) throws Exception {
 Key key = (Key) keyMap.get(PRIVATE_KEY);
 return encryptBASE64(key.getEncoded());
}

/**
 * 取得公钥
 *
 * @param keyMap
 * @return
 * @throws Exception
 */
public static String getPublicKey(Map<String, Key> keyMap) throws Exception {
 Key key = keyMap.get(PUBLIC_KEY);
 return encryptBASE64(key.getEncoded());
}

/**
 * 初始化密钥
 *
 * @return
 * @throws Exception
 */
public static Map<String, Key> initKey() throws Exception {
 KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance(KEY_ALGORITHM);
 keyPairGen.initialize(1024);
 KeyPair keyPair = keyPairGen.generateKeyPair();
 Map<String, Key> keyMap = new HashMap(2);
 keyMap.put(PUBLIC_KEY, keyPair.getPublic());// 公钥
 keyMap.put(PRIVATE_KEY, keyPair.getPrivate());// 私钥
 return keyMap;
}
}

```

结果：

```

{RSAPublicKey=Sun RSA public key, 1024 bits
 modulus: 1153288260860478739026064565710349765388365539987453679818489116779680625718316266744996508543182072
8041996076702060125307173955516138813558948728483454394036148839677137496052688313364180017227019245376245731
80276356615050309809260289965219855862692230362893996010057188170525719351126759886050891484226169
public exponent: 65537, RSAPrivatekey=sun.security.rsa.RSAPrivateCrtKeyImpl@93479}

MIGFMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCkO9PBTOFJQTkzznALN62PU7xd9YFjXrt2dP0Gj3wwhymb0U8HLoCztjwpLXHgbpBUJlGmbu
RV955M1BkZ1kr5dkZYR5x1g04x0nu8rEipy4AAMcpFttfiarIZrtzL9pKEvE0xAbltVN4yzFDr3IjBqY46aHna7YjwhXI0xHieQIDAQAB

MIICdQIBADANBgkqhkiG9w0BAQEFAASCA18wggJbAgEAAoGBAKQ708FM4U1BOTPOcAs3rY9TuLF31gWNeu3Z084aPfDCHKzs5TwugL02PCktce
BukFQmUaZtRFX3nkzUGRnwSv12R1hHnHwA7jE6e7ysSKnLgAAxykW21+Jqshmu3Mv2koS8Q7EAGW1U3jLMU0vcimGpjpoedrtiPCFcjtEEj5Ag
MBAAECgYAK4sx0a8IjE0exv2U92Rrv/SSo3sCY7Z/QVdfT2V9xrewo09+V9HF/7iYDDWffKYInAimvVl7JM/iSLxza0ZFv29VMpyDcr4TigYmW
wBlk7ZbxSTkqdNwxldMme0Tn1py53MuM+1V1K3rzNvJjuzaZFAevU7vUnwQwD+JGQYQJBAM9HbaC+dF3PJ2mkXeKHpDS1ZPaSFdrdzd/GvhFi
/cJAMM+Uz6PmpkosNRtOpSYw10MRamLztrHhfQoqSk3S8CQQDK1ql1jGvVdqw50jqxktR7MmOsWUVZdWiBN+6ojxBgA0yVn0n7vkdaAgEZbj8
9WG0VPEu3hd4AgXFZHDfxeDXAkBvSn7nE9t/Et7ihfI2UhGJ08UxNMfNMB5Skebyb7eMYEDs67ZHdpjMOfypcMyTatzj5wjwQ3zyMvb1ZX+ON
bzAkAX4ysRy9wL+icXLuo0Gfhkk+WrnSyUldaUGH0yRb2kecn00xN/1gGlxSvB+ac910zRHCOT1+Uo6nbmq0g3PFAkAyqa4eT7G9GXfnckgW
1Kdkn72w/ODpozgfhTLNx0SGw1ITML3c4THtth5h3zLi3AF9zJ020+k6ajRbV0szHHI

```

```
[B@387c703b
[B@224aed64
1a4Hc4n/UbeBu0z9iLRuwKvv014Si0JMXk05qdJvKBsw0M1nsrM+89a3p73yMrb1dAnCU/2kg00PtFpvM68pzxTe1u/5nX/25iIyUXALlwVRptJ
yjzFE83g2IX0XEv/Dxqr1RCRcrMHOLQM0oBoxZCaChmyw1Ub4wsSs6Ndxb9M=
true
[B@c39f790
[B@71e7a66b
```

- 说明
- 1. KMP 算法
- 2. 替换空格
- 3. 最长公共前缀
- 4. 回文串
  - 4.1. 最长回文串
  - 4.2. 验证回文串
  - 4.3. 最长回文子串
  - 4.4. 最长回文子序列
- 5. 括号匹配深度
- 6. 把字符串转换成整数

## 说明

- 本文作者: wwwxmu
- 原文地址:<https://www.weiweiblog.cn/13string/>
- 作者的博客站点: <https://www.weiweiblog.cn/> (推荐哦! )

考虑到篇幅问题，我会分两次更新这个内容。本篇文章只是原文的一部分，我在原文的基础上增加了部分内容以及修改了部分代码和注释。另外，我增加了爱奇艺 2018 秋招 Java: `求给定合法括号序列的深度` 这道题。所有代码均编译成功，并带有注释，欢迎各位享用！

## 1. KMP 算法

谈到字符串问题，不得不提的就是 KMP 算法，它是用来解决字符串查找的问题，可以在一个字符串 (S) 中查找一个子串 (W) 出现的位置。KMP 算法把字符匹配的时间复杂度缩小到  $O(m+n)$ ，而空间复杂度也只有  $O(m)$ 。因为“暴力搜索”的方法会反复回溯主串，导致效率低下，而 KMP 算法可以利用已经部分匹配这个有效信息，保持主串上的指针不回溯，通过修改子串的指针，让模式串尽量地移动到有效的位置。

具体算法细节请参考：

- 字符串匹配的KMP算法：  
[http://www.ruanyifeng.com/blog/2013/05/Knuth%E2%80%93Morris%E2%80%93Pratt\\_algorithm.html](http://www.ruanyifeng.com/blog/2013/05/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm.html)
- 从头到尾彻底理解KMP：[https://blog.csdn.net/v\\_july\\_v/article/details/7041827](https://blog.csdn.net/v_july_v/article/details/7041827)
- 如何更好的理解和掌握 KMP 算法?: <https://www.zhihu.com/question/21923021>
- KMP 算法详细解析: <https://blog.sengxian.com/algorithms/kmp>
- 图解 KMP 算法: <http://blog.jobbole.com/76611/>
- 汪都能听懂的KMP字符串匹配算法【双语字幕】：<https://www.bilibili.com/video/av3246487/?from=search&seid=17173603269940723925>
- KMP字符串匹配算法1：<https://www.bilibili.com/video/av11866460?from=search&seid=12730654434238709250>

除此之外，再来了解一下BM算法！

BM 算法也是一种精确字符串匹配算法，它采用从右向左比较的方法，同时应用到了两种启发式规则，即坏字符规则 和好后缀规则，来决定向右跳跃的距离。基本思路就是从右往左进行字符匹配，遇到不匹配的字符后从坏字符表和好后缀表找一个最大的右移值，将模式串右移继续匹配。《字符串匹配的KMP算法》：[http://www.ruanyifeng.com/blog/2013/05/Knuth%E2%80%93Morris%E2%80%93Pratt\\_algorithm.html](http://www.ruanyifeng.com/blog/2013/05/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm.html)

## 2. 替换空格

剑指offer：请实现一个函数，将一个字符串中的每个空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

这里我提供了两种方法：①常规方法；②利用 API 解决。

```
//https://www.weiweiblog.cn/replacespace/
public class Solution {

 /**
 * 第一种方法：常规方法。利用String.charAt(i)以及String.valueOf(char).equals(" ")
 * 遍历字符串并判断元素是否为空格。是则替换为"%20",否则不替换
 */
 public static String replaceSpace(StringBuffer str) {

 int length = str.length();
 // System.out.println("length=" + length);
 StringBuffer result = new StringBuffer();
 for (int i = 0; i < length; i++) {
 char b = str.charAt(i);
 if (String.valueOf(b).equals(" ")) {
 result.append("%20");
 } else {
 result.append(b);
 }
 }
 return result.toString();
 }

 /**
 * 第二种方法：利用API替换掉所用空格，一行代码解决问题
 */
 public static String replaceSpace2(StringBuffer str) {

 return str.toString().replaceAll("\\s", "%20");
 }
}
```

### 3. 最长公共前缀

Leetcode: 编写一个函数来查找字符串数组中的最长公共前缀。如果不存在公共前缀，返回空字符串 ""。

示例 1：

```
输入: ["flower","flow","flight"]
输出: "fl"
```

示例 2：

```
输入: ["dog","racecar","car"]
输出: ""
解释: 输入不存在公共前缀。
```

思路很简单！先利用Arrays.sort(strs)为数组排序，再将数组第一个元素和最后一个元素的字符从前往后对比即可！

```
//https://leetcode-cn.com/problems/longest-common-prefix/description/
public class Main {
 public static String replaceSpace(String[] strs) {

 // 数组长度
 int len = strs.length;
```

```

// 用于保存结果
StringBuffer res = new StringBuffer();
// 注意: ==是赋值, ===是判断
if (strs == null || strs.length == 0) {
 return "";
}
// 给字符串数组的元素按照升序排序(包含数字的话, 数字会排在前面)
Arrays.sort(strs);
int m = strs[0].length();
int n = strs[len - 1].length();
int num = Math.min(m, n);
for (int i = 0; i < num; i++) {
 if (strs[0].charAt(i) == strs[len - 1].charAt(i)) {
 res.append(strs[0].charAt(i));
 } else
 break;
}
return res.toString();
}

//测试
public static void main(String[] args) {
 String[] strs = { "customer", "car", "cat" };
 System.out.println(Main.replaceSpace(strs)); //c
}
}

```

## 4. 回文串

### 4.1. 最长回文串

LeetCode: 给定一个包含大写字母和小写字母的字符串, 找到通过这些字母构造出的最长的回文串。在构造过程中, 请注意区分大小写。比如 "Aa" 不能当做一个回文字符串。注意: 假设字符串的长度不会超过 1010。

回文串: “回文串”是一个正读和反读都一样的字符串, 比如“level”或者“noon”等等就是回文串。——百度百科 地址: <https://baike.baidu.com/item/%E5%9B%9E%E6%96%87%E4%B8%B2/1274921?fr=aladdin>

示例 1:

```

输入:
"abccccdd"

输出:
7

解释:
我们可以构造的最长的回文串是"dccaccd", 它的长度是 7。

```

我们上面已经知道了什么是回文串? 现在我们考虑一下可以构成回文串的两种情况:

- 字符出现次数为双数的组合
- 字符出现次数为双数的组合+一个只出现一次的字符

统计字符出现的次数即可, 双数才能构成回文。因为允许中间一个数单独出现, 比如“abcba”, 所以如果最后有字母落单, 总长度可以加 1。首先将字符串转变为字符数组。然后遍历该数组, 判断对应字符是否在hashset中, 如果不在就加进去, 如果在就让count++, 然后移除该字符! 这样就能找到出现次数为双数的字符个数。

```

//https://leetcode-cn.com/problems/longest-palindrome/description/
class Solution {
 public int longestPalindrome(String s) {

```

```

if (s.length() == 0)
 return 0;
// 用于存放字符
HashSet<Character> hashset = new HashSet<Character>();
char[] chars = s.toCharArray();
int count = 0;
for (int i = 0; i < chars.length; i++) {
 if (!hashset.contains(chars[i])) {// 如果hashset没有该字符就保存进去
 hashset.add(chars[i]);
 } else {// 如果有，就让count++（说明找到了一个成对的字符），然后把该字符移除
 hashset.remove(chars[i]);
 count++;
 }
}
return hashset.isEmpty() ? count * 2 : count * 2 + 1;
}
}

```

## 4.2. 验证回文串

LeetCode: 给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。说明：本题中，我们将空字符串定义为有效的回文串。

示例 1：

```

输入: "A man, a plan, a canal: Panama"
输出: true

```

示例 2：

```

输入: "race a car"
输出: false

```

```

//https://leetcode-cn.com/problems/valid-palindrome/description/
class Solution {
 public boolean isPalindrome(String s) {
 if (s.length() == 0)
 return true;
 int l = 0, r = s.length() - 1;
 while (l < r) {
 // 从头和尾开始向中间遍历
 if (!Character.isLetterOrDigit(s.charAt(l))) {// 字符不是字母和数字的情况
 l++;
 } else if (!Character.isLetterOrDigit(s.charAt(r))) {// 字符不是字母和数字的情况
 r--;
 } else {
 // 判断二者是否相等
 if (Character.toLowerCase(s.charAt(l)) != Character.toLowerCase(s.charAt(r)))
 return false;
 l++;
 r--;
 }
 }
 return true;
 }
}

```

## 4.3. 最长回文子串

Leetcode: LeetCode: 最长回文子串 给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

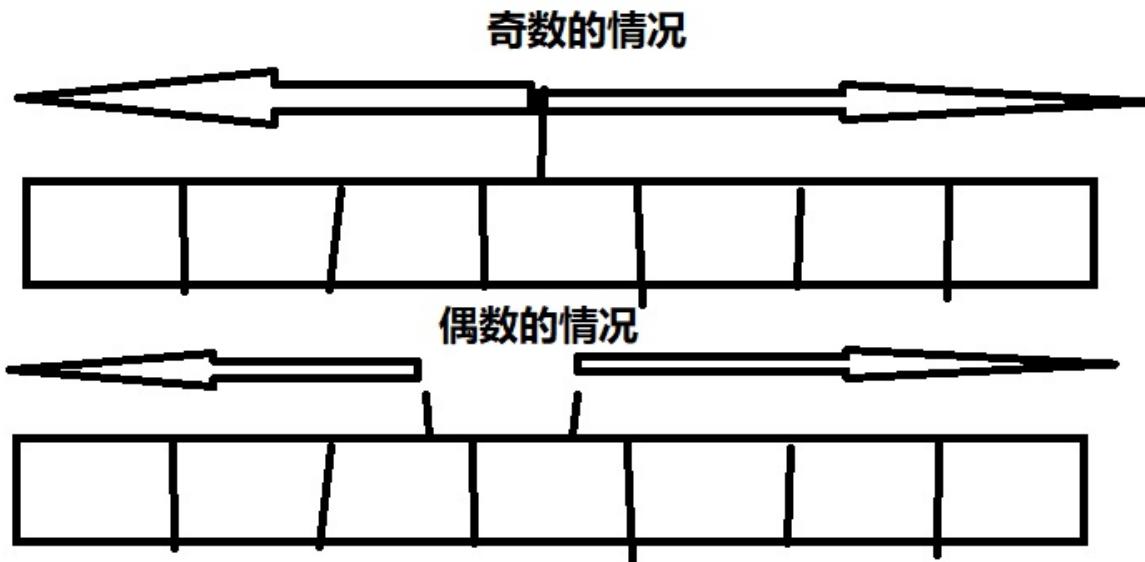
## 示例 1：

```
输入: "babad"
输出: "bab"
注意: "aba"也是一个有效答案。
```

## 示例 2：

```
输入: "cbbd"
输出: "bb"
```

以某个元素为中心，分别计算偶数长度的回文最大长度和奇数长度的回文最大长度。给大家大致花了个草图，不要嫌弃！



```
//https://leetcode-cn.com/problems/longest-palindromic-substring/description/
class Solution {
 private int index, len;

 public String longestPalindrome(String s) {
 if (s.length() < 2)
 return s;
 for (int i = 0; i < s.length() - 1; i++) {
 PalindromeHelper(s, i, i);
 PalindromeHelper(s, i, i + 1);
 }
 return s.substring(index, index + len);
 }

 public void PalindromeHelper(String s, int l, int r) {
 while (l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) {
 l--;
 r++;
 }
 if (len < r - l - 1) {
 index = l + 1;
 len = r - l - 1;
 }
 }
}
```

## 4.4. 最长回文子序列

LeetCode: 最长回文子序列 给定一个字符串s，找到其中最长的回文子序列。可以假设s的最大长度为1000。最长回文子序列和上一题最长回文子串的区别是，子串是字符串中连续的一个序列，而子序列是字符串中保持相对位置的字符序列，例如，“bbbb”可以是字符串“bbbab”的子序列但不是子串。

给定一个字符串s，找到其中最长的回文子序列。可以假设s的最大长度为1000。

示例 1：

```
输入：
"bbbab"
输出：
4
```

一个可能的最长回文子序列为 "bbbb"。

示例 2：

```
输入：
"cbbd"
输出：
2
```

一个可能的最长回文子序列为 "bb"。

动态规划：  $dp[i][j] = dp[i+1][j-1] + 2$  if  $s.charAt(i) == s.charAt(j)$  otherwise,  $dp[i][j] = \max(dp[i+1][j], dp[i][j-1])$

```
class Solution {
 public int longestPalindromeSubseq(String s) {
 int len = s.length();
 int [][] dp = new int[len][len];
 for(int i = len - 1; i >= 0; i--){
 dp[i][i] = 1;
 for(int j = i+1; j < len; j++){
 if(s.charAt(i) == s.charAt(j))
 dp[i][j] = dp[i+1][j-1] + 2;
 else
 dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1]);
 }
 }
 return dp[0][len-1];
 }
}
```

## 5. 括号匹配深度

爱奇艺 2018 秋招 Java：一个合法的括号匹配序列有以下定义：

1. 空串""是一个合法的括号匹配序列
2. 如果“X”和“Y”都是合法的括号匹配序列,“XY”也是一个合法的括号匹配序列
3. 如果“X”是一个合法的括号匹配序列,那么“(X)”也是一个合法的括号匹配序列
4. 每个合法的括号序列都可以由以上规则生成。

例如：“”, “()”, “()()”, “((()))”都是合法的括号序列 对于一个合法的括号序列我们又有以下定义它的深度：

1. 空串""的深度是0
2. 如果字符串“X”的深度是x,字符串“Y”的深度是y,那么字符串“XY”的深度为 $\max(x,y)$
3. 如果“X”的深度是x,那么字符串“(X)”的深度是x+1

例如: "()()"的深度是1,"((()))"的深度是3。牛牛现在给你一个合法的括号序列,需要你计算出其深度。

输入描述:

输入包括一个合法的括号序列s,s长度length( $2 \leq \text{length} \leq 50$ ),序列中只包含'('和')'。

输出描述:

输出一个正整数,即这个序列的深度。

示例:

输入:

(())

输出:

2

思路草图:

**从第一个字符开始向后遍历,碰到'(', count+1, 否则count-1。用Max保存, max=Math.max(max,count).max是上次循环的保存的最大值。**



代码如下:

```
import java.util.Scanner;

/**
 * https://www.nowcoder.com/test/8246651/summary
 *
 * @author Snailclimb
 * @date 2018年9月6日
 * @Description: TODO 求给定合法括号序列的深度
 */
public class Main {
 public static void main(String[] args) {
 Scanner sc = new Scanner(System.in);
 String s = sc.nextLine();
 int cnt = 0, max = 0, i;
 for (i = 0; i < s.length(); ++i) {
 if (s.charAt(i) == '(')
 cnt++;
 else
 cnt--;
 max = Math.max(max, cnt);
 }
 sc.close();
 System.out.println(max);
 }
}
```

## 6. 把字符串转换成整数

剑指offer: 将一个字符串转换成一个整数(实现Integer.valueOf(string)的功能，但是string不符合数字要求时返回0)，要求不能使用字符串转换整数的库函数。数值为0或者字符串不是一个合法的数值则返回0。

```
//https://www.weiweiblog.cn/strtoint/
public class Main {

 public static int StrToInt(String str) {
 if (str.length() == 0)
 return 0;
 char[] chars = str.toCharArray();
 // 判断是否存在符号位
 int flag = 0;
 if (chars[0] == '+')
 flag = 1;
 else if (chars[0] == '-')
 flag = 2;
 int start = flag > 0 ? 1 : 0;
 int res = 0;// 保存结果
 for (int i = start; i < chars.length; i++) {
 if (Character.isDigit(chars[i])) {// 调用Character.isDigit(char)方法判断是否是数字，是返回True，否则False
 int temp = chars[i] - '0';
 res = res * 10 + temp;
 } else {
 return 0;
 }
 }
 return flag == 1 ? res : -res;
 }

 public static void main(String[] args) {
 // TODO Auto-generated method stub
 String s = "-12312312";
 System.out.println("使用库函数转换: " + Integer.valueOf(s));
 int res = Main.StrToInt(s);
 System.out.println("使用自己写的方法转换: " + res);

 }
}
```

- 1. 两数相加
  - 题目描述
  - 问题分析
  - Solution
- 2. 翻转链表
  - 题目描述
  - 问题分析
  - Solution
- 3. 链表中倒数第k个节点
  - 题目描述
  - 问题分析
  - Solution
- 4. 删除链表的倒数第N个节点
  - 问题分析
  - Solution
- 5. 合并两个排序的链表
  - 题目描述
  - 问题分析
  - Solution

## 1. 两数相加

### 题目描述

Leetcode: 给定两个非空链表来表示两个非负整数。位数按照逆序方式存储，它们的每个节点只存储单个数字。将两数相加返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

示例：

```
输入: (2 -> 4 -> 3) + (5 -> 6 -> 4)
输出: 7 -> 0 -> 8
原因: 342 + 465 = 807
```

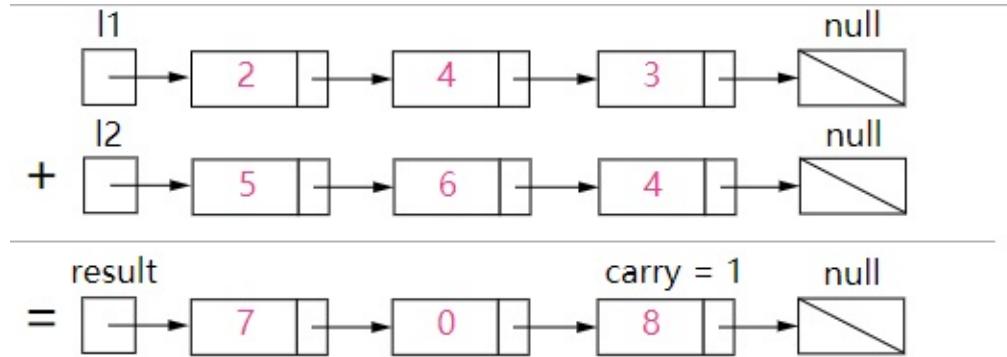
### 问题分析

Leetcode官方详细解答地址：

<https://leetcode-cn.com/problems/add-two-numbers/solution/>

要对头结点进行操作时，考虑创建哑节点dummy，使用dummy->next表示真正的头节点。这样可以避免处理头节点为空的边界问题。

我们使用变量来跟踪进位，并从包含最低有效位的表头开始模拟逐位相加的过程。



## Solution

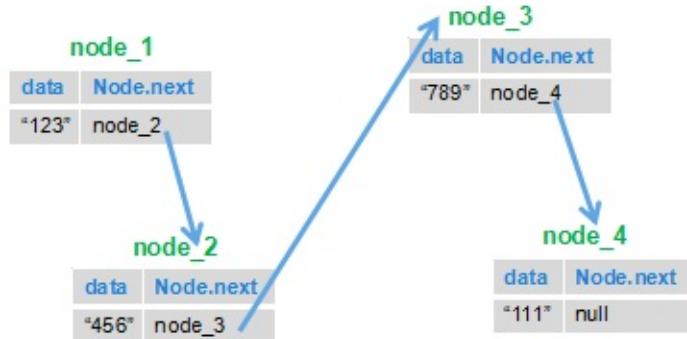
我们首先从最低有效位也就是列表  $l_1$  和  $l_2$  的表头开始相加。注意需要考虑到进位的情况！

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 * int val;
 * ListNode next;
 * ListNode(int x) { val = x; }
 * }
 //https://leetcode-cn.com/problems/add-two-numbers/description/
class Solution {
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
 ListNode dummyHead = new ListNode(0);
 ListNode p = l1, q = l2, curr = dummyHead;
 //carry 表示进位数
 int carry = 0;
 while (p != null || q != null) {
 int x = (p != null) ? p.val : 0;
 int y = (q != null) ? q.val : 0;
 int sum = carry + x + y;
 //进位数
 carry = sum / 10;
 //新节点的数值为sum % 10
 curr.next = new ListNode(sum % 10);
 curr = curr.next;
 if (p != null) p = p.next;
 if (q != null) q = q.next;
 }
 if (carry > 0) {
 curr.next = new ListNode(carry);
 }
 return dummyHead.next;
}
}
```

## 2. 翻转链表

### 题目描述

剑指 offer: 输入一个链表，反转链表后，输出链表的所有元素。



## 问题分析

这道算法题，说直白点就是：如何让后一个节点指向前一个节点！在下面的代码中定义了一个 `next` 节点，该节点主要是保存要反转到头的那个节点，防止链表“断裂”。

## Solution

```

public class ListNode {
 int val;
 ListNode next = null;

 ListNode(int val) {
 this.val = val;
 }
}

/**
 *
 * @author Snailclimb
 * @date 2018年9月19日
 * @Description: TODO
 */
public class Solution {

 public ListNode ReverseList(ListNode head) {
 ListNode next = null;
 ListNode pre = null;

 while (head != null) {
 // 保存要反转到头的那个节点
 next = head.next;
 // 要反转的那个节点指向已经反转的上一个节点(备注:第一次反转的时候会指向null)
 head.next = pre;
 // 上一个已经反转到头部的节点
 pre = head;
 // 一直向链表尾走
 head = next;
 }
 return pre;
 }
}

```

测试方法：

```

public static void main(String[] args) {

```

```

ListNode a = new ListNode(1);
ListNode b = new ListNode(2);
ListNode c = new ListNode(3);
ListNode d = new ListNode(4);
ListNode e = new ListNode(5);
a.next = b;
b.next = c;
c.next = d;
d.next = e;
new Solution().ReverseList(a);
while (e != null) {
 System.out.println(e.val);
 e = e.next;
}
}

```

输出：

```

5
4
3
2
1

```

## 3. 链表中倒数第k个节点

### 题目描述

剑指offer：输入一个链表，输出该链表中倒数第k个结点。

### 问题分析

链表中倒数第k个节点也就是正数第( $L-K+1$ )个节点，知道了这一点，这一题基本就没问题！

首先两个节点/指针，一个节点 node1 先开始跑，指针 node1 跑到  $k-1$  个节点后，另一个节点 node2 开始跑，当 node1 跑到最后时，node2 所指的节点就是倒数第k个节点也就是正数第( $L-K+1$ )个节点。

### Solution

```

/*
public class ListNode {
 int val;
 ListNode next = null;

 ListNode(int val) {
 this.val = val;
 }
} */

// 时间复杂度O(n)，一次遍历即可
// https://www.nowcoder.com/practice/529d3ae5a407492994ad2a246518148a?tpId=13&tqId=11167&tPage=1&rp=1&ru=/ta/co
ding-interviews&ru=/ta/coding-interviews/question-ranking
public class Solution {
 public ListNode FindKthToTail(ListNode head, int k) {
 // 如果链表为空或者k小于等于0
 if (head == null || k <= 0) {
 return null;
 }
 // 声明两个指向头结点的节点

```

```

ListNode node1 = head, node2 = head;
// 记录节点的个数
int count = 0;
// 记录k值，后面要使用
int index = k;
// p指针先跑，并且记录节点数，当node1节点跑了k-1个节点后，node2节点开始跑，
// 当node1节点跑到最后时，node2节点所指的节点就是倒数第k个节点
while (node1 != null) {
 node1 = node1.next;
 count++;
 if (k < 1 && node1 != null) {
 node2 = node2.next;
 }
 k--;
}
// 如果节点个数小于所求的倒数第k个节点，则返回空
if (count < index)
 return null;
return node2;
}
}

```

## 4. 删除链表的倒数第N个节点

Leetcode: 给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

示例：

给定一个链表：1->2->3->4->5，和 n = 2.

当删除了倒数第二个节点后，链表变为 1->2->3->5.

说明：

给定的 n 保证是有效的。

进阶：

你能尝试使用一趟扫描实现吗？

该题在 Leetcode 上有详细解答，具体可参考 Leetcode.

### 问题分析

我们注意到这个问题可以容易地简化成另一个问题：删除从列表开头数起的第  $(L - n + 1)$  个结点，其中 L 是列表的长度。只要我们找到列表的长度 L，这个问题就很容易解决。



## Solution

### 两次遍历法

首先我们将添加一个 **哑结点** 作为辅助，该结点位于列表头部。哑结点用来简化某些极端情况，例如列表中只含有一个结点，或需要删除列表的头部。在第一次遍历中，我们找出列表的长度  $L$ 。然后设置一个指向哑结点的指针，并移动它遍历列表，直至它到达第  $(L - n)$  个结点那里。我们把第  $(L - n)$  个结点的 `next` 指针重新链接至第  $(L - n + 2)$  个结点，完成这个算法。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 * int val;
 * ListNode next;
 * ListNode(int x) { val = x; }
 * }
 // https://leetcode-cn.com/problems/remove-nth-node-from-end-of-list/description/
public class Solution {
 public ListNode removeNthFromEnd(ListNode head, int n) {
 // 哑结点，哑结点用来简化某些极端情况，例如列表中只含有一个结点，或需要删除列表的头部
 ListNode dummy = new ListNode(0);
 // 哑结点指向头结点
 dummy.next = head;
 // 保存链表长度
 int length = 0;
 ListNode len = head;
 while (len != null) {
 length++;
 len = len.next;
 }
 length = length - n;
 ListNode target = dummy;
 // 找到 L-n 位置的节点
 while (length > 0) {
 target = target.next;
 length--;
 }
 // 把第 (L - n) 个结点的 next 指针重新链接至第 (L - n + 2) 个结点
 target.next = target.next.next;
 return dummy.next;
 }
}
```

### 复杂度分析：

- **时间复杂度  $O(L)$** ：该算法对列表进行了两次遍历，首先计算了列表的长度  $L$ ，其次找到第  $(L - n)$  个结点。操作执行了  $2L - n$  步，时间复杂度为  $O(L)$ 。

- 空间复杂度 O(1)：我们只用了常量级的额外空间。

### 进阶——一次遍历法：

\*\*链表中倒数第N个节点也就是正数第(L-N+1)个节点。

其实这种方法就和我们上面第四题找“链表中倒数第k个节点”所用的思想是一样的。基本思路就是：定义两个节点 node1、node2;node1 节点先跑，node1 节点跑到第 n+1 个节点的时候，node2 节点开始跑。当 node1 节点跑到最后一个节点时，node2 节点所在的位置就是第 (L-n) 个节点 (L 代表总链表长度，也就是倒数第 n+1 个节点)

```
/*
 * Definition for singly-linked list.
 * public class ListNode {
 * int val;
 * ListNode next;
 * ListNode(int x) { val = x; }
 * }
 */
public class Solution {
 public ListNode removeNthFromEnd(ListNode head, int n) {

 ListNode dummy = new ListNode(0);
 dummy.next = head;
 // 声明两个指向头结点的节点
 ListNode node1 = dummy, node2 = dummy;

 // node1 节点先跑，node1 节点跑到第 n 个节点的时候，node2 节点开始跑
 // 当 node1 节点跑到最后一个节点时，node2 节点所在的位置就是第 (L-n) 个节点，也就是倒数第 n+1 (L 代表总链表长度)
 while (node1 != null) {
 node1 = node1.next;
 if (n < 1 && node1 != null) {
 node2 = node2.next;
 }
 n--;
 }

 node2.next = node2.next.next;

 return dummy.next;
 }
}
```

## 5. 合并两个排序的链表

### 题目描述

剑指offer:输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

### 问题分析

我们可以这样分析：

1. 假设我们有两个链表 A,B；
2. A 的头节点 A1 的值与 B 的头结点 B1 的值比较，假设 A1 小，则 A1 为头节点；
3. A2 再和 B1 比较，假设 B1 小，则 A1 指向 B1；
4. A2 再和 B2 比较，就这样循环往复就行了，应该还算好理解。

考虑通过递归的方式实现！

## Solution

递归版本：

```
/*
public class ListNode {
 int val;
 ListNode next = null;

 ListNode(int val) {
 this.val = val;
 }
}*/
//https://www.nowcoder.com/practice/d8b6b4358f774294a89de2a6ac4d9337?tpId=13&tqId=11169&tPage=1&rp=1&ru=/ta/coding-interviews&qu=ta/coding-interviews/question-ranking
public class Solution {
 public ListNode Merge(ListNode list1,ListNode list2) {
 if(list1 == null){
 return list2;
 }
 if(list2 == null){
 return list1;
 }
 if(list1.val <= list2.val){
 list1.next = Merge(list1.next, list2);
 return list1;
 }else{
 list2.next = Merge(list1, list2.next);
 return list2;
 }
 }
}
```

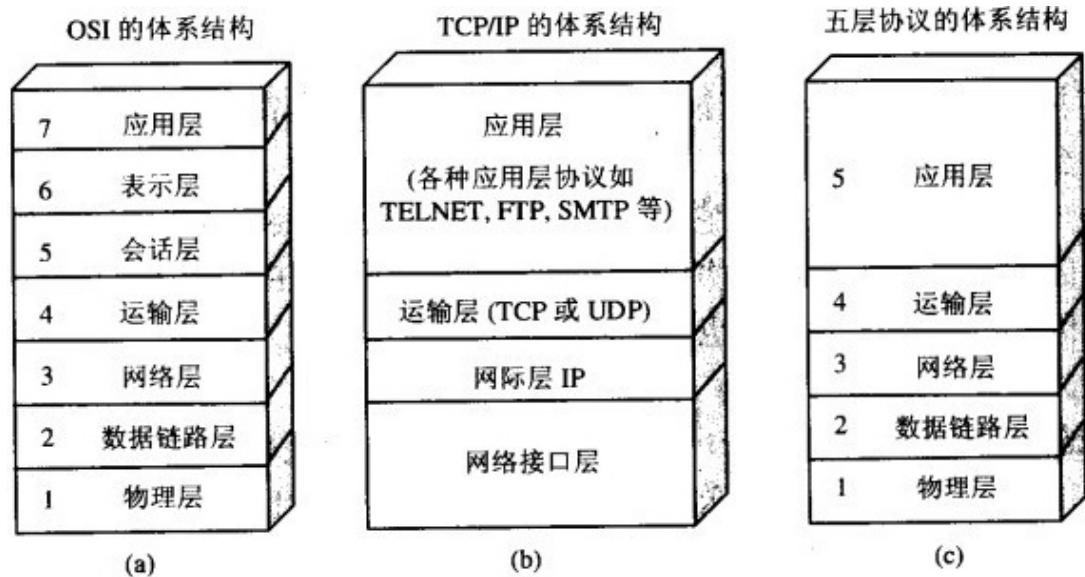
- 一 OSI与TCP/IP各层的结构与功能,都有哪些协议
  - 五层协议的体系结构
  - 1 应用层
    - 域名系统
    - HTTP协议
  - 2 运输层
    - 运输层主要使用以下两种协议
    - UDP 的主要特点
    - TCP 的主要特点
  - 3 网络层
  - 4 数据链路层
  - 5 物理层
  - 总结一下
- 二 TCP 三次握手和四次挥手(面试常客)
  - 为什么要三次握手
  - 为什么要传回 SYN
  - 传了 SYN,为啥还要传 ACK
  - 为什么要四次挥手
- 三 TCP、UDP 协议的区别
- 四 TCP 协议如何保证可靠传输
  - 停止等待协议
  - 自动重传请求 ARQ 协议
  - 连续ARQ协议
  - 滑动窗口
  - 流量控制
  - 拥塞控制
- 五 在浏览器中输入url地址 ->> 显示主页的过程 (面试常客)
- 六 状态码
- 七 各种协议与HTTP协议之间的关系
- 八 HTTP长连接、短连接
- 写在最后
  - 计算机网络常见问题回顾
  - 建议

相对与上一个版本的计算机网路面试知识总结，这个版本增加了“TCP协议如何保证可靠传输”包括超时重传、停止等待协议、滑动窗口、流量控制、拥塞控制等内容并且对一些已有内容做了补充。

## 一 OSI与TCP/IP各层的结构与功能,都有哪些协议

### 五层协议的体系结构

学习计算机网络时我们一般采用折中的办法，也就是中和 OSI 和 TCP/IP 的优点，采用一种只有五层协议的体系结构，这样既简洁又能将概念阐述清楚。



计算机网络体系结构: (a) OSI 的七层协议; (b) TCP/IP 的四层协议; (c) 五层协议

结合互联网的情况，自上而下地，非常简要的介绍一下各层的作用。

## 1 应用层

应用层(**application-layer**)的任务是通过应用进程间的交互来完成特定网络应用。应用层协议定义的是应用进程（进程：主机中正在运行的程序）间的通信和交互的规则。对于不同的网络应用需要不同的应用层协议。在互联网中应用层协议很多，如域名系统**DNS**，支持万维网应用的**HTTP协议**，支持电子邮件的**SMTP协议**等等。我们把应用层交互的数据单元称为报文。

## 域名系统

域名系统(Domain Name System缩写 DNS, Domain Name被译为域名)是因特网的一项核心服务，它作为可以将域名和IP地址相互映射的一个分布式数据库，能够使人更方便的访问互联网，而不用去记住能够被机器直接读取的IP数串。（百度百科）例如：一个公司的 Web 网站可看作是它在网上的门户，而域名就相当于其门牌地址，通常域名都使用该公司的名称或简称。例如上面提到的微软公司的域名，类似的还有：IBM 公司的域名是 www.ibm.com、Oracle 公司的域名是 www.oracle.com、Cisco公司的域名是 www.cisco.com 等。

## HTTP协议

超文本传输协议 (HTTP, HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络协议。所有的 WWW (万维网) 文件都必须遵守这个标准。设计 HTTP 最初的目的是为了提供一种发布和接收 HTML 页面的方法。（百度百科）

## 2 运输层

运输层(**transport layer**)的主要任务就是负责向两台主机进程之间的通信提供通用的数据传输服务。应用进程利用该服务传送应用层报文。“通用的”是指并不针对某一个特定的网络应用，而是多种应用可以使用同一个运输层服务。由于一台主机可同时运行多个线程，因此运输层有复用和分用的功能。所谓复用就是指多个应用层进程可同时使用下面运输层的服务，分用和复用相反，是运输层把收到的信息分别交付上面应用层中的相应进程。

## 运输层主要使用以下两种协议

1. 传输控制协议 **TCP** (Transmisson Control Protocol) --提供面向连接的，可靠的数据传输服务。

2. 用户数据协议 **UDP** (User Datagram Protocol) --提供无连接的，尽最大努力的数据传输服务（不保证数据传输的可靠性）。

## UDP 的主要特点

1. UDP 是无连接的；
2. UDP 使用尽最大努力交付，即不保证可靠交付，因此主机不需要维持复杂的链接状态（这里面有许多参数）；
3. UDP 是面向报文的；
4. UDP 没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如 直播，实时视频会议等）；
5. UDP 支持一对一、一对多、多对一和多对多的交互通信；
6. UDP 的首部开销小，只有8个字节，比TCP的20个字节的首部要短。

## TCP 的主要特点

1. TCP 是面向连接的。（就好像打电话一样，通话前需要先拨号建立连接，通话结束后要挂机释放连接）；
2. 每一条 TCP 连接只能有两个端点，每一条TCP连接只能是点对点的（一对一）；
3. TCP 提供可靠交付的服务。通过TCP连接传送的数据，无差错、不丢失、不重复、并且按序到达；
4. TCP 提供全双工通信。TCP 允许通信双方的应用进程在任何时候都能发送数据。TCP 连接的两端都设有发送缓存和接收缓存，用来临时存放双方通信的数据；
5. 面向字节流。TCP 中的“流”（Stream）指的是流入进程或从进程流出的字节序列。“面向字节流”的含义是：虽然应用程序和 TCP 的交互是一次一个数据块（大小不等），但 TCP 把应用程序交下来的数据仅仅看成是一连串的无结构的字节流。

## 3 网络层

在 计算机网络中进行通信的两个计算机之间可能会经过很多个数据链路，也可能还要经过很多通信子网。网络层的任务就是选择合适的网间路由和交换结点，确保数据及时传送。在发送数据时，网络层把运输层产生的报文段或用户数据报封装成分组和包进行传送。在 TCP/IP 体系结构中，由于网络层使用 **IP 协议**，因此分组也叫 **IP 数据报**，简称 **数据报**。

这里要注意：不要把运输层的“用户数据报 **UDP** ”和网络层的“ **IP 数据报**”弄混。另外，无论是哪一层的数据单元，都可笼统地用“分组”来表示。

这里强调指出，网络层中的“网络”二字已经不是我们通常谈到的具体网络，而是指计算机网络体系结构模型中第三层的名称。

互联网是由大量的异构 (heterogeneous) 网络通过路由器 (router) 相互连接起来的。互联网使用的网络层协议是无连接的网际协议 (Internet Protocol) 和许多路由选择协议，因此互联网的网络层也叫做**网际层**或**IP层**。

## 4 数据链路层

**数据链路层(data link layer)**通常简称为**链路层**。两台主机之间的数据传输，总是在一段一段的链路上传送的，这就需要使用专门的**链路层的协议**。在两个相邻节点之间传送数据时，**数据链路层**将**网络层交下来的 IP 数据报**组装成帧，在两个相邻节点间的链路上传送帧。每一帧包括数据和必要的控制信息（如同步信息，地址信息，差错控制等）。

在接收数据时，控制信息使接收端能够知道一个帧从哪个比特开始和到哪个比特结束。这样，**数据链路层**在收到一个帧后，就可从中提出数据部分，上交给**网络层**。控制信息还使接收端能够检测到所收到的帧中有误差错。如果发现差错，**数据链路层**就简单地丢弃这个出了差错的帧，以避免继续在网络中传送下去白白浪费**网络资源**。如果需要改正数据在**链路层**传输时出现差错（这就是说，**数据链路层**不仅要检错，而且还要纠错），那么就要采用**可靠性传输协议**来纠正出现的差错。这种方法会使**链路层的协议**复杂些。

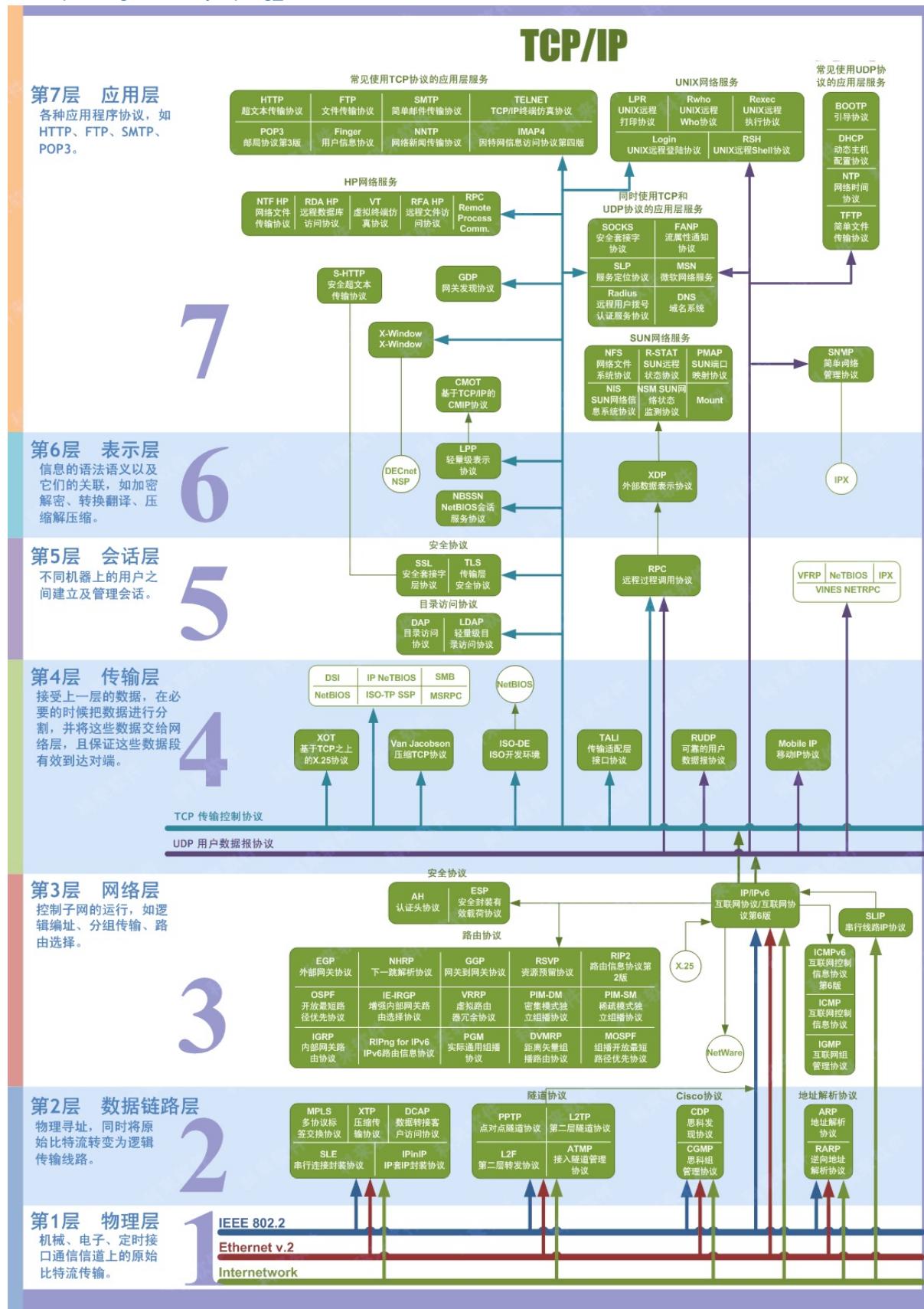
## 5 物理层

在物理层上所传送的数据单位是比特。物理层(**physical layer**)的作用是实现相邻计算机节点之间比特流的透明传送，尽可能屏蔽掉具体传输介质和物理设备的差异。使其上面的数据链路层不必考虑网络的具体传输介质是什么。“透明传送比特流”表示经实际电路传送后的比特流没有发生变化，对传送的比特流来说，这个电路好像是看不见的。

在互联网使用的各种协议中最重要和最著名的就是 TCP/IP 两个协议。现在人们经常提到的TCP/IP并不一定单指TCP和IP这两个具体的协议，而往往表示互联网所使用的整个TCP/IP协议族。

## 总结一下

上面我们对计算机网络的五层体系结构有了初步的了解，下面附送一张七层体系结构图总结一下。图片来源：[https://blog.csdn.net/yaopeng\\_2005/article/details/7064869](https://blog.csdn.net/yaopeng_2005/article/details/7064869)

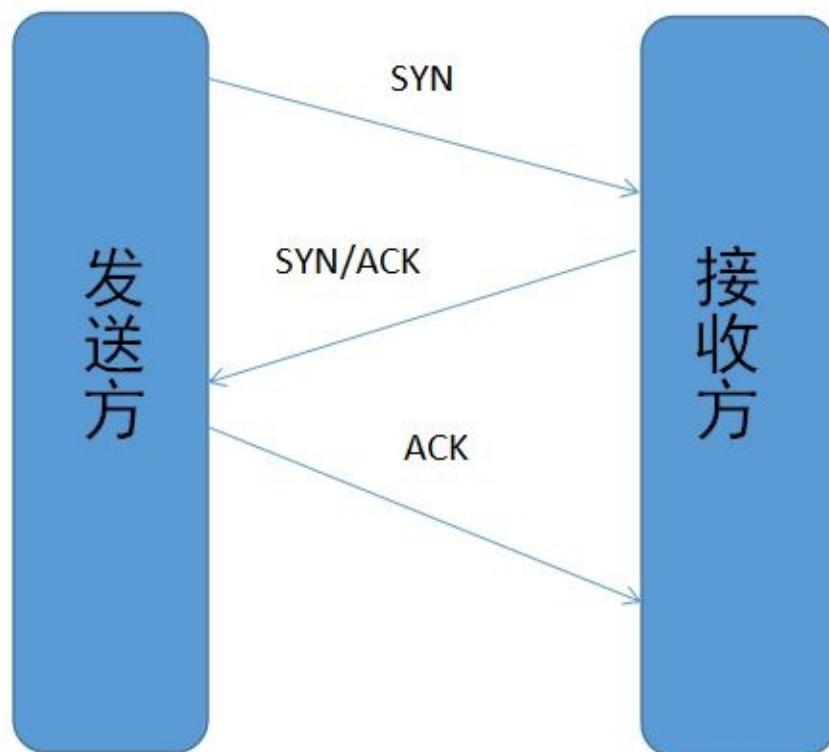
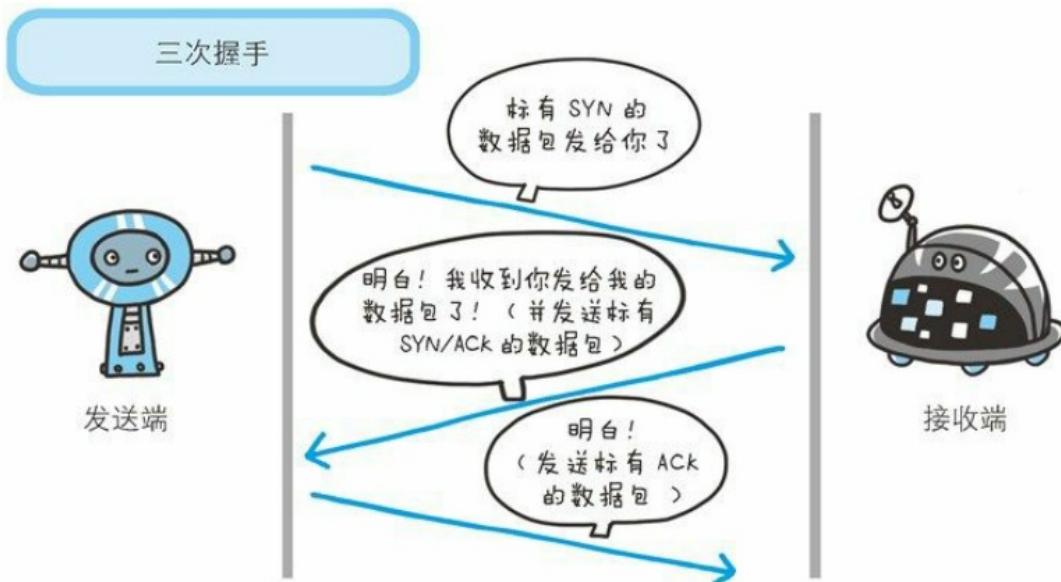


## 二 TCP 三次握手和四次挥手(面试常客)

为了准确无误地把数据送达目标处，TCP协议采用了三次握手策略。

漫画图解：

图片来源：《图解HTTP》



简单示意图：

- 客户端-发送带有 SYN 标志的数据包-一次握手-服务端
- 服务端-发送带有 SYN/ACK 标志的数据包-二次握手-客户端
- 客户端-发送带有带有 ACK 标志的数据包-三次握手-服务端

## 为什么要三次握手

三次握手的目的是建立可靠的通信信道，说到通讯，简单来说就是数据的发送与接收，而三次握手最主要的目的就是双方确认自己与对方的发送与接收是正常的。

第一次握手：Client 什么都不能确认；Server 确认了对方发送正常

第二次握手：Client 确认了：自己发送、接收正常，对方发送、接收正常；Server 确认了：自己接收正常，对方发送正常

第三次握手：Client 确认了：自己发送、接收正常，对方发送、接收正常；Server 确认了：自己发送、接收正常，对方发送接收正常

所以三次握手就能确认双发收发功能都正常，缺一不可。

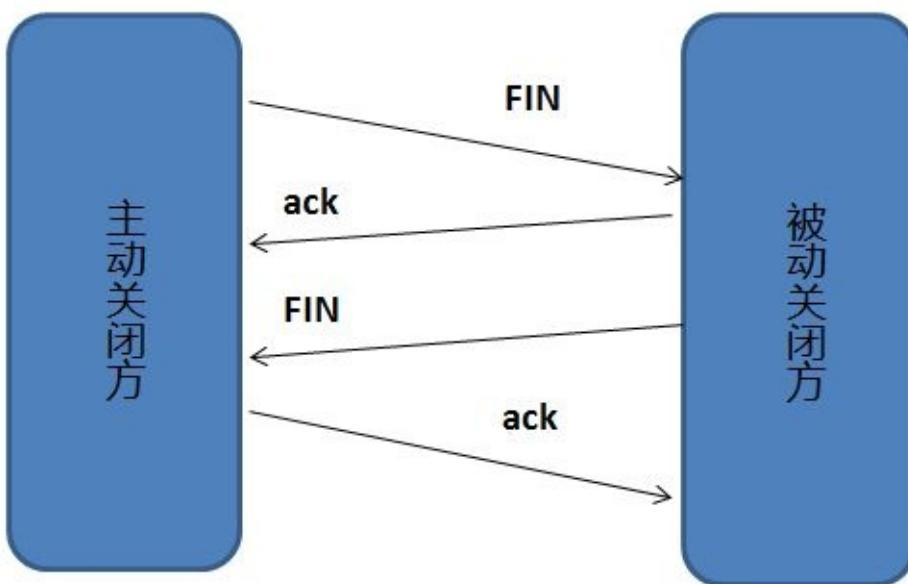
## 为什么要传回 SYN

接收端传回发送端所发送的 SYN 是为了告诉发送端，我接收到的信息确实就是你所发送的信号了。

SYN 是 TCP/IP 建立连接时使用的握手信号。在客户机和服务器之间建立正常的 TCP 网络连接时，客户机首先发出一个 SYN 消息，服务器使用 SYN-ACK 应答表示接收到了这个消息，最后客户机再以 ACK(Acknowledgement[汉译：确认字符，在数据通信传输中，接收站发给发送站的一种传输控制字符。它表示确认发来的数据已经接受无误。]) 消息响应。这样在客户机和服务器之间才能建立起可靠的TCP连接，数据才可以在客户机和服务器之间传递。

## 传了 SYN,为啥还要传 ACK

双方通信无误必须是两者互相发送信息都无误。传了 SYN，证明发送方到接收方的通道没有问题，但是接收方到发送方的通道还需要 ACK 信号来进行验证。



断开一个 TCP 连接则需要“四次挥手”：

- 客户端-发送一个 FIN，用来关闭客户端到服务器的数据传送
- 服务器-收到这个 FIN，它发回一个 ACK，确认序号为收到的序号加1。和 SYN一样，一个 FIN 将占用一个序号
- 服务器-关闭与客户端的连接，发送一个FIN给客户端
- 客户端-发回 ACK 报文确认，并将确认序号设置为收到序号加1

## 为什么要四次挥手

任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。当另一方也没有数据再发送的时候，则发出连接释放通知，对方确认后就完全关闭了TCP连接。

举个例子：A 和 B 打电话，通话即将结束后，A 说“我没啥要说的了”，B回答“我知道了”，但是 B 可能还会有要说的话，A 不能要求 B 跟着自己的节奏结束通话，于是 B 可能又巴拉巴拉说了一通，最后 B 说“我说完了”，A 回答“知道了”，这样通话才算结束。

上面讲的比较概括，推荐一篇讲的比较细致的文章：<https://blog.csdn.net/qzcsu/article/details/72861891>

## 三 TCP、UDP 协议的区别

| 类型  | 特点     |       |       | 性能   |      | 应用场景                     | 首部字节              |
|-----|--------|-------|-------|------|------|--------------------------|-------------------|
|     | 是否面向连接 | 传输可靠性 | 传输形式  | 传输效率 | 所需资源 |                          |                   |
| TCP | 面向连接   | 可靠    | 字节流   | 慢    | 多    | 要求通信数据可靠<br>(如文件传输、邮件传输) | 20-60             |
| UDP | 无连接    | 不可靠   | 数据报文段 | 快    | 少    | 要求通信速度高<br>(如域名转换)       | 8个字节<br>(由4个字段组成) |

UDP 在传送数据之前不需要先建立连接，远地主机在收到 UDP 报文后，不需要给出任何确认。虽然 UDP 不提供可靠交付，但在某些情况下 UDP 确是一种最有效的工作方式（一般用于即时通信），比如：QQ 语音、QQ 视频、直播等等

TCP 提供面向连接的服务。在传送数据之前必须先建立连接，数据传送结束后要释放连接。TCP 不提供广播或多播服务。由于 TCP 要提供可靠的，面向连接的运输服务（TCP的可靠体现在TCP在传递数据之前，会有三次握手来建立连接，而且在数据传递时，有确认、窗口、重传、拥塞控制机制，在数据传完后，还会断开连接用来节约系统资源），这一难以避免增加了许多开销，如确认，流量控制，计时器以及连接管理等。这不仅使协议数据单元的首部增大很多，还要占用许多处理器资源。TCP 一般用于文件传输、发送和接收邮件、远程登录等场景。

## 四 TCP 协议如何保证可靠传输

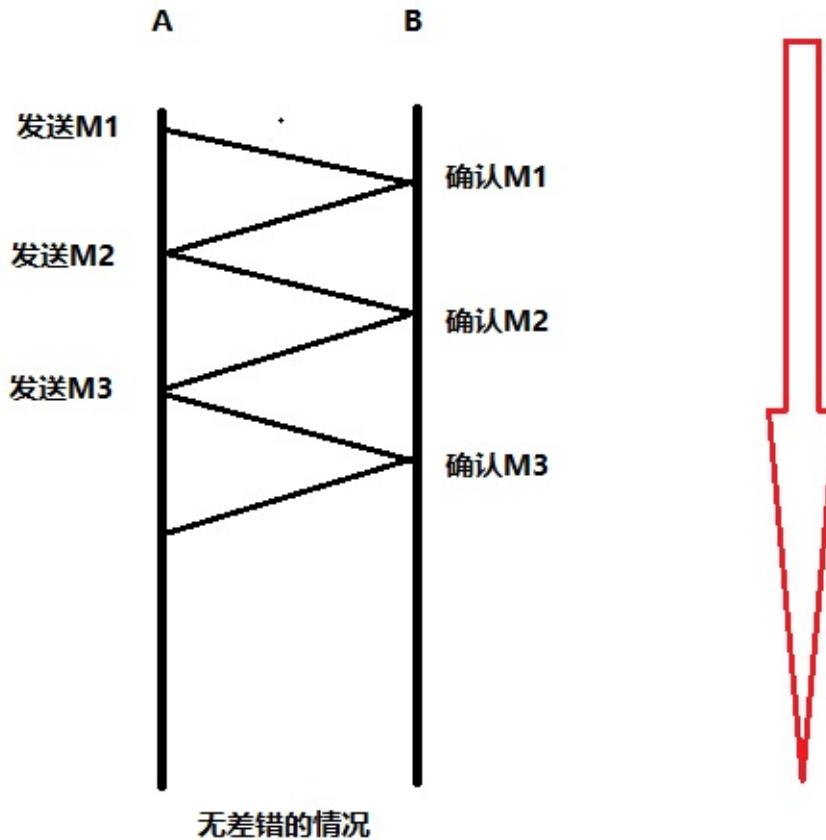
- 应用数据被分割成 TCP 认为最适合发送的数据块。
- TCP 给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。
- 校验和：** TCP 将保持它首部和数据的校验和。这是一个端到端的校验和，目的是检测数据在传输过程中的任何变化。如果收到段的校验和有差错，TCP 将丢弃这个报文段和不确认收到此报文段。
- TCP 的接收端会丢弃重复的数据。
- 流量控制：** TCP 连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP 使用的流量控制协议是可变大小的滑动窗口协议。（TCP 利用滑动窗口实现流量控制）
- 拥塞控制：** 当网络拥塞时，减少数据的发送。
- 停止等待协议** 也是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组。**超时重传：** 当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

## 停止等待协议

- 停止等待协议是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组；

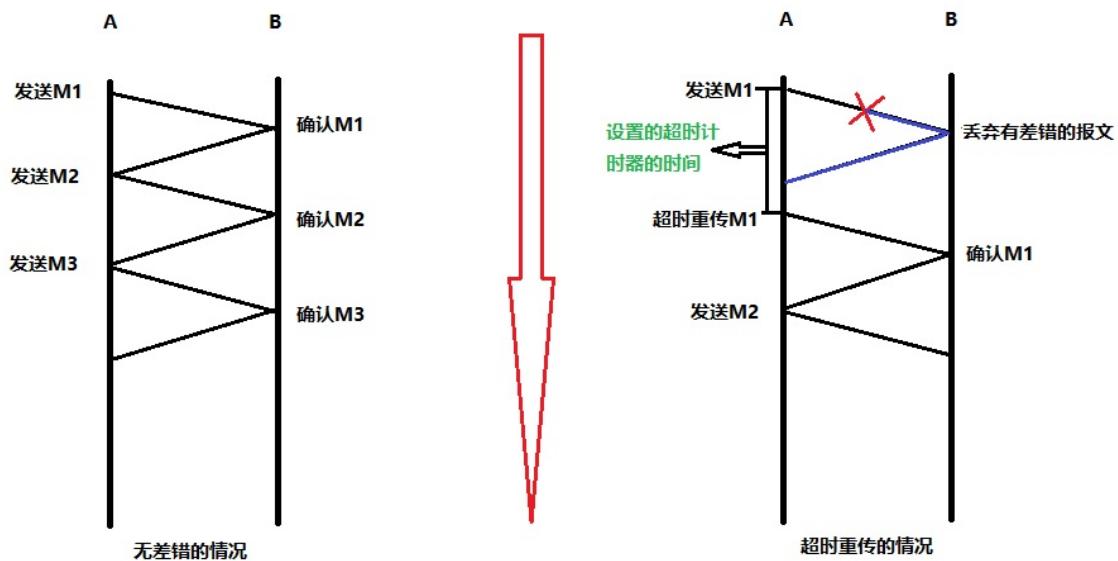
- 在停止等待协议中，若接收方收到重复分组，就丢弃该分组，但同时还要发送确认；

### 1) 无差错情况：



发送方发送分组,接收方在规定时间内收到,并且回复确认.发送方再次发送。

### 2) 出现差错情况（超时重传）：

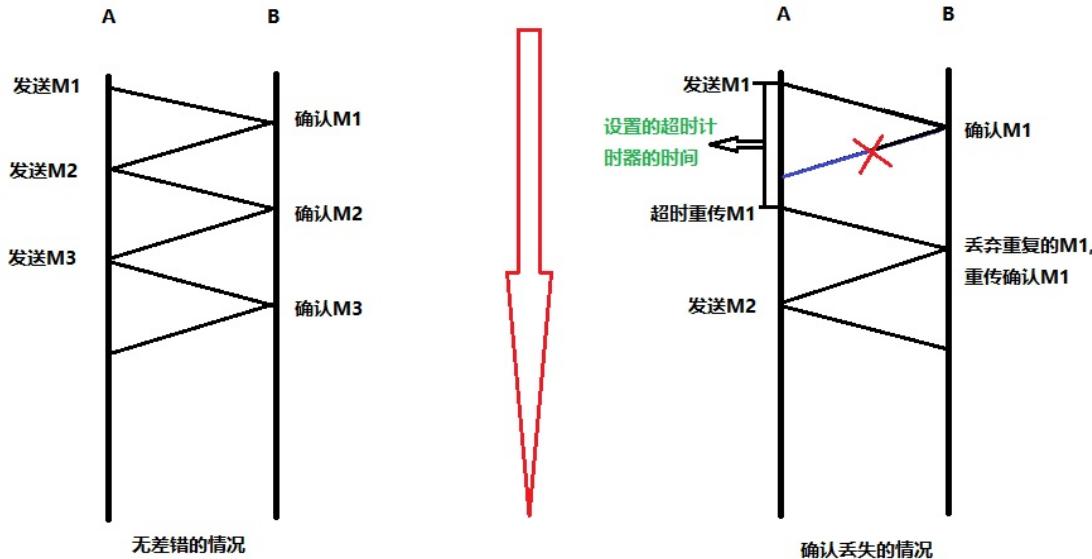


停止等待协议中超时重传是指只要超过一段时间仍然没有收到确认，就重传前面发送过的分组（认为刚才发送过的分组丢失了）。因此每发送完一个分组需要设置一个超时计时器，其重传时间应比数据在分组传输的平均往返时间更长一些。这种自动重传方式常称为 **自动重传请求 ARQ**。另外在停止等待协议中若收到重复分组，就丢弃该分组，但同时还

要发送确认。连续 ARQ 协议 可提高信道利用率。发送维持一个发送窗口，凡位于发送窗口内的分组可连续发送出去，而不需要等待对方确认。接收方一般采用累积确认，对按序到达的最后一个分组发送确认，表明到这个分组位置的所有分组都已经正确收到了。

### 3) 确认丢失和确认迟到

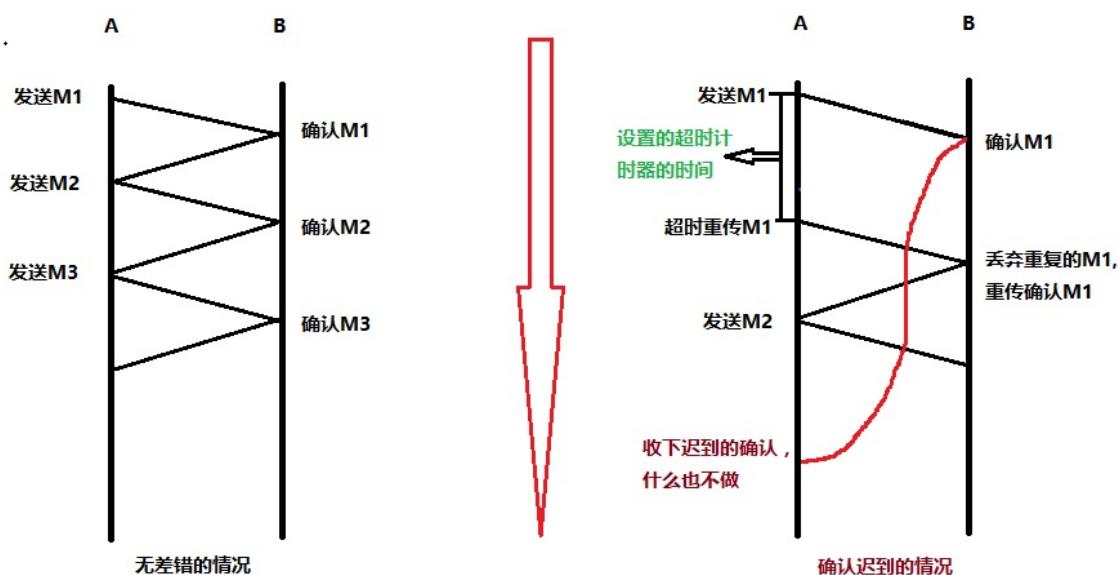
- 确认丢失：确认消息在传输过程丢失



当A发送M1消息，B收到后，B向A发送了一个M1确认消息，但却在传输过程中丢失。而A并不知道，在超时计时过后，A重传M1消息，B再次收到该消息后采取以下两点措施：

1. 丢弃这个重复的M1消息，不向上层交付。
2. 向A发送确认消息。（不会认为已经发送过了，就不再发送。A能重传，就证明B的确认消息丢失）。

- 确认迟到：确认消息在传输过程中迟到



A发送M1消息，B收到并发送确认。在超时时间内没有收到确认消息，A重传M1消息，B仍然收到并继续发送确认消息（B收到了2份M1）。此时A收到了B第二次发送的确认消息。接着发送其他数据。过了一会，A收到了B第一次发送的对M1的确认消息（A也收到了2份确认消息）。处理如下：

1. A收到重复的确认后，直接丢弃。
2. B收到重复的M1后，也直接丢弃重复的M1。

## 自动重传请求 ARQ 协议

停止等待协议中超时重传是指只要超过一段时间仍然没有收到确认，就重传前面发送过的分组（认为刚才发送过的分组丢失了）。因此每发送完一个分组需要设置一个超时计时器，其重传时间应比数据在分组传输的平均往返时间更长一些。这种自动重传方式常称为自动重传请求ARQ。

**优点：**简单

**缺点：**信道利用率低

## 连续ARQ协议

连续 ARQ 协议可提高信道利用率。发送方维持一个发送窗口，凡位于发送窗口内的分组可以连续发送出去，而不需要等待对方确认。接收方一般采用累计确认，对按序到达的最后一个分组发送确认，表明到这个分组为止的所有分组都已经正确收到了。

**优点：**信道利用率高，容易实现，即使确认丢失，也不必重传。

**缺点：**不能向发送方反映出接收方已经正确收到的所有分组的信息。比如：发送方发送了 5 条消息，中间第三条丢失（3 号），这时接收方只能对前两个发送确认。发送方无法知道后三个分组的下落，而只好把后三个全部重传一次。这也叫 Go-Back-N（回退 N），表示需要退回来重传已经发送过的 N 个消息。

## 滑动窗口

- TCP 利用滑动窗口实现流量控制的机制。
- 滑动窗口（Sliding window）是一种流量控制技术。早期的网络通信中，通信双方不会考虑网络的拥挤情况直接发送数据。由于大家不知道网络拥塞状况，同时发送数据，导致中间节点阻塞掉包，谁也发不了数据，所以就有了滑动窗口机制来解决此问题。
- TCP 中采用滑动窗口来进行传输控制，滑动窗口的大小意味着接收方还有多大的缓冲区可以用于接收数据。发送方可以通过滑动窗口的大小来确定应该发送多少字节的数据。当滑动窗口为 0 时，发送方一般不能再发送数据报，但有两种情况除外，一种情况是可以发送紧急数据，例如，允许用户终止在远端机上的运行进程。另一种情况是发送方可以发送一个 1 字节的数据报来通知接收方重新声明它希望接收的下一字节及发送方的滑动窗口大小。

## 流量控制

- TCP 利用滑动窗口实现流量控制。
- 流量控制是为了控制发送方发送速率，保证接收方来得及接收。
- 接收方发送的确认报文中的窗口字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为 0，则发送方不能发送数据。

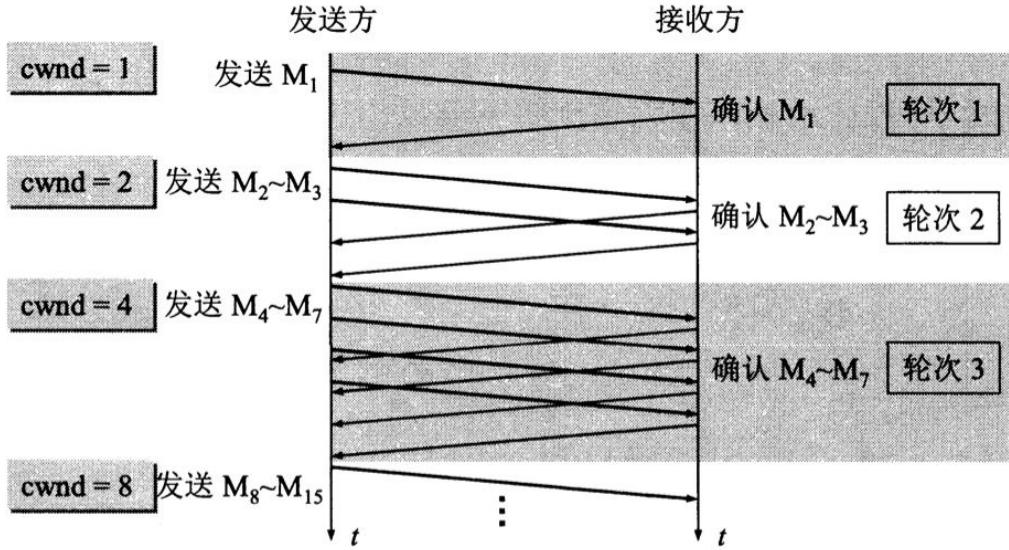
## 拥塞控制

在某段时间，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。这种情况就叫拥塞。拥塞控制就是为了防止过多的数据注入到网络中，这样就可以使网络中的路由器或链路不致过载。拥塞控制所要做的都有一个前提，就是网络能够承受现有的网络负荷。拥塞控制是一个全局性的过程，涉及到所有的主机，所有的路由器，以及与降低网络传输性能有关的所有因素。相反，流量控制往往是点对点通信量的控制，是个端到端的问题。流量控制所要做的就是抑制发送端发送数据的速率，以便使接收端来得及接收。

为了进行拥塞控制，TCP 发送方要维持一个 **拥塞窗口(cwnd)** 的状态变量。拥塞控制窗口的大小取决于网络的拥塞程度，并且动态变化。发送方让自己的发送窗口取为拥塞窗口和接收方的接受窗口中较小的一个。

TCP 的拥塞控制采用了四种算法，即 **慢开始**、**拥塞避免**、**快重传** 和 **快恢复**。在网络层也可以使路由器采用适当的分组丢弃策略（如主动队列管理 AQM），以减少网络拥塞的发生。

- **慢开始：**慢开始算法的思路是当主机开始发送数据时，如果立即把大量数据字节注入到网络，那么可能会引起网络阻塞，因为现在还不知道网络的符合情况。经验表明，较好的方法是先探测一下，即由小到大逐渐增大发送窗口，也就是由小到大逐渐增大拥塞窗口数值。cwnd 初始值为 1，每经过一个传播轮次，cwnd 加倍。



- **拥塞避免：** 拥塞避免算法的思路是让拥塞窗口cwnd缓慢增大，即每经过一个往返时间RTT就把发送放的cwnd加1。
- **快重传与快恢复：** 在TCP/IP中，快速重传和恢复（fast retransmit and recovery, FRR）是一种拥塞控制算法，它能快速恢复丢失的数据包。没有FRR，如果数据包丢失了，TCP将会使用定时器来要求传输暂停。在暂停的这段时间内，没有新的或复制的数据包被发送。有了FRR，如果接收机接收到一个不按顺序的数据段，它会立即给发送机发送一个重复确认。如果发送机接收到三个重复确认，它会假定确认件指出的数据段丢失了，并立即重传这些丢失的数据段。有了FRR，就不会因为重传时要求的暂停被耽误。当有单独的数据包丢失时，快速重传和恢复（FRR）能最有效地工作。当有多个数据信息包在某一段很短的时间内丢失时，它则不能很有效地工作。

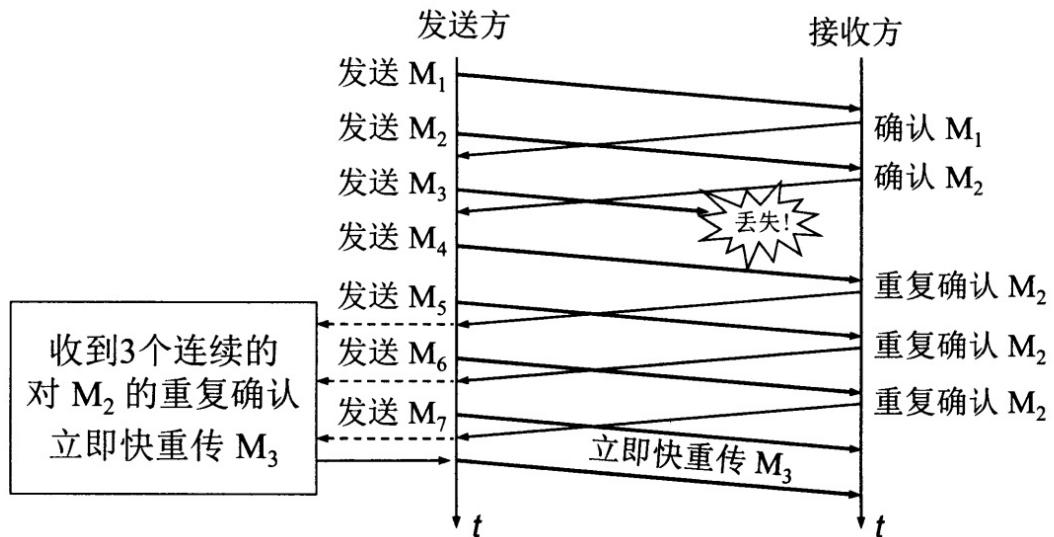


图 5-26 快重传的示意图

## 五 在浏览器中输入url地址 -> 显示主页的过程 (面试常客)

百度好像最喜欢问这个问题。

打开一个网页，整个过程会使用哪些协议

图解（图片来源：《图解HTTP》）：

| 过程                                                | 使用的协议                                                                                                                                                                                                                                                            |
|---------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. 浏览器查找域名的IP地址<br>(DNS查找过程: 浏览器缓存、路由器缓存、DNS 缓存)  | DNS: 获取域名对应IP                                                                                                                                                                                                                                                    |
| 2. 浏览器向web服务器发送一个HTTP请求<br>(cookies会随着请求发送给服务器)   |                                                                                                                                                                                                                                                                  |
| 3. 服务器处理请求<br>(请求 处理请求 & 它的参数、cookies、生成一个HTML响应) | <ul style="list-style-type: none"> <li>• TCP: 与服务器建立TCP连接</li> <li>• IP: 建立TCP协议时, 需要发送数据, 发送数据在网络层使用IP协议</li> <li>• OPSF: IP数据包在路由器之间, 路由选择使用OPSF协议</li> <li>• ARP: 路由器在与服务器通信时, 需要将ip地址转换为MAC地址, 需要使用ARP协议</li> <li>• HTTP: 在TCP建立完成后, 使用HTTP协议访问网页</li> </ul> |
| 4. 服务器发回一个HTML响应                                  |                                                                                                                                                                                                                                                                  |
| 5. 浏览器开始显示HTML                                    |                                                                                                                                                                                                                                                                  |

总体来说分为以下几个过程:

1. DNS解析
2. TCP连接
3. 发送HTTP请求
4. 服务器处理请求并返回HTTP报文
5. 浏览器解析渲染页面
6. 连接结束

具体可以参考下面这篇文章:

- <https://segmentfault.com/a/1190000006879700>

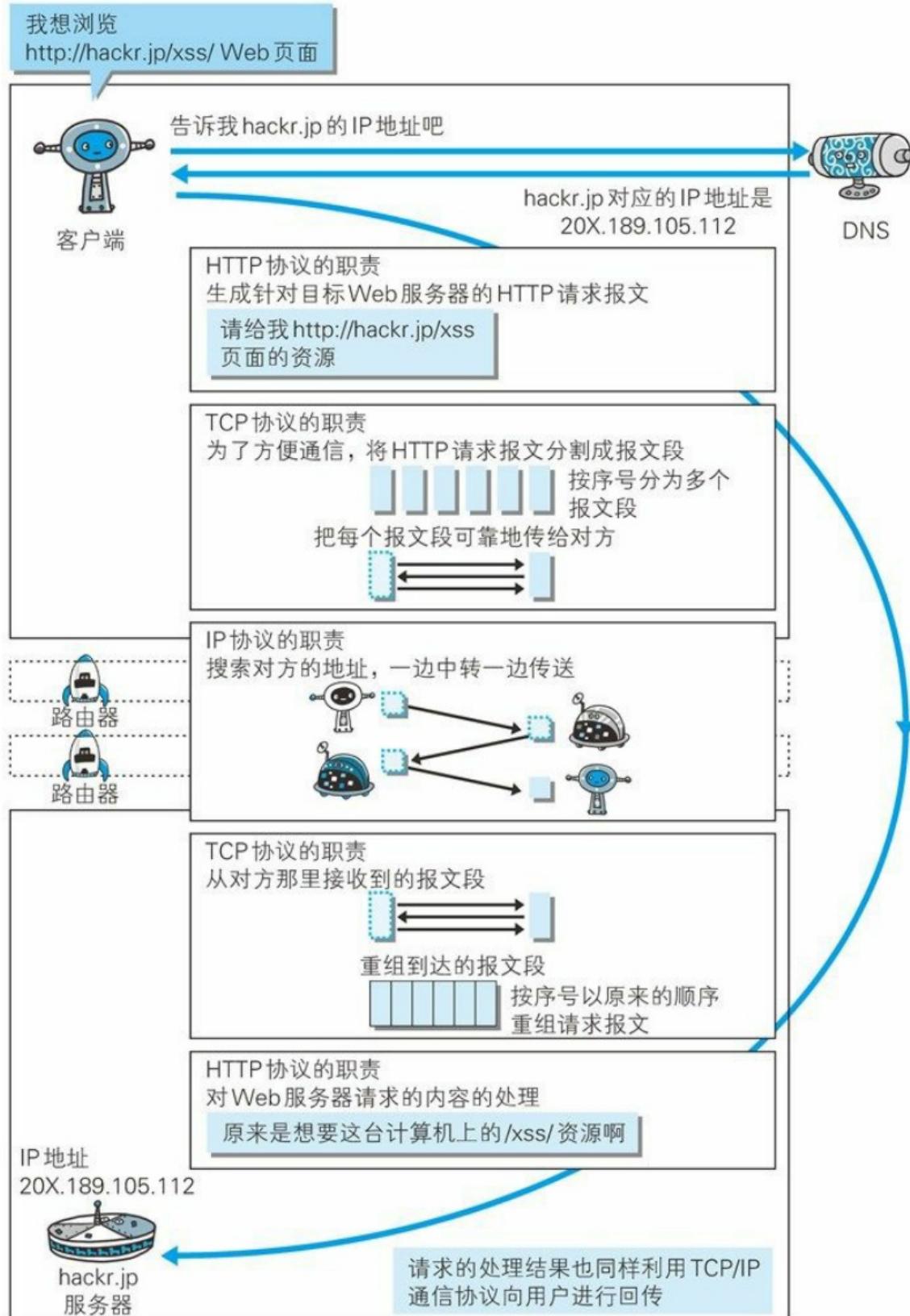
## 六 状态码

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

## 七 各种协议与HTTP协议之间的关系

一般面试官会通过这样的问题来考察你对计算机网络知识体系的理解。

图片来源：《图解HTTP》



## 八 HTTP长连接、短连接

在HTTP/1.0中默认使用短连接。也就是说，客户端和服务器每进行一次HTTP操作，就建立一次连接，任务结束就中断连接。当客户端浏览器访问的某个HTML或其他类型的Web页中包含有其他的Web资源（如JavaScript文件、图像文件、CSS文件等），每遇到这样一个Web资源，浏览器就会重新建立一个HTTP会话。

而从HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头加入这行代码：

```
Connection:keep-alive
```

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接需要客户端和服务端都支持长连接。

**HTTP协议的长连接和短连接，实质上是TCP协议的长连接和短连接。**

—— [《HTTP长连接、短连接究竟是什么？》](#)

## 写在最后

### 计算机网络常见问题回顾

- ①TCP三次握手和四次挥手、
- ②在浏览器中输入url地址->>显示主页的过程
- ③HTTP和HTTPS的区别
- ④TCP、UDP协议的区别
- ⑤常见的状态码。

### 建议

非常推荐大家看一下《图解HTTP》这本书，这本书页数不多，但是内容很是充实，不管是用来系统的掌握网络方面的一些知识还是说纯粹为了应付面试都有很大帮助。下面的一些文章只是参考。大二学习这门课程的时候，我们使用的教材是《计算机网络第七版》（谢希仁编著），不推荐大家看这本教材，书非常厚而且知识偏理论，不确定大家能不能心平气和的读完。

### 参考

- [https://blog.csdn.net/qq\\_16209077/article/details/52718250](https://blog.csdn.net/qq_16209077/article/details/52718250)
- <https://blog.csdn.net/zixiaomuwu/article/details/60965466>
- [https://blog.csdn.net/turn\\_back/article/details/73743641](https://blog.csdn.net/turn_back/article/details/73743641)

## 目录结构

1. 计算机概述

2. 物理层

3. 数据链路层

4. 网络层

5. 运输层

6. 应用层

## 一计算机概述

(1) , 基本术语

结点 (node) :

网络中的结点可以是计算机, 集线器, 交换机或路由器等。

链路 (link) :

从一个结点到另一个结点的一段物理线路。中间没有任何其他交点。

主机 (host) :

连接在因特网上的计算机。

ISP (Internet Service Provider) :

因特网服务提供者 (提供商) .

IXP (Internet eXchange Point) :

互联网交换点IXP的主要作用就是允许两个网络直接相连并交换分组, 而不需要再通过第三个网络来转发分组。.

RFC(Request For Comments)

意思是“请求评议”, 包含了关于Internet几乎所有的重要的文字资料。

广域网WAN (Wide Area Network)

任务是通过长距离运送主机发送的数据

## 城域网MAN (Metropolitan Area Network)

用来讲多个局域网进行互连

## 局域网LAN (Local Area Network)

学校或企业大多拥有多个互连的局域网

## 个人区域网PAN (Personal Area Network)

在个人工作的地方把属于个人使用的电子设备用无线技术连接起来的网络

### 端系统 (end system) :

处在因特网边缘的部分即是连接在因特网上的所有的主机.

### 分组 (packet) :

因特网中传送的数据单元。由首部header和数据段组成。分组又称为包，首部可称为包头。

### 存储转发 (store and forward) :

路由器收到一个分组，先存储下来，再检查其首部，查找转发表，按照首部中的目的地址，找到合适的接口转发出去。

### 带宽 (bandwidth) :

在计算机网络中，表示在单位时间内从网络中的某一点到另一点所能通过的“最高数据率”。常用来表示网络的通信线路所能传送数据的能力。单位是“比特每秒”，记为b/s。

### 吞吐量 (throughput) :

表示在单位时间内通过某个网络（或信道、接口）的数据量。吞吐量更经常地用于对现实世界中的网络的一种测量，以便知道实际上到底有多少数据量能够通过网络。吞吐量受网络的带宽或网络的额定速率的限制。

## (2) , 重要知识点总结

- 1, 计算机网络（简称网络）把许多计算机连接在一起，而互联网把许多网络连接在一起，是网络的网络。
- 2, 小写字母i开头的internet（互联网）是通用名词，它泛指由多个计算机网络相互连接而成的网络。在这些网络之间的通信协议（即通信规则）可以是任意的。  
大写字母I开头的Internet（互联网）是专用名词，它指全球最大的，开放的，由众多网络相互连接而成的特定的互联网，并采用TCP/IP协议作为通信规则，其前身为ARPANET。Internet的推荐译名为因特网，现在一般流行称为互联网。
- 3, 路由器是实现分组交换的关键构件，其任务是转发受到的分组，这是网络核心部分最重要的功能。分组交换采用存

储转发技术，表示把一个报文（要发送的整块数据）分为几个分组后在进行传送。在发送报文之前，先把较长的报文划分成为一个个更小的等长数据段。在每个数据端的前面加上一些由必要的控制信息组成的头部后，就构成了一个分组。分组有称为包。分组是在互联网中传送的数据单元，正式由于分组的头部包含了诸如目的地址和源地址等重要控制信息，每一个分组才能在互联网中独立的选择传输路径，并正确地交付到分组传输的终点。

4，互联网按工作方式可划分为边缘部分和核心部分。主机在网络的边缘部分，其作用是进行信息处理。由大量网络和连接这些网络的路由组成边缘部分，其作用是提供连通性和交换。

5，计算机通信是计算机中进程（即运行着的程序）之间的通信。计算机网络采用的通信方式是客户-服务器方式（C/S方式）和对等连接方式（P2P方式）。

6，客户和服务器都是指通信中所涉及的应用进程。客户是服务请求方，服务器是服务提供方。

7，按照作用范围的不同，计算机网络分为广域网WAN，城域网MAN，局域网LAN，个人区域网PAN。

8，计算机网络最常用的性能指标是：速率，带宽，吞吐量，时延（发送时延，处理时延，排队时延），时延带宽积，往返时间和信道利用率。

9，网络协议即协议，是为进行网络中的数据交换而建立的规则。计算机网络的各层以及其协议集合，称为网络的体系结构。

10，五层体系结构由应用层，运输层，网络层（网际层），数据链路层，物理层组成。运输层最主要的协议是TCP和UDP协议，网络层最重要的协议是IP协议。##二物理层##

### （1），基本术语

####

数据（data）：

运送消息的实体。####

信号（signal）：

数据的电气的或电磁的表现。或者说信号是适合在传输介质上传输的对象。####

码元（code）：

在使用时间域（或简称为时域）的波形来表示数字信号时，代表不同离散数值的基本波形。####

单工（simplex）：

只能有一个方向的通信而没有反方向的交互。####

半双工（half duplex）：

通信的双方都可以发送信息，但不能双方同时发送（当然也就不能同时接收）。####

全双工（full duplex）：

通信的双方可以同时发送和接收信息。####

奈氏准则：

在任何信道中，码元的传输的效率是有上限的，传输速率超过此上限，就会出现严重的码间串扰问题，使接收端对码元的判决（即识别）成为不可能。####

基带信号（baseband signal）：

来自信源的信号。指没有经过调制的数字信号或模拟信号。####

带通（频带）信号（bandpass signal）：

把基带信号经过载波调制后，把信号的频率范围搬移到较高的频段以便在信道中传输（即仅在一段频率范围内能够通过信道），这里调制过后的信号就是带通信号。####

调制（modulation）：

对信号源的信息进行处理后加到载波信号上，使其变为适合在信道传输的形式的过程。####

信噪比（signal-to-noise ratio）：

指信号的平均功率和噪声的平均功率之比，记为S/N。信噪比（dB）=10\*log10(S/N) ####

信道复用（channel multiplexing）：

指多个用户共享同一个信道。（并不一定是同时）####

比特率（bit rate）：

单位时间（每秒）内传送的比特数。####

波特率（baud rate）：

单位时间载波调制状态改变的次数。针对数据信号对载波的调制速率。####

复用（multiplexing）：

共享信道的方法####

ADSL（Asymmetric Digital Subscriber Line）：

**非对称数字用户线。####**

**光纤同轴混合网（HFC网）：**

**在目前覆盖范围很广的有线电视网的基础上开发的一种居民宽带接入网 ####**

## (2) , 重要知识点总结

1, 物理层的主要任务就是确定与传输媒体接口有关的一些特性，如机械特性，电气特性，功能特性，过程特性。2, 一个数据通信系统可划分为三大部分，即源系统，传输系统，目的系统。源系统包括源点（或源站，信源）和发送器，目的系统包括接收器和终点。3, 通信的目的是传送消息。如话音，文字，图像等都是消息，数据是运送消息的实体。信号则是数据的电器或电磁的表现。4, 根据信号中代表消息的参数的取值方式不同，信号可分为模拟信号（或连续信号）和数字信号（或离散信号）。在使用时间域（简称时域）的波形表示数字信号时，代表不同离散数值的基本波形称为码元。5, 根据双方信息交互的方式，通信可划分为单向通信（或单工通信），双向交替通信（或半双工通信），双向同时通信（全双工通信）。6, 来自信源的信号称为基带信号。信号要在信道上传输就要经过调制。调制有基带调制和带通调制之分。最基本的带通调制方法有调幅，调频和调相。还有更复杂的调制方法，如正交振幅调制。7, 要提高数据在信道上的传递速率，可以使用更好的传输媒体，或使用先进的调制技术。但数据传输速率不可能任意被提高。8, 传输媒体可分为两大类，即导引型传输媒体（双绞线，同轴电缆，光纤）和非导引型传输媒体（无线，红外，大气激光）。9, 为了有效利用光纤资源，在光纤干线和用户之间广泛使用无源光网络PON。无源光网络无需配备电源，其长期运营成本和管理成本都很低。最流行的无源光网络是以太网无源光网络EPON和吉比特无源光网络GPON。

## (3) , 最重要的知识点

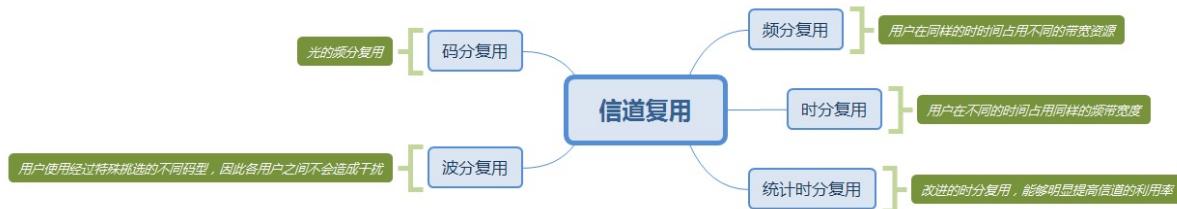
### ①, 物理层的任务

透明地传送比特流。也可以将物理层的主要任务描述为确定与传输媒体的接口的一些特性，即：机械特性（接口所用接线器的一些物理属性如形状尺寸），电气特性（接口电缆的各条线上出现的电压的范围），功能特性（某条线上出现的某一电平的电压的意义），过程特性（对于不同功能能的各种可能事件的出现顺序）。

### 拓展：

物理层考虑的是怎样才能在连接各种计算机的传输媒体上上传输数据比特流，而不是指具体的传输媒体。现有的计算机网络中的硬件设备和传输媒体的种类非常繁多，而且通信手段也有许多不同的方式。物理层的作用正是尽可能地屏蔽掉这些传输媒体和通信手段的差异，使物理层上面的数据链路层感觉不到这些差异，这样就可以使数据链路层只考虑完成本层的协议和服务，而不必考虑网络的具体传输媒体和通信手段是什么。

### ②, 几种常用的信道复用技术



自由主题

[http://blog.csdn.net/qq\\_34337272](http://blog.csdn.net/qq_34337272)

### ③, 几种常用的宽带接入技术，主要是ADSL和FTTx

用户到互联网的宽带接入方法有非对称数字用户线ADSL（用数字技术对现有的模拟电话线进行改造，而不需要重新布线。ADSL的快速版本是甚高速数字用户线VDSL。），光纤同轴混合网HFC（是在目前覆盖范围很广的有线电视网的基础上开发的一种居民宽带接入网）和FTTx（即光纤到……）。

## 三数据链路层

### (1) , 基本术语

#### 链路 (link) :

一个结点到相邻结点的一段物理链路

#### 数据链路 (data link) :

把实现控制数据运输的协议的硬件和软件加到链路上就构成了数据链路

#### 循环冗余检验CRC (Cyclic Redundancy Check) :

为了保证数据传输的可靠性, CRC是数据链路层广泛使用的一种检错技术

#### 帧 (frame) :

一个数据链路层的传输单元, 由一个数据链路层首部和其携带的封包所组成协议数据单元。

#### MTU (Maximum Transfer Unit) :

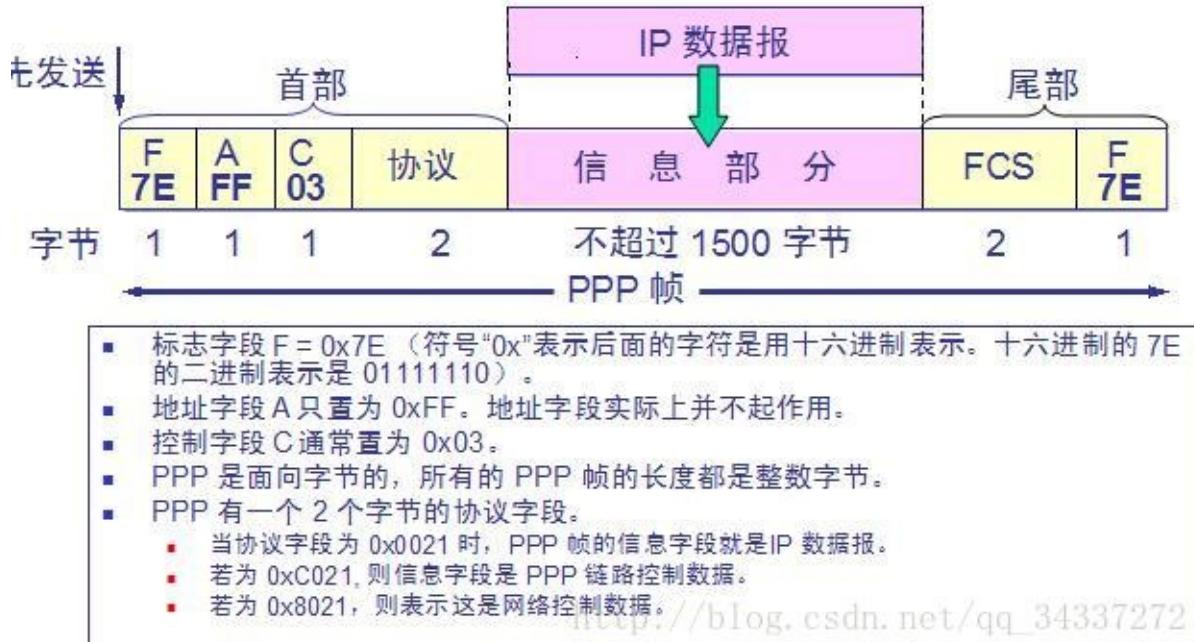
最大传送单元。帧的数据部分的的长度上限。

#### 误码率BER (Bit Error Rate) :

在一段时间内, 传输错误的比特占所传输比特总数的比率。

#### PPP (Point-to-Point Protocol) :

点对点协议。即用户计算机和ISP进行通信时所使用的数据链路层协议。以下是PPP帧的示意图:



## MAC地址 (Media Access Control或者Medium Access Control) :

意译为媒体访问控制，或称为物理地址、硬件地址，用来定义网络设备的位置。  
 在OSI模型中，第三层网络层负责 IP地址，第二层数据链路层则负责 MAC地址。  
 因此一个主机会有一个MAC地址，而每个网络位置会有一个专属于它的IP地址。  
 地址是识别某个系统的重要标识符，“名字指出我们所要寻找的资源，地址指出资源所在的地方，路由告诉我们如何到达该处”

## 网桥 (bridge) :

一种用于数据链路层实现中继，连接两个或多个局域网的网络互连设备。

## 交换机 (switch) :

广义的来说，交换机指的是一种通信系统中完成信息交换的设备。这里工作在数据链路层的交换机指的是交换式集线器，其实质是一个多接口的网桥

## (2) , 重要知识点总结

1, 链路是从一个结点到相邻节点的一段物理链路，数据链路则在链路的基础上增加了一些必要的硬件（如网络适配器）和软件（如协议的实现） 2, 数据链路层使用的主要有\*\*点对点信道\*\*和\*\*广播信道\*\*两种。 3, 数据链路层传输的协议数据单元是帧。数据链路层的三个基本问题是：\*\*封装成帧\*\*， \*\*透明传输\*\*和\*\*差错检测\*\* 4, \*\*循环冗余检验 CRC\*\*是一种检错方法，而帧检验序列FCS是添加在数据后面的冗余码 5, \*\*点对点协议PPP\*\*是数据链路层使用最多的一种协议，它的特点是：简单，只检测差错而不去纠正差错，不使用序号，也不进行流量控制，可同时支持多种网络层协议 6, PPPoE是为宽带上网的主机使用的链路层协议 7, 局域网的优点是：具有广播功能，从一个站点可方便地访问全网；便于系统的扩展和逐渐演变；提高了系统的可靠性，可用性和生存性。 8, 共享媒体通信资源的方法有二：一是静态划分信道(各种复用技术)，而是动态媒体接入控制，又称为多点接入（随即接入或受控接入） 9, 计算机与外接局域网通信需要通过通信适配器（或网络适配器），它又称为网络接口卡或网卡。\*\*计算机的硬件地址就在适配器的 ROM中\*\*。 10, 以太网采用的无连接的工作方式，对发送的数据帧不进行编号，也不要对方发回确认。目的站收到有差错帧就把它丢掉，其他什么也不做 11, 以太网采用的协议是具有冲突检测的\*\*载波监听多点接入CSMA/CD\*\*。协议的特点是：\*\*发送前先监听，边发送边监听，一旦发现总线上出现了碰撞，就立即停止发送。然后按照退避算法等待一段随机时间后再次发送。\*\*因此，每一个站点在自己发送数据之后的一小段时间内，存在这遭遇碰撞的可能性。以太

网上的各站点平等的争用以太网信道 12，以太网的适配器具有过滤功能，它只接收单播帧，广播帧和多播帧。 13，使用集线器可以在物理层扩展以太网（扩展后的以太网任然是一个网络） ####

### (3) , 最重要的知识点

#### ①

数据链路层的点对点信道和广播信道的特点，以及这两种信道所使用的协议（PPP协议以及CSMA/CD协议）的特点

#### ②

数据链路层的三个基本问题：\*\*封装成帧\*\*，\*\*透明传输\*\*，\*\*差错检测\*\*

#### ③

以太网的MAC层硬件地址

#### ④

适配器，转发器，集线器，网桥，以太网交换机的作用以及适用场合

##

## 四网络层

###

### (1) , 基本术语

####

虚电路（Virtual Circuit）：

在两个终端设备的逻辑或物理端口之间，通过建立的双向的透明传输通道。虚电路表示这是一条逻辑上的连接，分组都沿着这条逻辑连接按照存储转发方式传送，而并不是真正建立了一条物理连接。 ####

IP（Internet Protocol）：

网际协议 IP 是 TCP/IP 体系中两个最主要的协议之一，是 TCP/IP 体系结构网际层的核心。配套的有 ARP，RARP，ICMP，IGMP。![这里写图片描述](https://user-gold-cdn.xitu.io/2018/4/1/1627f92f98436286?w=453&h=331&f=jpeg&s=27535) ####

ARP（Address Resolution Protocol）：

地址解析协议 ####

ICMP（Internet Control Message Protocol）：

网际控制报文协议（ICMP 允许主机或路由器报告差错情况和提供有关异常情况的报告。） ####

子网掩码（subnet mask）：

它是一种用来指明一个IP地址的哪些位标识的是主机所在的子网以及哪些位标识的是主机的位掩码。子网掩码不能单独存在，它必须结合IP地址一起使用。 ####

CIDR（Classless Inter-Domain Routing）：

无分类域间路由选择（特点是消除了传统的 A 类、B 类和 C 类地址以及划分子网的概念，并使用各种长度的“网络前缀”(network-prefix)来代替分类地址中的网络号和子网号） ####

默认路由（default route）：

当在路由表中查不到能到达目的地址的路由时，路由器选择的路由。默认路由还可以减小路由表所占用的空间和搜索路由表所用的时间。 ####

路由选择算法（Virtual Circuit）：

路由选择协议的核心部分。因特网采用自适应的，分层次的路由选择协议。 ###

### (2) , 重要知识点总结

1，TCP/IP协议中的网络层向上只提供简单灵活的，无连接的，尽最大努力交付的数据报服务。网络层不提供服务质量的承诺，不保证分组交付的时限所传送的分组可能出错，丢失，重复和失序。进程之间通信的可靠性由运输层负责 2，在互联网的交付有两种，一是在本网络直接交付不用经过路由器，另一种是和其他网络的间接交付，至少经过一个路由器，但最后一次一定是直接交付 3，分类的IP地址由网络号字段（指明网络）和主机号字段（指明主机）组成。网络号字段最前面的类别指明IP地址的类别。IP地址市一中分等级的地址结构。IP地址管理机构分配IP地址时只分配网络号，主机号由得到该网络号的单位自行分配。路由器根据目的主机所连接的网络号来转发分组。一个路由器至少连接到两个网络，所以一个路由器至少应当有两个不同的IP地址 4，IP数据报分为首部和数据两部分。首部的前一部分是固定长度，共20字节，是所有IP数据包必须具有的（源地址，目的地址，总长度等重要字段都固定在首部）。一些长度可变的可选字段固定在首部的后面。IP首部中的生存时间给出了IP数据报在互联网中所能经过的最大路由器数。可防止IP数据报在互联网中无限制的兜圈子。 5，地址解析协议ARP把IP地址解析为硬件地址。ARP的高速缓存可以大大减少网络上的通信量。因为这样可以使主机下次再与同样地址的主机通信时，可以直接从高速缓存中找到所需要的硬件地址而不需要再去广播方式发送ARP请求分组 6，无分类域间路由选择CIDR是解决目前IP地址紧缺的一个好办法。CIDR记法把IP

地址后面加上斜线“/”，然后写上前缀所占的位数。前缀（或网络前缀用来指明网络），前缀后面的部分是后缀，用来指明主机。CIDR把前缀都相同的连续的IP地址组成一个“CIDR地址块”，IP地址分配都以CIDR地址块为单位。7，网际控制报文协议是IP层的协议.ICMP报文作为IP数据报的数据，加上首部后组成IP数据报发送出去。使用ICMP数据报并不是为了实现可靠传输。ICMP允许主机或路由器报告差错情况和提供有关异常情况的报告。ICMP报文的种类有两种 ICMP差错报告报文和ICMP询问报文。8，要解决IP地址耗尽的问题，最根本的办法是采用具有更大地址空间的新版本IP协议-IPv6。IPv6所带来的变化有①更大的地址空间（采用128位地址）②灵活的首部格式③改进的选项④支持即插即用⑤支持资源的预分配⑥IPv6的首部改为8字节对齐。另外IP数据报的目的地址可以是以下三种基本类型地址之一：单播，多播和任播9，虚拟专用网络VPN利用公用的互联网作为本机构专用网之间的通信载体。VPN内使用互联网的专用地址。一个VPN至少要有一个路由器具有合法的全球IP地址，这样才能和本系统的另一个VPN通过互联网进行通信。所有通过互联网传送的数据都需要加密10，MPLS的特点是：①支持面向连接的服务质量②支持流量工程，平衡网络负载③有效的支持虚拟专用网VPN。MPLS在入口节点给每一个IP数据报打上固定长度的“标记”，然后根据标记在第二层（链路层）用硬件进行转发（在标记交换路由器中进行标记交换），因而转发速率大大加快。

### (3) , 最重要知识点

① 虚拟互联网络的概念

② IP地址和物理地址的关系

③ 传统的分类的IP地址（包括子网掩码）和误分类域间路由选择CIDR

④ 路由选择协议的工作原理

## 五运输层

### (1) , 基本术语

进程 (process) :

指计算机中正在运行的程序实体

应用进程互相通信：

一台主机的进程和另一台主机中的一个进程交换数据的过程（另外注意通信真正的端点不是主机而是主机中的进程，也就是说端到端的通信是应用进程之间的通信）

传输层的复用与分用：

复用指发送方不同的进程都可以通过统一个运输层协议传送数据。分用指接收方的运输层在剥去报文的首部后能把这些数据正确的交付到目的应用进程。

TCP (Transmission Control Protocol) :

传输控制协议

UDP (User Datagram Protocol) :

用户数据报协议

## 端口 (port) (link) :

端口的目的是为了确认对方机器是那个进程在自己进行交互，比如MSN和QQ的端口不同，如果没有端口就可能出现QQ进程和MSN交互错误。端口又称协议端口号。

## 停止等待协议 (link) :

指发送方每发送完一个分组就停止发送，等待对方确认，在收到确认之后再发送下一个分组。

## 流量控制 (link) :

就是让发送方的发送速率不要太快，既要让接收方来得及接收，也不要使网络发生拥塞。

## 拥塞控制 (link) :

防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。拥塞控制所要做的都有一个前提，就是网络能够承受现有的网络负荷。

## (2) , 重要知识点总结

1, 运输层提供应用进程之间的逻辑通信，也就是说，运输层之间的通信并不是真正在两个运输层之间直接传输数据。运输层向应用层屏蔽了下面网络的细节（如网络拓扑，所采用的路由选择协议等），它使应用进程之间看起来好像两个运输层实体之间有一条端到端的逻辑通信信道。

2, 网络层为主机提供逻辑通信，而运输层为应用进程之间提供端到端的逻辑通信。

3, 运输层的两个重要协议是用户数据报协议UDP和传输控制协议TCP。按照OSI的术语，两个对等运输实体在通信时传送的数据单位叫做运输协议数据单元TPDU (Transport Protocol Data Unit)。但在TCP/IP体系中，则根据所使用的协议是TCP或UDP，分别称之为TCP报文段或UDP用户数据报。

4, UDP在传送数据之前不需要先建立连接，远地主机在收到UDP报文后，不需要给出任何确认。虽然UDP不提供可靠交付，但在某些情况下UDP确是一种最有效的工作方式。TCP提供面向连接的服务。在传送数据之前必须先建立连接，数据传送结束后要释放连接。TCP不提供广播或多播服务。由于TCP要提供可靠的，面向连接的运输服务，这一难以避免增加了许多开销，如确认，流量控制，计时器以及连接管理等。这不仅使协议数据单元的首部增大很多，还要占用许多处理机资源。

5, 硬件端口是不同硬件设备进行交互的接口，而软件端口是应用层各种协议进程与运输实体进行层间交互的一种地址。UDP和TCP的首部格式中都有源端口和目的端口这两个重要字段。当运输层收到IP层交上来的运输层报文时，就能够根据其首部中的目的端口号把数据交付应用层的目的应用层。（两个进程之间进行通信不光要知道对方IP地址而且要知道对方的端口号(为了找到对方计算机中的应用进程)）

6, 运输层用一个16位端口号标志一个端口。端口号只有本地意义，它只是为了标志计算机应用层中的各个进程在和运输层交互时的层间接口。在互联网的不同计算机中，相同的端口号是没有关联的。协议端口号简称端口。虽然通信的终点是应用进程，但只要把所发送的报文交到目的主机的某个合适端口，剩下的工作（最后交付目的进程）就由TCP和UDP来完成。

7, 运输层的端口号分为服务器端使用的端口号（0~1023指派给熟知端口，1024~49151是登记端口号）和客户端暂时使用的端口号（49152~65535）

8, UDP的主要特点是①无连接②尽最大努力交付③面向报文④无拥塞控制⑤支持一对一，一对多，多对一和多对多的交互通信⑥首部开销小（只有四个字段：源端口，目的端口，长度和检验和）

9, TCP的主要特点是①面向连接②每一条TCP连接只能是一对一的③提供可靠交付④提供全双工通信⑤面向字节流

10, TCP用主机的IP地址加上主机上的端口号作为TCP连接的端点。这样的端点就叫做套接字（socket）或插口。套接字用（IP地址：端口号）来表示。每一条TCP连接唯一被通信两端的两个端点所确定。

11, 停止等待协议是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组。

12, 为了提高传输效率，发送方可以不使用低效率的停止等待协议，而是采用流水线传输。流水线传输就是发送方可连续发送多个分组，不必每发完一个分组就停下来等待对方确认。这样可使信道上一直有数据不间断的在传送。这种传输方式可以明显提高信道利用率。

13, 停止等待协议中超时重传是指只要超过一段时间仍然没有收到确认，就重传前面发送过的分组（认为刚才发送过的分组丢失了）。因此每发送完一个分组需要设置一个超时计时器，其重传时间应比数据在分组传输的平均往返时间更长一些。这种自动重传方式常称为自动重传请求ARQ。另外在停止等待协议中若收到重复分组，就丢弃该分组，但同时还要发送确认。连续ARQ协议可提高信道利用率。发送维持一个发送窗口，凡位于发送窗口内的分组可连续发送出去，而不需要等待对方确认。接收方一般采用累积确认，对按序到达的最后一个分组发送确认，表明到这个分组位置的所有分组都已经正确收到了。

14, TCP报文段的前20个字节是固定的，后面有 $4n$ 字节是根据需要增加的选项。因此，TCP首部的最小长度是20字节。

15, TCP使用滑动窗口机制。发送窗口里面的序号表示允许发送的序号。发送窗口后沿的后面部分表示已发送且已收到确认，而发送窗口前沿的前面部分表示不晕与发送。发送窗口后沿的变化情况有两种可能，即不动（没有收到新的确认）和前移（收到了新的确认）。发送窗口的前沿通常是不断向前移动的。一般来说，我们总是希望数据传输更快一些。但如果发送方把数据发送的过快，接收方就可能来不及接收，这就会造成数据的丢失。所谓流量控制就是让发送方的发送速率不要太快，要让接收方来得及接收。

16, 在某段时间，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。这种情况就叫拥塞。拥塞控制就是为了防止过多的数据注入到网络中，这样就可以使网络中的路由器或链路不致过载。拥塞控制所要做的都有一个前提，就是网络能够承受现有的网络负荷。拥塞控制是一个全局性的过程，涉及到所有的主机，所有的路由器，以及与降低网络传输性能有关的所有因素。相反，流量控制往往是点对点通信量的控制，是个端到端的问题。流量控制所要做的就是抑制发送端发送数据的速率，以便使接收端来得及接收。

17, 为了进行拥塞控制，TCP发送方要维持一个拥塞窗口cwnd的状态变量。拥塞控制窗口的大小取决于网络的拥塞程度，并且动态变化。发送方让自己的发送窗口取为拥塞窗口和接收方的接受窗口中较小的一个。

18, TCP的拥塞控制采用了四种算法，即慢开始，拥塞避免，快重传和快恢复。在网络层也可以使路由器采用适当的分组丢弃策略（如主动队列管理AQM），以减少网络拥塞的发生。

19, 运输连接的三个阶段，即：连接建立，数据传送和连接释放。

20, 主动发起TCP连接建立的应用进程叫做客户，而被动等待连接建立的应用进程叫做服务器。TCP连接采用三报文握手机制。服务器要确认用户的连接请求，然后客户要对服务器的确认进行确认。

21, TCP的连接释放采用四报文握手机制。任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。当另一方也没有数据再发送时，则发送连接释放通知，对方确认后就完全关闭了TCP连接

### (3) , 最重要的知识点

① 端口和套接字的意义

② 无连接UDP的特点

### ③ 面向连接TCP的特点

### ④ 在不可靠的网络上实现可靠传输的工作原理，停止等待协议和ARQ协议

### ① TCP的滑动窗口，流量控制，拥塞控制和连接管理

## 六应用层

### (1) , 基本术语

#### 域名系统（DNS）：

DNS (Domain Name System, 域名系统)，万维网上作为域名和IP地址相互映射的一个分布式数据库，能够使用户更方便的访问互联网，而不用去记住能够被机器直接读取的IP地址。

通过域名，最终得到该域名对应的IP地址的过程叫做域名解析（或主机名解析）。DNS协议运行在UDP协议之上，使用端口号53。在RFC文档中RFC 2181对DNS有规范说明，RFC 2136对DNS的动态更新进行说明，RFC 2308对DNS查询的反向缓存进行说明。

#### 文件传输协议（FTP）：

FTP是File Transfer Protocol（文件传输协议）的英文简称，而中文简称为“文传协议”。用于Internet上的控制文件的双向传输。同时，它也是一个应用程序（Application）。

基于不同的操作系统有不同的FTP应用程序，而所有这些应用程序都遵守同一种协议以传输文件。在FTP的使用当中，用户经常遇到两个概念：“下载”（Download）和“上传”（Upload）。

“下载”文件就是从远程主机拷贝文件至自己的计算机上；“上传”文件就是将文件从自己的计算机中拷贝至远程主机上。用Internet语言来说，用户可通过客户机程序向（从）远程主机上传（下载）文件。

#### 简单文件传输协议（TFTP）：

TFTP (Trivial File Transfer Protocol, 简单文件传输协议) 是TCP/IP协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务。端口号为69。

#### 远程终端协议（TELNET）：

Telnet协议是TCP/IP协议族中的一员，是Internet远程登陆服务的标准协议和主要方式。它为用户提供了在本地计算机上完成远程主机工作的能力。在终端使用者的电脑上使用telnet程序，用它连接到服务器。终端使用者可以在telnet程序中输入命令，这些命令会在服务器上运行，就像直接在服务器的控制台上输入一样。

可以在本地就能控制服务器。要开始一个telnet会话，必须输入用户名和密码来登录服务器。Telnet是常用的远程控制Web服务器的方法。

#### 万维网（WWW）：

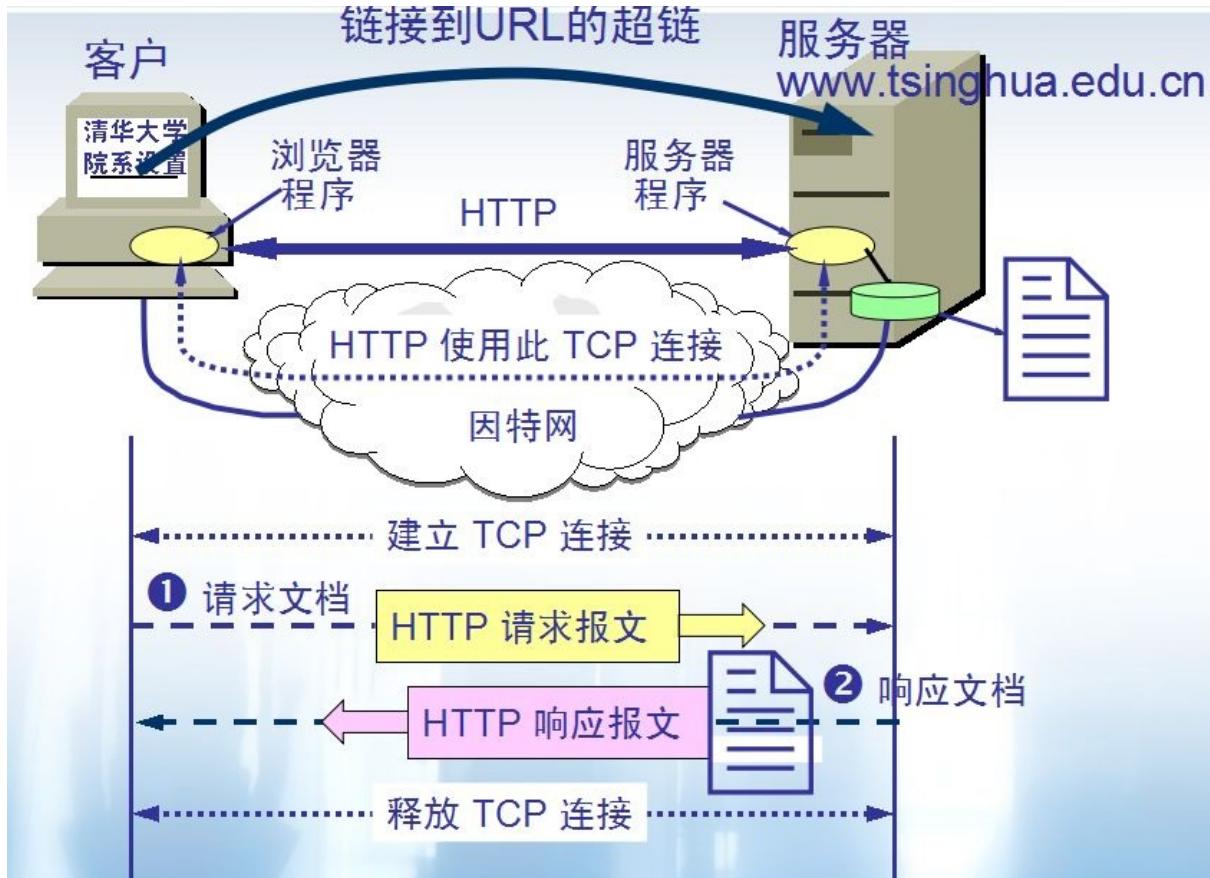
WWW是环球信息网的缩写，（亦作“Web”、“Www”、“'W3'”，英文全称为“World Wide Web”），中文名字为“万维网”，“环球网”等，常简称为Web。分为Web客户端和Web服务器程序。

WWW可以让Web客户端（常用浏览器）访问浏览Web服务器上的页面。是一个由许多互相链接的超文本组成的系统，通过互联网访问。在这个系统中，每个有用的事物，称为一样“资源”；并且由一个全局“统一资源标识符”（URI）标识；这些资源通过超文本传输协议（Hypertext Transfer Protocol）传送给用户，而后者通过点击链接来获得资源。

万维网联盟（英语：World Wide Web Consortium，简称W3C），又称W3C理事会。1994年10月在麻省理工学院（MIT）计算机科学实验室成立。万维网联盟的创建者是万维网的发明者蒂姆·伯纳斯-李。

万维网并不等同互联网，万维网只是互联网所能提供的服务其中之一，是靠着互联网运行的一项服务。

#### 万维网的大致工作流程：



## 统一资源定位符（URL）：

统一资源定位符是对可以从互联网上得到的资源的位置和访问方法的一种简洁的表示，是互联网上标准资源的地址。互联网上的每个文件都有一个唯一的URL，它包含的信息指出文件的位置以及浏览器应该怎么处理它。

## 超文本传输协议（HTTP）：

超文本传输协议（HTTP, HyperText Transfer Protocol）是互联网上应用最为广泛的一种网络协议。所有的WWW文件都必须遵守这个标准。设计HTTP最初的是为了提供一种发布和接收HTML页面的方法。1960年美国人Ted Nelson构思了一种通过计算机处理文本信息的方法，并称之为超文本（hypertext），这成为了HTTP超文本传输协议标准架构的发展根基。

## 代理服务器（Proxy Server）：

代理服务器（Proxy Server）是一种网络实体，它又称为万维网高速缓存。  
代理服务器把最近的一些请求和响应暂存在本地磁盘中。当新请求到达时，若代理服务器发现这个请求与暂时存放的请求相同，就返回暂存的响应，而不需要按URL的地址再次去互联网访问该资源。  
代理服务器可在客户端或服务器工作，也可以在中间系统工作。

## http请求头：

http请求头，HTTP客户程序（例如浏览器），向服务器发送请求的时候必须指明请求类型（一般是GET或者POST）。如有必要，客户程序还可以选择发送其他的请求头。

- **Accept**: 浏览器可接受的MIME类型。
- **Accept-Charset**: 浏览器可接受的字符集。
- **Accept-Encoding**: 浏览器能够进行解码的数据编码方式，比如gzip。Servlet能够向支持gzip的浏览器返回经gzip编码的HTML页面。许多情况下这可以减少5到10倍的下载时间。
- **Accept-Language**: 浏览器所希望的语言种类，当服务器能够提供一种以上的语言版本时要用到。

- **Authorization**: 授权信息，通常出现在对服务器发送的WWW-Authenticate头的应答中。
- **Connection**: 表示是否需要持久连接。如果Servlet看到这里的值为“Keep-Alive”，或者看到请求使用的是HTTP 1.1 (HTTP 1.1默认进行持久连接)，它就可以利用持久连接的优点，当页面包含多个元素时（例如Applet, 图片），显著地减少下载所需的时间。要实现这一点，Servlet需要在应答中发送一个Content-Length头，最简单的实现方法是：先把内容写入ByteArrayOutputStream，然后在正式写出内容之前计算它的大小。
- **Content-Length**: 表示请求消息正文的长度。
- **Cookie**: 这是最重要的请求头信息之一。
- **From**: 请求发送者的email地址，由一些特殊的Web客户程序使用，浏览器不会用到它。
- **Host**: 初始URL中的主机和端口。
- **If-Modified-Since**: 只有当所请求的内容在指定的日期之后又经过修改才返回它，否则返回304“Not Modified”应答。
- **Pragma**: 指定“no-cache”值表示服务器必须返回一个刷新后的文档，即使它是代理服务器而且已经有了页面的本地拷贝。
- **Referer**: 包含一个URL，用户从该URL代表的页面出发访问当前请求的页面。
- **User-Agent**: 浏览器类型，如果Servlet返回的内容与浏览器类型有关则该值非常有用。

## 简单邮件传输协议(SMTP):

SMTP (Simple Mail Transfer Protocol) 即简单邮件传输协议，它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。  
SMTP协议属于TCP/IP协议簇，它帮助每台计算机在发送或中转信件时找到下一个目的地。  
通过SMTP协议所指定的服务器，就可以把E-mail寄到收信人的服务器上了，整个过程只要几分钟。SMTP服务器则是遵循SMTP协议的发送邮件服务器，用来发送或中转发出的电子邮件。

## 搜索引擎:

搜索引擎 (Search Engine) 是指根据一定的策略、运用特定的计算机程序从互联网上搜集信息，在对信息进行组织和处理后，为用户提供检索服务，将用户检索相关的信息展示给用户的系统。  
搜索引擎包括全文索引、目录索引、元搜索引擎、垂直搜索引擎、集合式搜索引擎、门户搜索引擎与免费链接列表等。

## 全文索引:

全文索引技术是目前搜索引擎的关键技术。  
试想在1M大小的文件中搜索一个词，可能需要几秒，在100M的文件中可能需要几十秒，如果在更大的文件中搜索那么就需要更大的系统开销，这样的开销是不现实的。  
所以在这样的矛盾下出现了全文索引技术，有时候有人叫倒排文档技术。

## 目录索引:

目录索引 ( search index/directory)，顾名思义就是将网站分门别类地存放在相应的目录中，因此用户在查询信息时，可选择关键词搜索，也可按分类目录逐层查找。

## 垂直搜索引擎:

垂直搜索引擎是针对某一个行业的专业搜索引擎，是搜索引擎的细分和延伸，是对网页库中的某类专门的信息进行一次整合，定向分字段抽取出需要的数据进行处理后再以某种形式返回给用户。  
垂直搜索是相对通用搜索引擎的信息量大、查询不准确、深度不够等提出来的新的搜索引擎服务模式，通过针对某一特定领域、某一特定人群或某一特定需求提供的有一定价值的信息和相关服务。  
其特点就是“专、精、深”，且具有行业色彩，相比较通用搜索引擎的海量信息无序化，垂直搜索引擎则显得更加专注、具体和深入。

## (2) , 重要知识点总结

1, 文件传输协议 (FTP) 使用TCP可靠的运输服务。FTP使用客户服务器方式。一个FTP服务器进程可以同时为多个用户提供服务。在进行文件传输时，FTP的客户和服务器之间要先建立两个并行的TCP连接:控制连接和数据连接。实际用于传输文件的是数据连接。

2, 万维网客户程序与服务器之间进行交互使用的协议时超文本传输协议HTTP。HTTP使用TCP连接进行可靠传输。但HTTP本身是无连接、无状态的。HTTP/1.1协议使用了持续连接（分为非流水线方式和流水线方式）

3, 电子邮件把邮件发送到收件人使用的邮件服务器，并放在其中的收件人邮箱中，收件人可随时上网到自己使用的邮件服务器读取，相当于电子邮箱。

4, 一个电子邮件系统有三个重要组成构件：用户代理、邮件服务器、邮件协议（包括邮件发送协议，如SMTP，和邮件读取协议，如POP3和IMAP）。用户代理和邮件服务器都要运行这些协议。

### **(3) , 最重要知识点总结**

**① 域名系统-从域名解析出IP地址**

**② 访问一个网站大致的过程**

**③ 系统调用和应用编程接口概念**

## RPC

**RPC (Remote Procedure Call) —远程过程调用**，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发分布式程序就像开发本地程序一样简单。

**RPC采用客户端（服务调用方）/服务器端（服务提供方）模式**，都运行在自己的JVM中。客户端只需要引入要使用的接口，接口的实现和运行都在服务器端。RPC主要依赖的技术包括序列化、反序列化和数据传输协议，这是一种定义与实现相分离的设计。

目前Java使用比较多的RPC方案主要有**RMI (JDK自带)**、**Hessian**、**Dubbo**以及**Thrift**等。

注意：RPC主要指内部服务之间的调用，RESTful也可以用于内部服务之间的调用，但其主要用途还在于外部系统提供服务，因此没有将其包含在本知识点内。

### 常见RPC框架：

- **RMI (JDK自带)**： JDK自带的RPC

详细内容可以参考：[从懵逼到恍然大悟之Java中RMI的使用](#)

- **Dubbo**: Dubbo是阿里巴巴公司开源的一个高性能优秀的服务框架，使得应用可通过高性能的RPC实现服务的输出和输入功能，可以和Spring框架无缝集成。

详细内容可以参考：

- [高性能优秀的服务框架-dubbo介绍](#)
- [Dubbo是什么？能做什么？](#)

- **Hessian**: Hessian是一个轻量级的remotingonhttp工具，使用简单的方法提供了RMI的功能。相比WebService，Hessian更简单、快捷。采用的是二进制RPC协议，因为采用的是二进制协议，所以它很适合于发送二进制数据。

详细内容可以参考：[Hessian的使用以及理解](#)

- **Thrift**: Apache Thrift是Facebook开源的跨语言的RPC通信框架，目前已经捐献给Apache基金会管理，由于其跨语言特性和出色的性能，在很多互联网公司得到应用，有能力的公司甚至会基于thrift研发一套分布式服务框架，增加诸如服务注册、服务发现等功能。

详细内容可以参考： [【Java】分布式RPC通信框架Apache Thrift 使用总结] (<https://www.cnblogs.com/zeze/p/8628585.html>)

### 如何进行选择：

- 是否允许代码侵入：即需要依赖相应的代码生成器生成代码，比如Thrift。
- 是否需要长连接获取高性能：如果对于性能需求较高的haul，那么可以果断选择基于TCP的Thrift、Dubbo。
- 是否需要跨越网段、跨越防火墙：这种情况一般选择基于HTTP协议的Hessian和Thrift的HTTP Transport。

此外，Google推出的基于HTTP2.0的gRPC框架也开始得到应用，其序列化协议基于Protobuf，网络框架使用的是Netty4，但是其需要生成代码，可扩展性也比较差。

## 消息中间件

消息中间件，也可以叫做中央消息队列或者是消息队列（区别于本地消息队列，本地消息队列指的是JVM内的队列实现），是一种独立的队列系统，消息中间件经常用来解决内部服务之间的 异步调用问题。请求服务方把请求队列放到队列中即可返回，然后等待服务提供方去队列中获取请求进行处理，之后通过回调等机制把结果返回给请求服务方。

异步调用只是消息中间件一个非常常见的应用场景。此外，常用的消息队列应用场景还偷如下几个：

- **解耦**：一个业务的非核心流程需要依赖其他系统，但结果并不重要，有通知即可。
- **最终一致性**：指的是两个系统的状态保持一致，可以有一定的延迟，只要最终达到一致性即可。经常用在解决分布式事务上。
- **广播**：消息队列最基本的功能。生产者只负责生产消息，订阅者接收消息。
- **错峰和流控**

具体可以参考：

[《消息队列深入解析》](#)

当前使用较多的消息队列有ActiveMQ（性能差，不推荐使用）、RabbitMQ、RocketMQ、Kafka等等，我们之前提到的redis数据库也可以实现消息队列，不过不推荐，redis本身设计就不是用来做消息队列的。

- **ActiveMQ**: ActiveMQ是Apache出品，最流行的，能力强劲的开源消息总线。ActiveMQ是一个完全支持JMS1.1 和J2EE 1.4规范的JMSProvider实现,尽管JMS规范出台已经是很久的事情了,但是JMS在当今的J2EE应用中间仍然扮演着特殊的地位。

具体可以参考：

[《消息队列ActiveMQ的使用详解》](#)

- **RabbitMQ**: RabbitMQ 是一个由 Erlang 语言开发的 AMQP 的开源实现。RabbitMQ 最初起源于金融系统，用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗

AMQP : Advanced Message Queue，高级消息队列协议。它是应用层协议的一个开放标准，为面向消息的中间件设计，基于此协议的客户端与消息中间件可传递消息，并不受产品、开发语言等条件的限制。

具体可以参考：

[《消息队列之 RabbitMQ》](#)

- **RocketMQ**:

具体可以参考：

[《RocketMQ 实战之快速入门》](#)

[《十分钟入门RocketMQ》](#) (阿里中间件团队博客)

- **Kafka**: Kafka是一个分布式的、可分区的、可复制的、基于发布/订阅的消息系统,Kafka主要用于大数据领域,当然在分布式系统中也有应用。目前市面上流行的消息队列RocketMQ就是阿里借鉴Kafka的原理、用Java开发而得。

具体可以参考：

[《Kafka应用场景》](#)

[《初谈Kafka》](#)

推荐阅读：

[《Kafka、RabbitMQ、RocketMQ等消息中间件的对比 —— 消息发送性能和区别》](#)



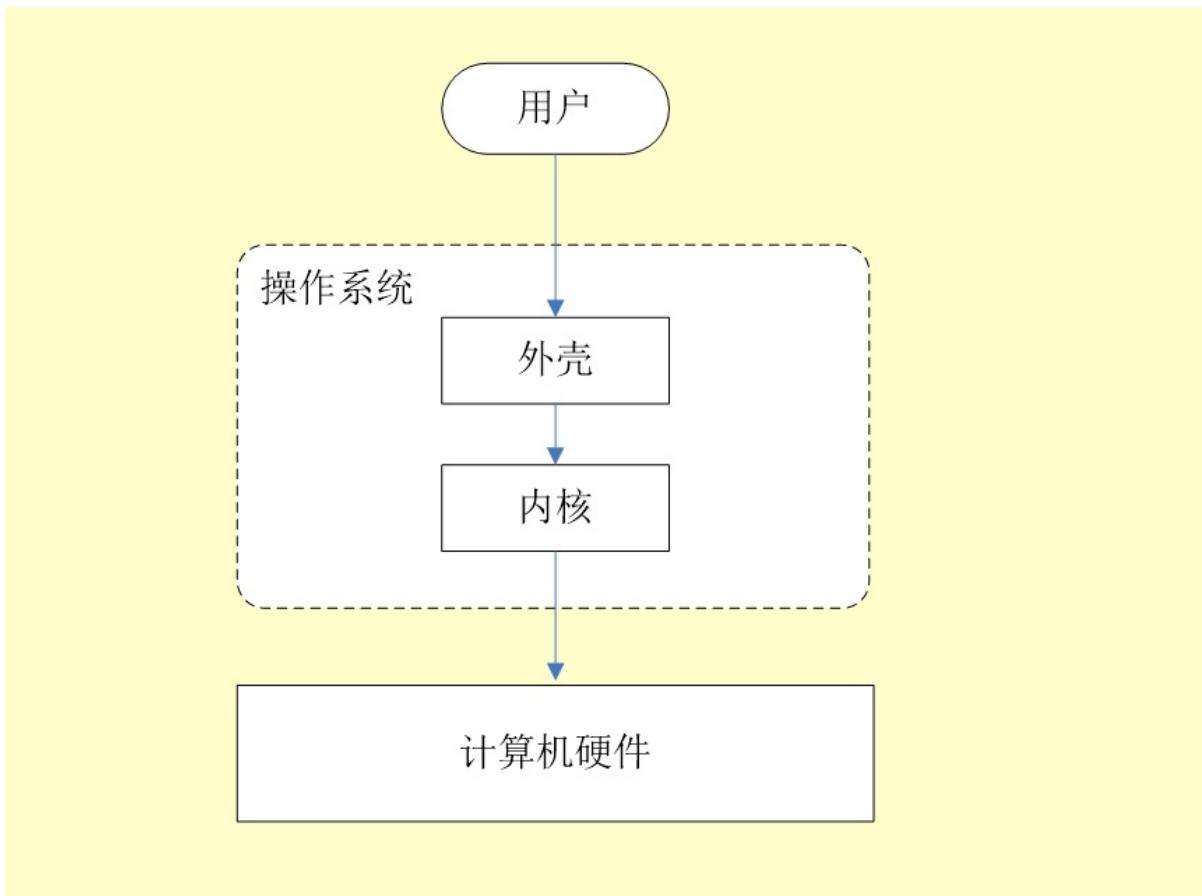
学习Linux之前，我们先来简单的认识一下操作系统。

# 一 从认识操作系统开始

## 1.1 操作系统简介

我通过以下四点介绍什么操作系统：

- 操作系统（Operation System，简称OS）是管理计算机硬件与软件资源的程序，是计算机系统的内核与基石；
- 操作系统本质上是运行在计算机上的软件程序；
- 为用户提供一个与系统交互的操作界面；
- 操作系统分内核与外壳（我们可以把外壳理解成围绕着内核的应用程序，而内核就是能操作硬件的程序）。

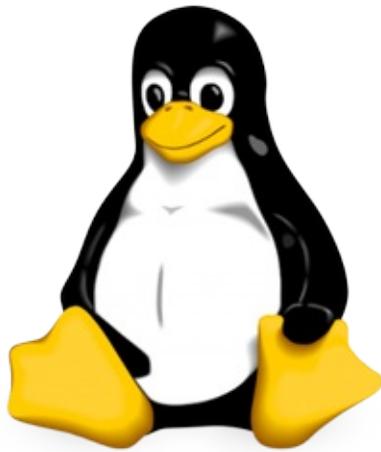


## 1.2 操作系统简单分类

1. **Windows**: 目前最流行的个人桌面操作系统，不做多的介绍，大家都清楚。
2. **Unix**: 最早的多用户、多任务操作系统。按照操作系统的分类，属于分时操作系统。Unix 大多被用在服务器、工作站，现在也有用在个人计算机上。它在创建互联网、计算机网络或客户端/服务器模型方面发挥着非常重要的作用。



3. **Linux**: Linux是一套免费使用和自由传播的类Unix操作系统。Linux存在着许多不同的Linux版本，但它们都使用了**Linux内核**。Linux可安装在各种计算机硬件设备中，比如手机、平板电脑、路由器、视频游戏控制台、台式计算机、大型机和超级计算机。严格来讲，Linux这个词本身只表示Linux内核，但实际上人们已经习惯了用Linux来形容整个基于Linux内核，并且使用GNU 工程各种工具和数据库的操作系统。



## 二 初探Linux

### 2.1 Linux简介

我们上面已经介绍到了Linux，我们这里只强调三点。

- **类Unix系统**: Linux是一种自由、开放源码的类似Unix的操作系统
- **Linux内核**: 严格来说，Linux这个词本身只表示Linux内核
- **Linux之父**: 一个编程领域的传奇式人物。他是Linux内核的最早作者，随后发起了这个开源项目，担任Linux内核的首要架构师与项目协调者，是当今世界最著名的电脑程序员、黑客之一。他还发起了Git这个开源项目，并为主要的开发者。



## 2.2 Linux诞生简介

- 1991年，芬兰的业余计算机爱好者Linus Torvalds编写了一款类似Minix的系统（基于微内核架构的类Unix操作系统）被ftp管理员命名为Linux 加入到自由软件基金的GNU计划中；
- Linux以一只可爱的企鹅作为标志，象征着敢作敢为、热爱生活。

## 2.3 Linux的分类

Linux根据原生程度，分为两种：

1. 内核版本： Linux不是一个操作系统，严格来讲，Linux只是一个操作系统中的内核。内核是什么？内核建立了计算机软件与硬件之间通讯的平台，内核提供系统服务，比如文件管理、虚拟内存、设备I/O等；
2. 发行版本：一些组织或公司在内核版基础上进行二次开发而重新发行的版本。Linux发行版本有很多（ubuntu和CentOS用的都很多，初学建议选择CentOS），如下图所示：



## 三 Linux文件系统概览

### 3.1 Linux文件系统简介

在Linux操作系统中，所有被操作系统管理的资源，例如网络接口卡、磁盘驱动器、打印机、输入输出设备、普通文件或是目录都被看作是一个文件。

也就是说在LINUX系统中有一个重要的概念：一切都是文件。其实这是UNIX哲学的一个体现，而Linux是重写UNIX而来，所以这个概念也就传承了下来。在UNIX系统中，把一切资源都看作是文件，包括硬件设备。UNIX系统把每个硬件都看成是一个文件，通常称为设备文件，这样用户就可以用读写文件的方式实现对硬件的访问。

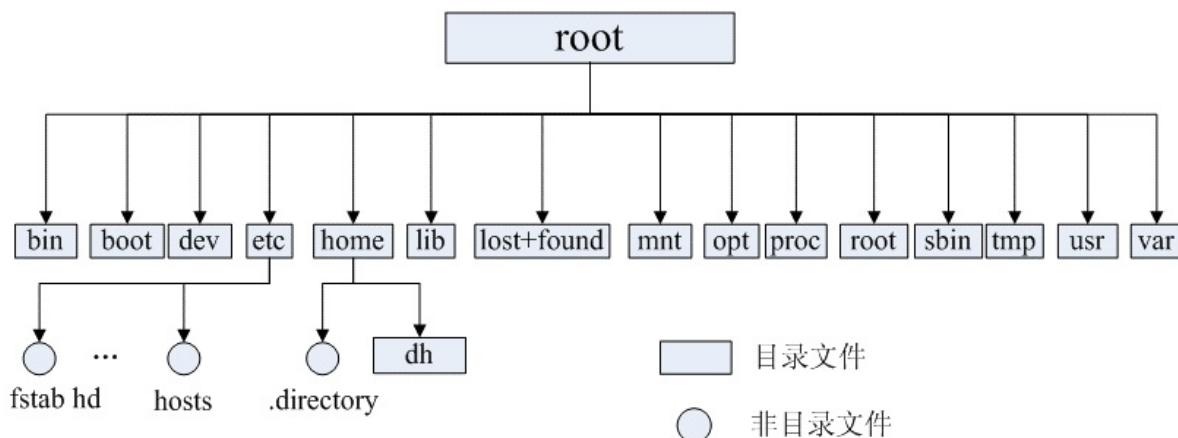
### 3.2 文件类型与目录结构

Linux支持5种文件类型：

| 文件类型       | 描述                                  | 示例                                          |
|------------|-------------------------------------|---------------------------------------------|
| 普通文件       | 用来在辅助存储设备（如磁盘）上存储信息和数据              | 包含程序源代码（用C、C++、Java等语言所编写）、可执行程序、图片、声音、图像等  |
| 目录文件       | 用于表示和管理系统中的文件，目录文件中包含一些文件名和子目录名     | /root、/home                                 |
| 链接文件       | 用于不同目录下文件的共享                        | 当创建一个已存在文件的符号链接时，系统就创建一个链接文件，这个链接文件指向已存在的文件 |
| 设备文件       | 用来访问硬件设备                            | 包括键盘、硬盘、光驱、打印机等                             |
| 命名管道（FIFO） | 是一种特殊类型的文件，Linux系统下，进程之间通信可以通过该文件完成 |                                             |

Linux的目录结构如下：

Linux文件系统的结构层次鲜明，就像一棵倒立的树，最顶层是其根目录：



常见目录说明：

- **/bin:** 存放二进制可执行文件(ls,cat,mkdir等)，常用命令一般都在这里；
- **/etc:** 存放系统管理和配置文件；
- **/home:** 存放所有用户文件的根目录，是用户主目录的基点，比如用户user的主目录就是/home/user，可以用~user表示；
- **/usr :** 用于存放系统应用程序；
- **/opt:** 额外安装的可选应用程序包所放置的位置。一般情况下，我们可以把tomcat等都安装到这里；
- **/proc:** 虚拟文件系统目录，是系统内存的映射。可直接访问这个目录来获取系统信息；
- **/root:** 超级用户（系统管理员）的主目录（特权阶级^o^）；
- **/sbin:** 存放二进制可执行文件，只有root才能访问。这里存放的是系统管理员使用的系统级别的管理命令和程序。如ifconfig等；
- **/dev:** 用于存放设备文件；
- **/mnt:** 系统管理员安装临时文件系统的安装点，系统提供这个目录是让用户临时挂载其他的文件系统；
- **/boot:** 存放用于系统引导时使用的各种文件；
- **/lib :** 存放着和系统运行相关的库文件；
- **/tmp:** 用于存放各种临时文件，是公用的临时文件存储点；
- **/var:** 用于存放运行时需要改变数据的文件，也是某些大文件的溢出区，比方说各种服务的日志文件（系统启动日志等。）等；
- **/lost+found:** 这个目录平时是空的，系统非正常关机而留下“无家可归”的文件（windows下叫什么.chk）就在这里。

## 四 Linux基本命令

下面只是给出了一些比较常用的命令。推荐一个Linux命令快查网站，非常不错，大家如果遗忘某些命令或者对某些命令不理解都可以在这里得到解决。

Linux命令大全：<http://man.linuxde.net/>

### 4.1 目录切换命令

- `cd usr` : 切换到该目录下usr目录
- `cd ..` (或`cd..`) : 切换到上一层目录
- `cd /` : 切换到系统根目录
- `cd ~` : 切换到用户主目录
- `cd -` : 切换到上一个所在目录

### 4.2 目录的操作命令（增删改查）

1. `mkdir 目录名称` : 增加目录
2. `ls或者ll` (`ll`是`ls -l`的缩写，`ll`命令以看到该目录下的所有目录和文件的详细信息) : 查看目录信息
3. `find 目录 参数` : 寻找目录 (查)

示例：

- 列出当前目录及子目录下所有文件和文件夹: `find .`
  - 在 `/home` 目录下查找以.txt结尾的文件名: `find /home -name "*.txt"`
  - 同上，但忽略大小写: `find /home -iname "*.txt"`
  - 当前目录及子目录下查找所有以.txt和.pdf结尾的文件: `find . \(-name "*.txt" -o -name "*.pdf"\)` 或 `find . -name "*.txt" -o -name "*.pdf"`
4. `mv 目录名称 新目录名称` : 修改目录的名称 (改)

注意：`mv`的语法不仅可以对目录进行重命名而且也可以对各种文件，压缩包等进行重命名的操作。`mv`命令用来对文件或目录重新命名，或者将文件从一个目录移到另一个目录中。后面会介绍到`mv`命令的另一个用法。

## 5. mv 目录名称 目录的新位置 : 移动目录的位置---剪切 (改)

注意: mv语法不仅可以对目录进行剪切操作, 对文件和压缩包等都可执行剪切操作。另外mv与cp的结果不同, mv好像文件“搬家”, 文件个数并未增加。而cp对文件进行复制, 文件个数增加了。

## 6. cp -r 目录名称 目录拷贝的目标位置 : 拷贝目录 (改) , -r代表递归拷贝

注意: cp命令不仅可以拷贝目录还可以拷贝文件, 压缩包等, 拷贝文件和压缩包时不 用写-r递归

## 7. rm [-rf] 目录 : 删除目录 (删)

注意: rm不仅可以删除目录, 也可以删除其他文件或压缩包, 为了增强大家的记忆, 无论删除任何目录或文件, 都直接使用 rm -rf 目录/文件/压缩包

## 4.3 文件的操作命令 (增删改查)

### 1. touch 文件名称 : 文件的创建 (增)

### 2. cat/more/less/tail 文件名称 文件的查看 (查)

- o cat : 只能显示最后一屏内容
- o more : 可以显示百分比, 回车可以向下一行, 空格可以向一页, q可以退出查看
- o less : 可以使用键盘上的PgUp和PgDn向上 和向下翻页, q结束查看
- o tail-10 : 查看文件的后10行, Ctrl+C结束

注意: 命令 tail -f 文件 可以对某个文件进行动态监控, 例如tomcat的日志文件, 会随着程序的运行, 日志会变化, 可以使用tail -f catalina-2016-11-11.log 监控文件的变化

### 3. vim 文件 : 修改文件的内容 (改)

vim编辑器是Linux中的强大组件, 是vi编辑器的加强版, vim编辑器的命令和快捷方式有很多, 但此处不一一阐述, 大家也无需研究的很透彻, 使用vim编辑修改文件的方式基本会使用就可以了。

在实际开发中, 使用vim编辑器主要作用就是修改配置文件, 下面是一般步骤:

vim 文件----->进入文件----->命令模式----->按i进入编辑模式----->编辑文件 ----->按Esc进入底行模式----->输入:wq/q! (输入wq代表写入内容并退出, 即保存; 输入q!代表强制退出不保存。)

### 4. rm -rf 文件 : 删除文件 (删)

同目录删除: 熟记 rm -rf 文件 即可

## 4.4 压缩文件的操作命令

### 1) 打包并压缩文件:

Linux中的打包文件一般是以.tar结尾的, 压缩的命令一般是以.gz结尾的。

而一般情况下打包和压缩是一起进行的, 打包并压缩后的文件的后缀名一般.tar.gz。命令: tar -zcvf 打包压缩后的文件名 要打包压缩的文件 其中:

z: 调用gzip压缩命令进行压缩

c: 打包文件

v: 显示运行过程

f: 指定文件名

比如: 加入test目录下有三个文件分别是 aaa.txt bbb.txt ccc.txt, 如果我们要打包test目录并指定压缩后的压缩包名称为 test.tar.gz可以使用命令: tar -zcvf test.tar.gz aaa.txt bbb.txt ccc.txt 或: tar -zcvf test.tar.gz /test/

### 2) 解压压缩包:

命令: tar [-xvf] 压缩文件

其中: x: 代表解压

示例:

1 将/test下的test.tar.gz解压到当前目录下可以使用命令: `tar -xvf test.tar.gz`

2 将/test下的test.tar.gz解压到根目录/usr下: `tar -xvf xxx.tar.gz -C /usr` (- C代表指定解压的位置)

## 4.5 Linux的权限命令

操作系统中每个文件都拥有特定的权限、所属用户和所属组。权限是操作系统用来限制资源访问的机制，在Linux中权限一般分为读(readable)、写(writable)和执行(exutable)，分为三组。分别对应文件的属主(owner)，属组(group)和其他用户(other)，通过这样的机制来限制哪些用户、哪些组可以对特定的文件进行什么样的操作。通过 `ls -l` 命令我们可以查看某个目录下的文件或目录的权限

示例：在随意某个目录下 `ls -l`

```
-rw-r--r--. 1 root root 128 Jun 14 23:04 aaa.txt
-rw-r--r--. 1 root root 0 Jun 14 23:03 bbb.txt
dr-xr-xr-x. 2 root root 36864 Jun 14 18:08 bin
-rw-r--r--. 1 root root 0 Jun 14 23:04 ccc.txt
drwxr-xr-x. 2 root root 4096 Sep 23 2011 etc
drwxr-xr-x. 2 root root 4096 Sep 23 2011 games
drwxr-xr-x. 35 root root 4096 Jun 14 17:20 include
dr-xr-xr-x. 80 root root 36864 Jun 14 18:08 lib
drwxr-xr-x. 17 root root 4096 Jun 14 18:08 libexec
drwxr-xr-x. 11 root root 4096 Jun 14 17:17 local
dr-xr-xr-x. 2 root root 12288 Jun 14 18:08 sbin
drwxr-xr-x. 125 root root 4096 Jun 14 17:21 share
drwxr-xr-x. 4 root root 4096 Jun 14 17:17 src
lrwxrwxrwx. 1 root root 10 Jun 14 17:17 tmp -> ../../var/tmp
```

第一列的内容的信息解释如下：



下面将详细讲解文件的类型、Linux中权限以及文件所有者、所在组、其它组具体是什么？

文件的类型：

- d: 代表目录
- -: 代表文件
- l: 代表链接（可以认为是window中的快捷方式）

Linux中权限分为以下几种：

- r: 代表权限是可读，r也可以用数字4表示

- w: 代表权限是可写, w也可以用数字2表示
- x: 代表权限是可执行, x也可以用数字1表示

#### 文件和目录权限的区别:

对文件和目录而言, 读写执行表示不同的意义。

对于文件:

| 权限名称 | 可执行操作          |
|------|----------------|
| r    | 可以使用cat查看文件的内容 |
| w    | 可以修改文件的内容      |
| x    | 可以将其运行为二进制文件   |

对于目录:

| 权限名称 | 可执行操作        |
|------|--------------|
| r    | 可以查看目录下列表    |
| w    | 可以创建和删除目录下文件 |
| x    | 可以使用cd进入目录   |

在linux中的每个用户必须属于一个组, 不能独立于组外。在linux中每个文件有所有者、所在组、其它组的概念。

#### • 所有者

一般为文件的创建者, 谁创建了该文件, 就天然的成为该文件的所有者, 用ls -ahl命令可以看到文件的所有者 也可以使用chown 用户名 文件名来修改文件的所有者。

#### • 文件所在组

当某个用户创建了一个文件后, 这个文件的所在组就是该用户所在的组 用ls -ahl命令可以看到文件的所有组 也可以使用chgrp 组名 文件名来修改文件所在的组。

#### • 其它组

除开文件的所有者和所在组的用户外, 系统的其它用户都是文件的其它组

我们再来看看如何修改文件/目录的权限。

修改文件/目录的权限的命令: chmod

示例: 修改/test下的aaa.txt的权限为属主有全部权限, 属主所在的组有读写权限, 其他用户只有读的权限

```
chmod u=rwx,g=rw,o=r aaa.txt
```

```
[root@CentOS test]# ll
total 8
-rw-r--r--. 1 root root 2237 Jun 14 23:24 aaa.txt
-rw-r--r--. 1 root root 0 Jun 14 23:03 bbb.txt
-rw-r--r--. 1 root root 0 Jun 14 23:04 ccc.txt
-rw-r--r--. 1 root root 308 Jun 14 23:05 xxx.tar.gz
[root@CentOS test]# chmod u=rwx,g=rw,o=r aaa.txt
[root@CentOS test]# ll
total 8
-rwxrw-r--. 1 root root 2237 Jun 14 23:24 aaa.txt
-rw-r--r--. 1 root root 0 Jun 14 23:03 bbb.txt
-rw-r--r--. 1 root root 0 Jun 14 23:04 ccc.txt
-rw-r--r--. 1 root root 308 Jun 14 23:05 xxx.tar.gz
[root@CentOS test]# _
```

上述示例还可以使用数字表示：

```
chmod 764 aaa.txt
```

补充一个比较常用的东西：

假如我们装了一个zookeeper，我们每次开机到要求其自动启动该怎么办？

1. 新建一个脚本zookeeper
2. 为新建的脚本zookeeper添加可执行权限，命令是：chmod +x zookeeper
3. 把zookeeper这个脚本添加到开机启动项里面，命令是：chkconfig --add zookeeper
4. 如果想看看是否添加成功，命令是：chkconfig --list

## 4.6 Linux 用户管理

Linux系统是一个多用户多任务的分时操作系统，任何一个要使用系统资源的用户，都必须首先向系统管理员申请一个账号，然后以这个账号的身份进入系统。

用户的账号一方面可以帮助系统管理员对使用的用户进行跟踪，并控制他们对系统资源的访问；另一方面也可以帮助用户组织文件，并为用户提供安全性保护。

**Linux用户管理相关命令：**

- useradd 选项 用户名 :添加用户账号
- userdel 选项 用户名 :删除用户帐号
- usermod 选项 用户名 :修改帐号
- passwd 用户名 :更改或创建用户的密码
- passwd -S 用户名 :显示用户账号密码信息
- passwd -d 用户名 :清除用户密码

useradd命令用于Linux中创建新的系统用户。useradd可用来建立用户帐号。帐号建好之后，再用passwd设定帐号的密码。而可用userdel删除帐号。使用useradd指令所建立的帐号，实际上是保存在/etc/passwd文本文件中。

passwd命令用于设置用户的认证信息，包括用户密码、密码过期时间等。系统管理者则能用它管理系统用户的密码。只有管理者可以指定用户名，一般用户只能变更自己的密码。

## 4.7 Linux系统用户组的管理

每个用户都有一个用户组，系统可以对一个用户组中的所有用户进行集中管理。不同Linux 系统对用户组的规定有所不同，如Linux下的用户属于与它同名的用户组，这个用户组在创建用户时同时创建。

用户组的管理涉及用户组的添加、删除和修改。组的增加、删除和修改实际上就是对/etc/group文件的更新。

### Linux系统用户组的管理相关命令:

- `groupadd` 选项 用户组 :增加一个新的用户组
- `groupdel` 用户组 :要删除一个已有的用户组
- `groupmod` 选项 用户组 :修改用户组的属性

## 4.8 其他常用命令

- `pwd` : 显示当前所在位置
- `grep` 要搜索的字符串 要搜索的文件 `--color` : 搜索命令, `--color`代表高亮显示
- `ps -ef` / `ps aux` : 这两个命令都是查看当前系统正在运行进程, 两者的区别是展示格式不同。如果想要查看特定的进程可以使用这样的格式: `ps aux|grep redis` (查看包括redis字符串的进程)

注意: 如果直接用`ps` ( (Process Status) ) 命令, 会显示所有进程的状态, 通常结合`grep`命令查看某进程的状态。

- `kill -9` 进程的`pid` : 杀死进程 (-9 表示强制终止。)

先用`ps`查找进程, 然后用`kill`杀掉

- **网络通信命令:**
  - 查看当前系统的网卡信息: `ifconfig`
  - 查看与某台机器的连接情况: `ping`
  - 查看当前系统的端口使用: `netstat -an`
- `shutdown` : `shutdown -h now` : 指定现在立即关机; `shutdown +5 "System will shutdown after 5 minutes"` :指定5分钟后关机, 同时送出警告信息给登入用户。
- `reboot` : `reboot` : 重开机。 `reboot -w` : 做个重开机的模拟 (只有纪录并不会真的重开机) 。

## Spring相关教程/资料：

### 官网相关

[Spring官网](#)

[Spring系列主要项目](#)

从配置到安全性，Web应用到大数据 - 无论您的应用程序的基础架构需求如何，都有一个Spring Project来帮助您构建它。从小处着手，根据需要使用 - Spring是通过设计模块化的。

[Spring官网指南](#)

无论您在构建什么，这些指南都旨在尽可能快地提高您的工作效率 - 使用Spring团队推荐的最新Spring项目发布和技术。

[Spring官方文档翻译（1~6章）](#)

### 系统学习教程：

#### 文档：

[极客学院Spring Wiki](#)

[Spring W3Cschool教程](#)

#### 视频：

[网易云课堂——58集精通java教程Spring框架开发](#)

[慕课网相关视频](#)

**黑马视频（非常推荐）：**微信公众号：“Java面试通关手册”后台回复“资源分享第一波”免费领取。

### 一些常用的东西

[Spring Framework 4.3.17.RELEASE API](#)

默认浏览器打开，当需要查某个类的作用的时候，可以在浏览器通过ctrl+f搜索。

## 面试必备知识点

### SpringAOP,IOC实现原理

AOP实现原理、动态代理和静态代理、Spring IOC的初始化过程、IOC原理、自己实现怎么实现一个IOC容器？这些东西都是经常会被问到的。

[自己动手实现的 Spring IOC 和 AOP - 上篇](#)

[自己动手实现的 Spring IOC 和 AOP - 下篇](#)

## AOP:

AOP思想的实现一般都是基于 **代理模式**，在JAVA中一般采用JDK动态代理模式，但是我们都知道，**JDK动态代理模式只能代理接口而不能代理类**。因此，Spring AOP 会这样子来进行切换，因为Spring AOP 同时支持 CGLIB、ASPECTJ、JDK动态代理。

- 如果目标对象的实现类实现了接口，Spring AOP 将会采用 JDK 动态代理来生成 AOP 代理类；
- 如果目标对象的实现类没有实现接口，Spring AOP 将会采用 CGLIB 来生成 AOP 代理类——不过这个选择过程对开发者完全透明、开发者也无需关心。

[※静态代理、JDK动态代理、CGLIB动态代理讲解](#)

我们知道AOP思想的实现一般都是基于 **代理模式**，所以在看下面的文章之前建议先了解一下静态代理以及JDK动态代理、CGLIB动态代理的实现方式。

### [Spring AOP 入门](#)

带你入门的一篇文章。这篇文章主要介绍了AOP中的基本概念：5种类型的通知（Before, After, After-returning, After-throwing, Around）；Spring中对AOP的支持：AOP思想的实现一般都是基于代理模式，在JAVA中一般采用JDK动态代理模式，Spring AOP 同时支持 CGLIB、ASPECTJ、JDK动态代理，

[※Spring AOP 基于AspectJ注解如何实现AOP](#)

**AspectJ**是一个AOP框架，它能够对java代码进行AOP编译（一般在编译期进行），让java代码具有AspectJ的AOP功能（当然需要特殊的编译器），可以说AspectJ是目前实现AOP框架中最成熟，功能最丰富的语言，更幸运的是，AspectJ与java程序完全兼容，几乎是无缝关联，因此对于有java编程基础的工程师，上手和使用都非常容易

Spring注意到AspectJ在AOP的实现方式上依赖于特殊编译器(ajc编译器)，因此Spring很机智回避了这点，转向采用动态代理技术的实现原理来构建Spring AOP的内部机制（动态织入），这是与AspectJ（静态织入）最根本的区别。

[※探秘Spring AOP（慕课网视频，很不错）](#)

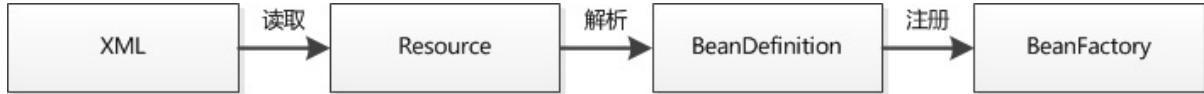
慕课网视频，讲解的很不错，详细且深入

[spring源码剖析（六）AOP实现原理剖析](#)

通过源码分析Spring AOP的原理

## IOC:

Spring IOC的初始化过程：



[\[Spring框架\]Spring IOC的原理及详解。](#)

[Spring IOC核心源码学习](#)

比较简短，推荐阅读。

[Spring IOC 容器源码分析](#)

强烈推荐，内容详尽，而且便于阅读。

## Spring事务管理

[可能是最漂亮的Spring事务管理详解](#)

[Spring编程式和声明式事务实例讲解](#)

## 其他

**Spring单例与线程安全：**

[Spring框架中的单例模式（源码解读）](#)

单例模式是一种常用的软件设计模式。通过单例模式可以保证系统中一个类只有一个实例。spring依赖注入时，使用了多重判断加锁的单例模式。

## Spring源码阅读

阅读源码不仅可以加深我们对Spring设计思想的理解，提高自己的编码水平，还可以让自己在面试中如鱼得水。下面是Github上的一个开源的Spring源码阅读，大家有时间可以看一下，当然你如果有时间也可以自己慢慢研究源码。

### Spring源码阅读

- [spring-core](#)
  - [spring-aop](#)
  - [spring-context](#)
  - [spring-task](#)
  - [spring-transaction](#)
  - [spring-mvc](#)
  - [guava-cache](#)

- 前言
- 一 bean的作用域
  - 1. singleton——唯一 bean 实例
  - 2. prototype——每次请求都会创建一个新的 bean 实例
  - 3. request——每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效
  - 4. session——每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效
  - 5. globalSession
- 二 bean的生命周期
  - initialization 和 destroy
  - 实现\*Aware接口 在Bean中使用Spring框架的一些对象
  - BeanPostProcessor
  - 总结
  - 单例管理的对象
  - 非单例管理的对象
- 三 说明

## 前言

在 Spring 中，那些组成应用程序的主体及由 Spring IOC 容器所管理的对象，被称之为 bean。简单地讲，bean 就是由 IOC 容器初始化、装配及管理的对象，除此之外，bean 就与应用程序中的其他对象没有什么区别了。而 bean 的定义以及 bean 相互间的依赖关系将通过配置元数据来描述。

Spring中的bean默认都是单例的，这些单例Bean在多线程程序下如何保证线程安全呢？例如对于Web应用来说，Web容器对于每个用户请求都创建一个单独的Sevlet线程来处理请求，引入Spring框架之后，每个Action都是单例的，那么对于Spring托管的单例Service Bean，如何保证其安全呢？Spring的单例是基于BeanFactory也就是Spring容器的，单例Bean在此容器内只有一个，Java的单例是基于 JVM，每个 JVM 内只有一个实例。

## 一 bean的作用域

创建一个bean定义，其实质是用该bean定义对应的类来创建真正实例的“配方”。把bean定义看成一个配方很有意义，它与class很类似，只根据一张“处方”就可以创建多个实例。不仅可以控制注入到对象中的各种依赖和配置值，还可以控制该对象的作用域。这样可以灵活选择所建对象的作用域，而不必在Java Class级定义作用域。Spring Framework支持五种作用域，分别阐述如下表。

| 类别            | 说明                                                                                       |
|---------------|------------------------------------------------------------------------------------------|
| singleton     | 在Spring IoC容器中仅存在一个Bean实例，Bean以单例方式存在，默认值                                                |
| prototype     | 每次从容器中调用 Bean时，都返回一个新的实例，即每次调用 <code>getBean()</code> 时，相当于执行 <code>new XxxBean()</code> |
| request       | 每次HTTP请求都会创建一个新的Bean，该作用域仅适用于 <code>WebApplicationContext</code> 环境                      |
| session       | 同一个HTTP Session 共享一个Bean，不同Session使用不同Bean，仅适用于 <code>WebApplicationContext</code> 环境    |
| globalSession | 一般用于 <code>Portlet</code> 应用环境，该作用域仅适用于 <code>WebApplicationContext</code> 环境            |

五种作用域中，**request**、**session** 和 **global session** 三种作用域仅在基于web的应用中使用（不必关心你所采用的是什么web应用框架），只能用在基于 web 的 Spring ApplicationContext 环境。

## 1. singleton——唯一 bean 实例

当一个 bean 的作用域为 **singleton**，那么Spring IoC容器中只会存在一个共享的 bean 实例，并且所有对 bean 的请求，只要 id 与该 bean 定义相匹配，则只会返回bean的同一实例。 singleton 是单例类型(对应于单例模式)，就是在创建起容器时就同时自动创建了一个bean的对象，不管你是否使用，他都存在了，每次获取到的对象都是同一个对象。注意，singleton 作用域是Spring中的缺省作用域。要在XML中将 bean 定义成 singleton，可以这样配置：

```
<bean id="ServiceImpl" class="cn.csdn.service.ServiceImpl" scope="singleton">
```

也可以通过 `@Scope` 注解（它可以显示指定bean的作用范围。）的方式

```
@Service
@Scope("singleton")
public class ServiceImpl{}
```

## 2. prototype——每次请求都会创建一个新的 bean 实例

当一个bean的作用域为 **prototype**，表示一个 bean 定义对应多个对象实例。**prototype** 作用域的 bean 会导致在每次对该 bean 请求（将其注入到另一个 bean 中，或者以程序的方式调用容器的 `getBean()` 方法）时都会创建一个新的 bean 实例。**prototype** 是原型类型，它在我们创建容器的时候并没有实例化，而是当我们获取bean的时候才会去创建一个对象，而且我们每次获取到的对象都不是同一个对象。根据经验，对有状态的 bean 应该使用 **prototype** 作用域，而对无状态的 bean 则应该使用 **singleton** 作用域。在 XML 中将 bean 定义成 prototype，可以这样配置：

```
<bean id="account" class="com.foo.DefaultAccount" scope="prototype"/>
或者
<bean id="account" class="com.foo.DefaultAccount" singleton="false"/>
```

通过 `@Scope` 注解的方式实现就不做演示了。

## 3. request——每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效

**request**只适用于Web程序，每一次 HTTP 请求都会产生一个新的bean，同时该bean仅在当前HTTP request内有效，当请求结束后，该对象的生命周期即告结束。在 XML 中将 bean 定义成 prototype，可以这样配置：

```
<bean id="loginAction" class="cn.csdn.LoginAction" scope="request"/>
```

## 4. session——每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效

**session**只适用于Web程序，**session** 作用域表示该针对每一次 HTTP 请求都会产生一个新的 bean，同时该 bean 仅在当前 HTTP session 内有效.与**request**作用域一样，可以根据需要放心的更改所创建实例的内部状态，而别的 HTTP session 中根据 `userPreferences` 创建的实例，将不会看到这些特定于某个 HTTP session 的状态变化。当HTTP session最终被废弃的时候，在该HTTP session作用域内的bean也会被废弃掉。

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

## 5. globalSession

global session 作用域类似于标准的 HTTP session 作用域，不过仅仅在基于 portlet 的 web 应用中才有意义。Portlet 规范定义了全局 Session 的概念，它被所有构成某个 portlet web 应用的各种不同的 portlet 所共享。在 global session 作用域中定义的 bean 被限定于全局 portlet Session 的生命周期范围内。

```
<bean id="user" class="com.foo.Preferences" scope="globalSession"/>
```

# 二 bean的生命周期

Spring Bean 是 Spring 应用中最最重要的部分了。所以来看看 Spring 容器在初始化一个 bean 的时候会做那些事情，顺序是怎样的，在容器关闭的时候，又会做哪些事情。

spring 版本：4.2.3.RELEASE 鉴于 Spring 源码是用 gradle 构建的，我也决定舍弃我大 maven，尝试下洪菊推荐过的 gradle。运行 beanLifeCycle 模块下的 junit test 即可在控制台看到如下输出，可以清楚了解 Spring 容器在创建，初始化和销毁 Bean 的时候依次做了那些事情。

```
Spring容器初始化
=====
调用GiraffeService无参构造函数
GiraffeService中利用set方法设置属性值
调用setBeanName:: Bean Name defined in context=giraffeService
调用setBeanClassLoader,ClassLoader Name = sun.misc.Launcher$AppClassLoader
调用setBeanFactory, setBeanFactory:: giraffe bean singleton=true
调用setEnvironment
调用setResourceLoader:: Resource File Name=spring-beans.xml
调用setApplicationEventPublisher
调用setApplicationContext:: Bean Definition Names=[giraffeService, org.springframework.context.annotation.CommonAnnotationBeanPostProcessor#0, com.giraffe.spring.service.GiraffeServicePostProcessor#0]
执行BeanPostProcessor的postProcessBeforeInitialization方法,beanName=giraffeService
调用PostConstruct注解标注的方法
执行InitializingBean接口的afterPropertiesSet方法
执行配置的init-method
执行BeanPostProcessor的postProcessAfterInitialization方法,beanName=giraffeService
Spring容器初始化完毕
=====
从容器中获取Bean
giraffe Name=李光洙
=====
调用preDestroy注解标注的方法
执行DisposableBean接口的destroy方法
执行配置的destroy-method
Spring容器关闭
```

先来看看，Spring 在 Bean 从创建到销毁的生命周期中可能做得事情。

## initialization 和 destroy

有时我们需要在 Bean 属性值 set 好之后和 Bean 销毁之前做一些事情，比如检查 Bean 中某个属性是否被正常的设置好值了。Spring 框架提供了多种方法让我们可以在 Spring Bean 的生命周期中执行 initialization 和 pre-destroy 方法。

### 1. 实现 InitializingBean 和 DisposableBean 接口

这两个接口都只包含一个方法。通过实现 InitializingBean 接口的 afterPropertiesSet() 方法可以在 Bean 属性值设置好之后做一些操作，实现 DisposableBean 接口的 destroy() 方法可以在销毁 Bean 之前做一些操作。

例子如下：

```

public class GiraffeService implements InitializingBean,DisposableBean {
 @Override
 public void afterPropertiesSet() throws Exception {
 System.out.println("执行InitializingBean接口的afterPropertiesSet方法");
 }
 @Override
 public void destroy() throws Exception {
 System.out.println("执行DisposableBean接口的destroy方法");
 }
}

```

这种方法比较简单，但是不建议使用。因为这样会将Bean的实现和Spring框架耦合在一起。

## 2.在bean的配置文件中指定init-method和destroy-method方法

Spring允许我们创建自己的 init 方法和 destroy 方法，只要在 Bean 的配置文件中指定 init-method 和 destroy-method 的值就可以在 Bean 初始化时和销毁之前执行一些操作。

例子如下：

```

public class GiraffeService {
 //通过<bean>的destroy-method属性指定的销毁方法
 public void destroyMethod() throws Exception {
 System.out.println("执行配置的destroy-method");
 }
 //通过<bean>的init-method属性指定的初始化方法
 public void initMethod() throws Exception {
 System.out.println("执行配置的init-method");
 }
}

```

配置文件中的配置：

```

<bean name="giraffeService" class="com.giraffe.spring.service.GiraffeService" init-method="initMethod" destroy-
method="destroyMethod">
</bean>

```

需要注意的是自定义的init-method和post-method方法可以抛异常但是不能有参数。

这种方式比较推荐，因为可以自己创建方法，无需将Bean的实现直接依赖于spring的框架。

## 3.使用@PostConstruct和@PreDestroy注解

除了xml配置的方式，Spring 也支持用 `@PostConstruct` 和 `@PreDestroy` 注解来指定 `init` 和 `destroy` 方法。这两个注解均在 `javax.annotation` 包中。为了注解可以生效，需要在配置文件中定义 `org.springframework.context.annotation.CommonAnnotationBeanPostProcessor` 或 `context:annotation-config`

例子如下：

```

public class GiraffeService {
 @PostConstruct
 public void initPostConstruct(){
 System.out.println("执行PostConstruct注解标注的方法");
 }
 @PreDestroy
 public void preDestroy(){
 System.out.println("执行preDestroy注解标注的方法");
 }
}

```

配置文件：

```
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor" />
```

## 实现\*Aware接口 在Bean中使用Spring框架的一些对象

有些时候我们需要在 Bean 的初始化中使用 Spring 框架自身的一些对象来执行一些操作，比如获取 ServletContext 的一些参数，获取 ApplicationContext 中的 BeanDefinition 的名字，获取 Bean 在容器中的名字等等。为了让 Bean 可以获取到框架自身的一些对象，Spring 提供了一组名为\*Aware的接口。

这些接口均继承于 org.springframework.beans.factory.Aware 标记接口，并提供一个将由 Bean 实现的set\*方法，Spring 通过基于setter的依赖注入方式使相应的对象可以被 Bean 使用。网上说，这些接口是利用观察者模式实现的，类似于 servlet listeners，目前还不明白，不过这也不在本文的讨论范围内。介绍一些重要的 Aware 接口：

- **ApplicationContextAware**: 获得 ApplicationContext 对象，可以用来获取所有 Bean definition 的名字。
- **BeanFactoryAware**: 获得 BeanFactory 对象，可以用来检测 Bean 的作用域。
- **BeanNameAware**: 获得 Bean 在配置文件中定义的名字。
- **ResourceLoaderAware**: 获得 ResourceLoader 对象，可以获得 classpath 中某个文件。
- **ServletContextAware**: 在一个 MVC 应用中可以获取 ServletContext 对象，可以读取 context 中的参数。
- **ServletConfigAware**: 在一个 MVC 应用中可以获取 ServletConfig 对象，可以读取 config 中的参数。

```
public class GiraffeService implements ApplicationContextAware,
 ApplicationEventPublisherAware, BeanClassLoaderAware, BeanFactoryAware,
 BeanNameAware, EnvironmentAware, ImportAware, ResourceLoaderAware{
 @Override
 public void setBeanClassLoader(ClassLoader classLoader) {
 System.out.println("执行setBeanClassLoader, ClassLoader Name = " + classLoader.getName());
 }
 @Override
 public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
 System.out.println("执行setBeanFactory, setBeanFactory:: giraffe bean singleton=" + beanFactory.isSingleton("giraffeService"));
 }
 @Override
 public void setBeanName(String s) {
 System.out.println("执行setBeanName:: Bean Name defined in context="
 + s);
 }
 @Override
 public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
 System.out.println("执行setApplicationContext:: Bean Definition Names="
 + Arrays.toString(applicationContext.getBeanDefinitionNames()));
 }
 @Override
 public void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher) {
 System.out.println("执行setApplicationEventPublisher");
 }
 @Override
 public void setEnvironment(Environment environment) {
 System.out.println("执行setEnvironment");
 }
 @Override
 public void setResourceLoader(ResourceLoader resourceLoader) {
 Resource resource = resourceLoader.getResource("classpath:spring-beans.xml");
 System.out.println("执行setResourceLoader:: Resource File Name="
 + resource.getFilename());
 }
 @Override
 public void setImportMetadata(AnnotationMetadata annotationMetadata) {
 System.out.println("执行setImportMetadata");
 }
}
```

## BeanPostProcessor

上面的\*Aware接口是针对某个实现这些接口的Bean定制初始化的过程，Spring同样可以针对容器中的所有Bean，或者某些Bean定制初始化过程，只需提供一个实现BeanPostProcessor接口的类即可。该接口中包含两个方法，postProcessBeforeInitialization和postProcessAfterInitialization。postProcessBeforeInitialization方法会在容器中的Bean初始化之前执行，postProcessAfterInitialization方法在容器中的Bean初始化之后执行。

例子如下：

```
public class CustomerBeanPostProcessor implements BeanPostProcessor {
 @Override
 public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
 System.out.println("执行BeanPostProcessor的postProcessBeforeInitialization方法,beanName=" + beanName);
 return bean;
 }
 @Override
 public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
 System.out.println("执行BeanPostProcessor的postProcessAfterInitialization方法,beanName=" + beanName);
 return bean;
 }
}
```

要将BeanPostProcessor的Bean像其他Bean一样定义在配置文件中

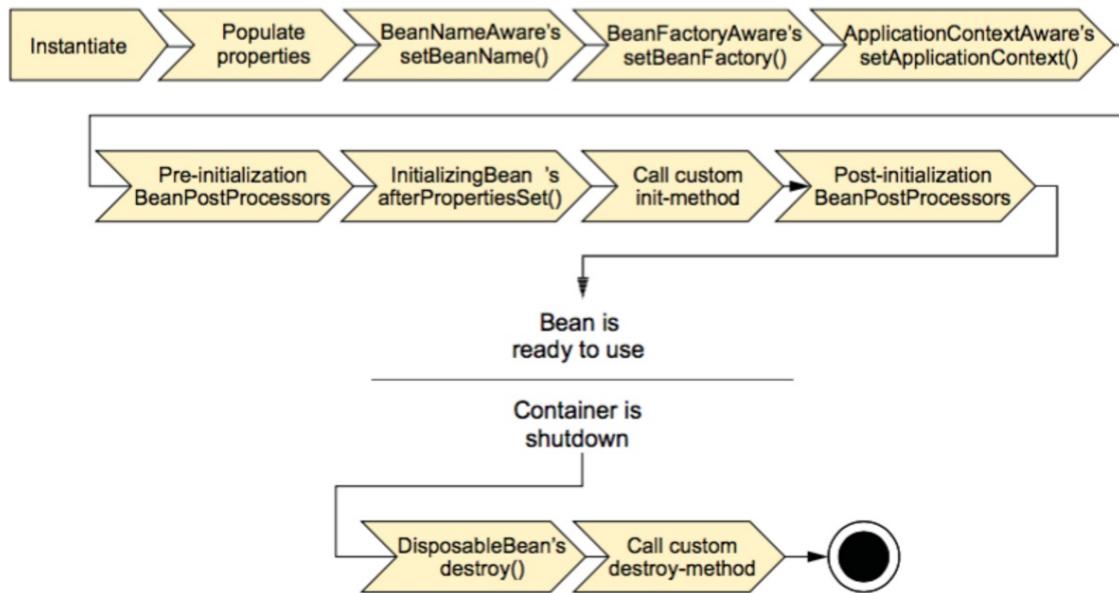
```
<bean class="com.giraffe.spring.service.CustomerBeanPostProcessor"/>
```

## 总结

所以。。。结合第一节控制台输出的内容，Spring Bean的生命周期是这样纸的：

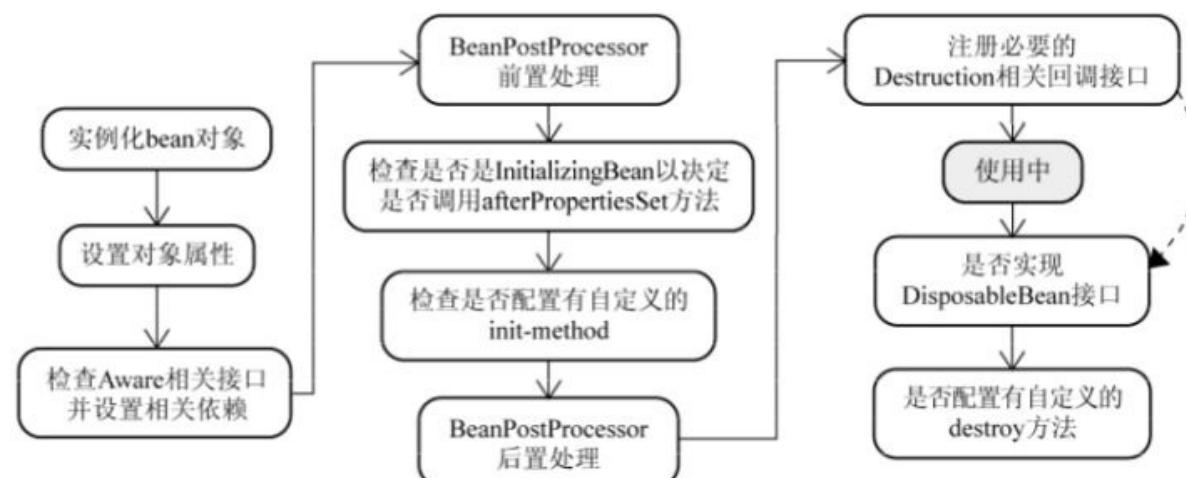
- Bean容器找到配置文件中 Spring Bean 的定义。
- Bean容器利用Java Reflection API创建一个Bean的实例。
- 如果涉及到一些属性值 利用set方法设置一些属性值。
- 如果Bean实现了BeanNameAware接口，调用setBeanName()方法，传入Bean的名字。
- 如果Bean实现了BeanClassLoaderAware接口，调用setBeanClassLoader()方法，传入ClassLoader对象的实例。
- 如果Bean实现了BeanFactoryAware接口，调用setBeanFactory()方法，传入BeanFactory对象的实例。
- 与上面的类似，如果实现了其他\*Aware接口，就调用相应的方法。
- 如果有和加载这个Bean的Spring容器相关的BeanPostProcessor对象，执行postProcessBeforeInitialization()方法
- 如果Bean实现了InitializingBean接口，执行afterPropertiesSet()方法。
- 如果Bean在配置文件中的定义包含init-method属性，执行指定的方法。
- 如果有和加载这个Bean的Spring容器相关的BeanPostProcessor对象，执行postProcessAfterInitialization()方法
- 当要销毁Bean的时候，如果Bean实现了DisposableBean接口，执行destroy()方法。
- 当要销毁Bean的时候，如果Bean在配置文件中的定义包含destroy-method属性，执行指定的方法。

用图表示一下(图来源:<http://www.jianshu.com/p/d00539babca5>)：



**Figure 1.5** A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.

与之比较类似的中文版本：



其实很多时候我们并不会真的去实现上面说描述的那些接口，那么下面我们就除去那些接口，针对bean的单例和非单例来描述下bean的生命周期：

## 单例管理的对象

当scope="singleton"，即默认情况下，会在启动容器时（即实例化容器时）时实例化。但我们可以指定Bean节点的lazy-init="true"来延迟初始化bean，这时候，只有在第一次获取bean时才会初始化bean，即第一次请求该bean时才初始化。如下配置：

```
<bean id="ServiceImpl" class="cn.csdn.service.ServiceImpl" lazy-init="true"/>
```

如果想对所有的默认单例bean都应用延迟初始化，可以在根节点beans设置default-lazy-init属性为true，如下所示：

```
<beans default-lazy-init="true" ...>
```

默认情况下，Spring 在读取 xml 文件的时候，就会创建对象。在创建对象的时候先调用构造器，然后调用 init-method 属性值中所指定的方法。对象在被销毁的时候，会调用 destroy-method 属性值中所指定的方法（例如调用 Container.destroy()方法的时候）。写一个测试类，代码如下：

```
public class LifeBean {
 private String name;

 public LifeBean(){
 System.out.println("LifeBean()构造函数");
 }
 public String getName() {
 return name;
 }

 public void setName(String name) {
 System.out.println("setName()");
 this.name = name;
 }

 public void init(){
 System.out.println("this is init of lifeBean");
 }

 public void destroy(){
 System.out.println("this is destroy of lifeBean " + this);
 }
}
```

life.xml配置如下：

```
<bean id="life_singleton" class="com.bean.LifeBean" scope="singleton"
 init-method="init" destroy-method="destroy" lazy-init="true"/>
```

测试代码：

```
public class LifeTest {
 @Test
 public void test() {
 AbstractApplicationContext container =
 new ClassPathXmlApplicationContext("life.xml");
 LifeBean life1 = (LifeBean)container.getBean("life");
 System.out.println(life1);
 container.close();
 }
}
```

运行结果：

```
LifeBean()构造函数
this is init of lifeBean
com.bean.LifeBean@573f2bb1
...
this is destroy of lifeBean com.bean.LifeBean@573f2bb1
```

## 非单例管理的对象

当 scope="prototype" 时，容器也会延迟初始化 bean，Spring 读取xml文件的时候，并不会立刻创建对象，而是在第一次请求该 bean 时才初始化（如调用getBean方法时）。在第一次请求每一个 prototype 的bean 时，Spring容器都会调用其构造器创建这个对象，然后调用 init-method 属性值中所指定的方法。对象销毁的时候，Spring 容器不会帮我们调

用任何方法，因为是非单例，这个类型的对象有很多个，Spring容器一旦把这个对象交给你之后，就不再管理这个对象了。

为了测试prototype bean的生命周期life.xml配置如下：

```
<bean id="life_prototype" class="com.bean.LifeBean" scope="prototype" init-method="init" destroy-method="destory"/>
```

测试程序：

```
public class LifeTest {
 @Test
 public void test() {
 AbstractApplicationContext container = new ClassPathXmlApplicationContext("life.xml");
 LifeBean life1 = (LifeBean)container.getBean("life_singleton");
 System.out.println(life1);

 LifeBean life3 = (LifeBean)container.getBean("life_prototype");
 System.out.println(life3);
 container.close();
 }
}
```

运行结果：

```
LifeBean()构造函数
this is init of lifeBean
com.bean.LifeBean@573f2bb1
LifeBean()构造函数
this is init of lifeBean
com.bean.LifeBean@5ae9a829
...
this is destroy of lifeBean com.bean.LifeBean@573f2bb1
```

可以发现，对于作用域为 prototype 的 bean，其 `destroy` 方法并没有被调用。如果 bean 的 scope 设为 `prototype` 时，当容器关闭时，`destroy` 方法不会被调用。对于 `prototype` 作用域的 bean，有一点非常重要，那就是 Spring 不能对一个 `prototype` bean 的整个生命周期负责：容器在初始化、配置、装饰或者是装配完一个 `prototype` 实例后，将它交给客户端，随后就对该 `prototype` 实例不闻不问了。不管何种作用域，容器都会调用所有对象的初始化生命周期回调方法。但对 `prototype` 而言，任何配置好的析构生命周期回调方法都将不会被调用。清除 `prototype` 作用域的对象并释放任何 `prototype` bean 所持有的昂贵资源，都是客户端代码的职责（让 Spring 容器释放被 `prototype` 作用域 bean 占用资源的一种可行方式是，通过使用 bean 的后置处理器，该处理器持有要被清除的 bean 的引用）。谈及 `prototype` 作用域的 bean 时，在某些方面你可以将 Spring 容器的角色看作是 Java `new` 操作的替代者，任何迟于该时间点的生命周期事宜都得交由客户端来处理。

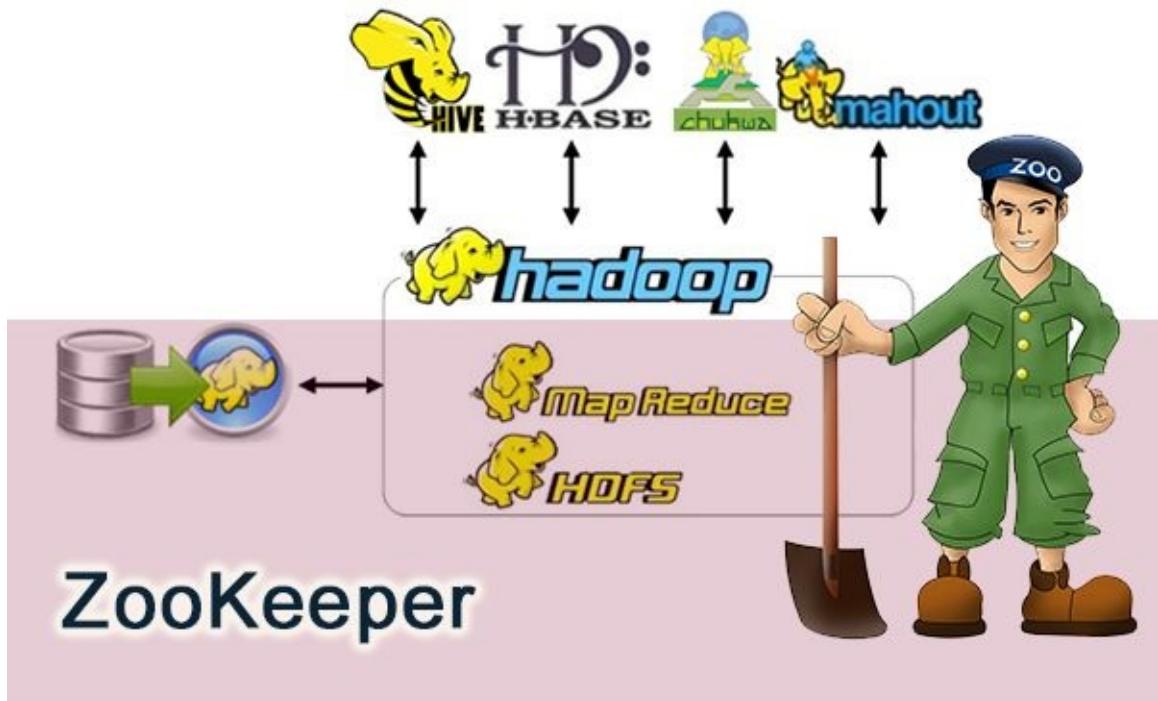
Spring 容器可以管理 `singleton` 作用域下 bean 的生命周期，在此作用域下，Spring 能够精确地知道 bean 何时被创建，何时初始化完成，以及何时被销毁。而对于 `prototype` 作用域的 bean，Spring 只负责创建，当容器创建了 bean 的实例后，bean 的实例就交给了客户端的代码管理，Spring 容器将不再跟踪其生命周期，并且不会管理那些被配置成 `prototype` 作用域的 bean 的生命周期。

## 三 说明

本文的完成结合了下面两篇文章，并做了相应修改：

- <https://blog.csdn.net/fuzhongmin05/article/details/73389779>
- <https://yemengying.com/2016/07/14/spring-bean-life-cycle/>

由于本文非本人独立原创，所以未声明为原创！在此说明！



## 前言

相信大家对 ZooKeeper 应该不算陌生。但是你真的了解 ZooKeeper 是个什么东西吗？如果别人/面试官让你给他讲讲 ZooKeeper 是个什么东西，你能回答到什么地步呢？

我本人曾经使用过 ZooKeeper 作为 Dubbo 的注册中心，另外在搭建 solr 集群的时候，我使用到了 ZooKeeper 作为 solr 集群的管理工具。前几天，总结项目经验的时候，我突然问自己 ZooKeeper 到底是个什么东西？想了半天，脑海中只是简单的能浮现出几句话：“①Zookeeper 可以被用作注册中心。②Zookeeper 是 Hadoop 生态系统的一员；③构建 Zookeeper 集群的时候，使用的服务器最好是奇数台。”可见，我对于 Zookeeper 的理解仅仅是停留在了表面。

所以，通过本文，希望带大家稍微详细的了解一下 ZooKeeper。如果没有学过 ZooKeeper，那么本文将会是你进入 ZooKeeper 大门的垫脚砖。如果你已经接触过 ZooKeeper，那么本文将带你回顾一下 ZooKeeper 的一些基础概念。

最后，本文只涉及 ZooKeeper 的一些概念，并不涉及 ZooKeeper 的使用以及 ZooKeeper 集群的搭建。网上有介绍 ZooKeeper 的使用以及搭建 ZooKeeper 集群的文章，大家有需要可以自行查阅。

## 一 什么是 ZooKeeper

### ZooKeeper 的由来

下面这段内容摘自《从Paxos到Zookeeper》第四章第一节的某段内容，推荐大家阅读以下：

Zookeeper最早起源于雅虎研究院的一个研究小组。在当时，研究人员发现，在雅虎内部很多大型系统基本都需要依赖一个类似的系统来进行分布式协调，但是这些系统往往都存在分布式单点问题。所以，雅虎的开发人员就试图开发一个通用的无单点问题的分布式协调框架，以便让开发人员将精力集中在处理业务逻辑上。

关于“ZooKeeper”这个项目的名字，其实也有一段趣闻。在立项初期，考虑到之前内部很多项目都是使用动物的名字来命名的（例如著名的Pig项目），雅虎的工程师希望给这个项目也取一个动物的名字。时任研究院的首席科学家RaghuRamakrishnan开玩笑地说：“就这样下去，我们这儿就变成动物园了！”此话一出，大家纷纷表示就叫动

物园管理员吧——因为各个以动物命名的分布式组件放在一起，雅虎的整个分布式系统看上去就像一个大型的动物园了，而Zookeeper正好要用来进行分布式环境的协调——于是，Zookeeper的名字也就由此诞生了。

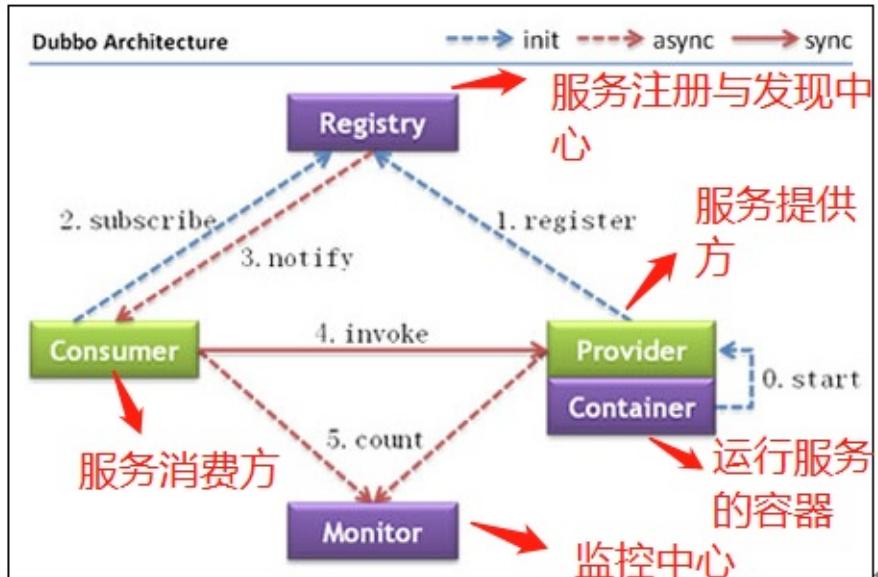
## 1.1 ZooKeeper 概览

ZooKeeper 是一个开源的分布式协调服务，ZooKeeper框架最初是在“Yahoo!”上构建的，用于以简单而稳健的方式访问他们的应用程序。后来，Apache ZooKeeper成为Hadoop，HBase和其他分布式框架使用的有组织服务的标准。例如，Apache HBase使用ZooKeeper跟踪分布式数据的状态。ZooKeeper 的设计目标是将那些复杂且容易出错的分布式一致性服务封装起来，构成一个高效可靠的原语集，并以一系列简单易用的接口提供给用户使用。

原语：操作系统或计算机网络用语范畴。是由若干条指令组成的，用于完成一定功能的一个过程。具有不可分割性·即原语的执行必须是连续的，在执行过程中不允许被中断。

ZooKeeper 是一个典型的分布式数据一致性解决方案，分布式应用程序可以基于 ZooKeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

Zookeeper 一个最常用的使用场景就是用于担任服务生产者和服务消费者的注册中心。服务生产者将自己提供的服务注册到Zookeeper中心，服务的消费者在进行服务调用的时候先到Zookeeper中查找服务，获取到服务生产者的详细信息之后，再去调用服务生产者的内容与数据。如下图所示，在 Dubbo架构中 Zookeeper 就担任了注册中心这一角色。



## 1.2 结合个人使用情况的讲一下 ZooKeeper

在我自己做过的项目中，主要使用到了 ZooKeeper 作为 Dubbo 的注册中心(Dubbo 官方推荐使用 ZooKeeper注册中心)。另外在搭建 solr 集群的时候，我使用 ZooKeeper 作为 solr 集群的管理工具。这时，ZooKeeper 主要提供下面几个功能：1、集群管理：容错、负载均衡。2、配置文件的集中管理3、集群的入口。

我个人觉得在使用 ZooKeeper 的时候，最好是使用 集群版的 ZooKeeper 而不是单机版的。官网给出的架构图就描述的是一个集群版的 ZooKeeper 。通常 3 台服务器就可以构成一个 ZooKeeper 集群了。

为什么最好使用奇数台服务器构成 ZooKeeper 集群？

所谓的zookeeper容错是指，当宕掉几个zookeeper服务器之后，剩下的个数必须大于宕掉的个数的话整个zookeeper才依然可用。假如我们的集群中有n台zookeeper服务器，那么也就是剩下的服务数必须大于n/2。先说一下结论，2n和2n-1的容忍度是一样的，都是n-1，大家可以先自己仔细想一想，这应该是一个很简单的数学问题了。比如假如我们有3台，那么最大允许宕掉1台zookeeper服务器，如果我们有4台的时候也同样只允许宕掉1台。假如我们有5台，那么最大允许宕掉2台zookeeper服务器，如果我们有6台的时候也同样只允许宕掉2台。

综上，何必增加那一个不必要的zookeeper呢？

## 二 关于 ZooKeeper 的一些重要概念

### 2.1 重要概念总结

- ZooKeeper 本身就是一个分布式程序（只要半数以上节点存活，ZooKeeper 就能正常服务）。
- 为了保证高可用，最好是以集群形态来部署 ZooKeeper，这样只要集群中大部分机器是可用的（能够容忍一定的机器故障），那么 ZooKeeper 本身仍然是可用的。
- ZooKeeper 将数据保存在内存中，这也就保证了高吞吐量和低延迟（但是内存限制了能够存储的容量不太大，此限制也是保持znode中存储的数据量较小的进一步原因）。
- ZooKeeper 是高性能的。在“读”多于“写”的应用程序中尤其地高性能，因为“写”会导致所有的服务器间同步状态。（“读”多于“写”是协调服务的典型场景。）
- ZooKeeper 有临时节点的概念。当创建临时节点的客户端会话一直保持活动，瞬时节点就一直存在。而当会话终结时，瞬时节点被删除。持久节点是指一旦这个ZNode被创建了，除非主动进行ZNode的移除操作，否则这个ZNode将一直保存在Zookeeper上。
- ZooKeeper 底层其实只提供了两个功能：①管理（存储、读取）用户程序提交的数据；②为用户程序提交数据节点监听服务。

下面关于会话（Session）、Znode、版本、Watcher、ACL概念的总结都在《从Paxos到Zookeeper》第四章第一节以及第七章第八节有提到，感兴趣的可以看看！

### 2.2 会话（Session）

Session 指的是 ZooKeeper 服务器与客户端会话。在 ZooKeeper 中，一个客户端连接是指客户端和服务器之间的一个 TCP 长连接。客户端启动的时候，首先会与服务器建立一个 TCP 连接，从第一次连接建立开始，客户端会话的生命周期也开始了。通过这个连接，客户端能够通过心跳检测与服务器保持有效的会话，也能够向Zookeeper服务器发送请求并接受响应，同时还能够通过该连接接收来自服务器的Watch事件通知。Session的 sessionTimeout 值用来设置一个客户端会话的超时时间。当由于服务器压力太大、网络故障或是客户端主动断开连接等各种原因导致客户端连接断开时，只要在 sessionTimeout 规定的时间内能够重新连接上集群中任意一台服务器，那么之前创建的会话仍然有效。

在为客户端创建会话之前，服务端首先会为每个客户端都分配一个sessionID。由于 sessionID 是 Zookeeper 会话的一个重要标识，许多与会话相关的运行机制都是基于这个 sessionID 的，因此，无论是哪台服务器为客户端分配的 sessionID，都务必保证全局唯一。

### 2.3 Znode

在谈到分布式的时候，我们通常说的“节点”是指组成集群的每一台机器。然而，在Zookeeper中，“节点”分为两类，第一类同样是指构成集群的机器，我们称之为机器节点；第二类则是指数据模型中的数据单元，我们称之为数据节点——ZNode。

Zookeeper将所有数据存储在内存中，数据模型是一棵树（Znode Tree），由斜杠（/）的进行分割的路径，就是一个Znode，例如/foo/path1。每个上都会保存自己的数据内容，同时还会保存一系列属性信息。

在Zookeeper中，node可以分为持久节点和临时节点两类。所谓持久节点是指一旦这个ZNode被创建了，除非主动进行ZNode的移除操作，否则这个ZNode将一直保存在Zookeeper上。而临时节点就不一样了，它的生命周期和客户端会话绑定，一旦客户端会话失效，那么这个客户端创建的所有临时节点都会被移除。另外，ZooKeeper还允许用户为每个节点添加一个特殊的属性：**SEQUENTIAL**.一旦节点被标记上这个属性，那么在这个节点被创建的时候，Zookeeper会自动在其节点名后面追加上一个整型数字，这个整型数字是一个由父节点维护的自增数字。

### 2.4 版本

在前面我们已经提到，Zookeeper 的每个 ZNode 上都会存储数据，对于每个 ZNode，Zookeeper 都会为其维护一个叫作 **Stat** 的数据结构，Stat 中记录了这个 ZNode 的三个数据版本，分别是 version（当前 ZNode 的版本）、cversion（当前 ZNode 子节点的版本）和 cversion（当前 ZNode 的 ACL 版本）。

## 2.5 Watcher

Watcher（事件监听器），是 Zookeeper 中的一个很重要的特性。Zookeeper 允许用户在指定节点上注册一些 Watcher，并且在一些特定事件触发的时候，ZooKeeper 服务端会将事件通知到感兴趣的客户端上去，该机制是 Zookeeper 实现分布式协调服务的重要特性。

## 2.6 ACL

Zookeeper 采用 ACL（Access Control Lists）策略来进行权限控制，类似于 UNIX 文件系统的权限控制。Zookeeper 定义了如下 5 种权限。

- **CREATE:** 创建子节点的权限。
- **READ:** 获取节点数据和子节点列表的权限。
- **WRITE:** 更新节点数据的权限。
- **DELETE:** 删除子节点的权限。
- **ADMIN:** 设置节点 ACL 的权限。

其中尤其需要注意的是，CREATE 和 DELETE 这两种权限都是针对子节点的权限控制。

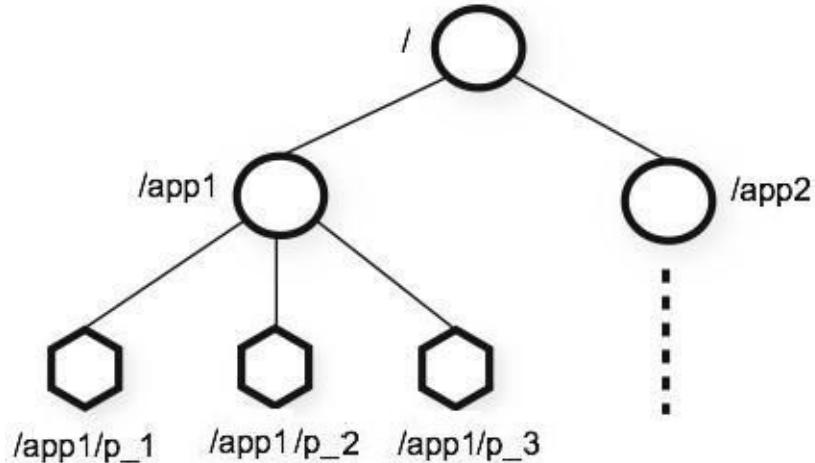
## 三 ZooKeeper 特点

- **顺序一致性：**从同一客户端发起的事务请求，最终将会严格地按照顺序被应用到 ZooKeeper 中去。
- **原子性：**所有事务请求的处理结果在整个集群中所有机器上的应用情况是一致的，也就是说，要么整个集群中所有的机器都成功应用了某一个事务，要么都没有应用。
- **单一系统映像：**无论客户端连到哪一个 ZooKeeper 服务器上，其看到的服务端数据模型都是一致的。
- **可靠性：**一旦一次更改请求被应用，更改的结果就会被持久化，直到被下一次更改覆盖。

## 四 ZooKeeper 设计目标

### 4.1 简单的数据模型

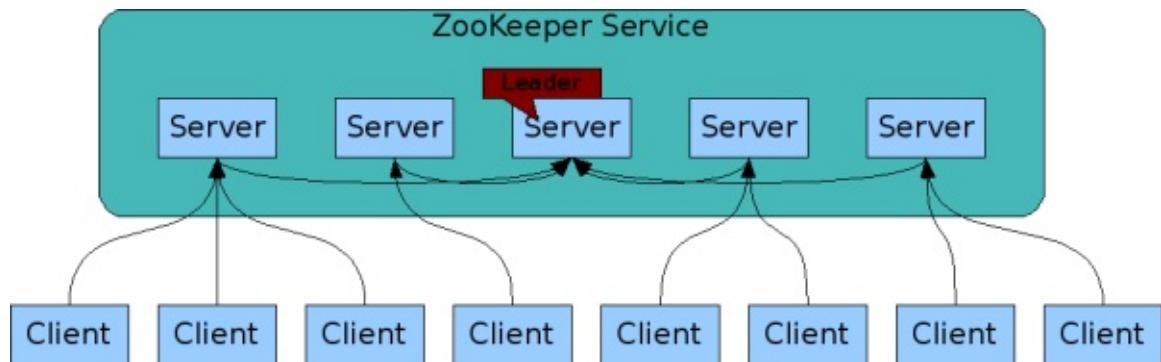
ZooKeeper 允许分布式进程通过共享的层次结构命名空间进行相互协调，这与标准文件系统类似。名称空间由 ZooKeeper 中的数据寄存器组成 - 称为 znode，这些类似于文件和目录。与为存储设计的典型文件系统不同，ZooKeeper 数据保存在内存中，这意味着 ZooKeeper 可以实现高吞吐量和低延迟。



## 4.2 可构建集群

为了保证高可用，最好是以集群形态来部署 ZooKeeper，这样只要集群中大部分机器是可用的（能够容忍一定的机器故障），那么 zookeeper 本身仍然是可用的。客户端在使用 ZooKeeper 时，需要知道集群机器列表，通过与集群中的某一台机器建立 TCP 连接来使用服务，客户端使用这个 TCP 链接来发送请求、获取结果、获取监听事件以及发送心跳包。如果这个连接异常断开了，客户端可以连接到另外的机器上。

ZooKeeper 官方提供的架构图：



上图中每一个 Server 代表一个安装 ZooKeeper 服务的服务器。组成 ZooKeeper 服务的服务器都会在内存中维护当前的服务器状态，并且每台服务器之间都互相保持着通信。集群间通过 Zab 协议（Zookeeper Atomic Broadcast）来保持数据的一致性。

## 4.3 顺序访问

对于来自客户端的每个更新请求，ZooKeeper 都会分配一个全局唯一的递增编号，这个编号反映了所有事务操作的先后顺序，应用程序可以使用 ZooKeeper 这个特性来实现更高层次的同步原语。这个编号也叫做时间戳——**xid** (Zookeeper Transaction Id)

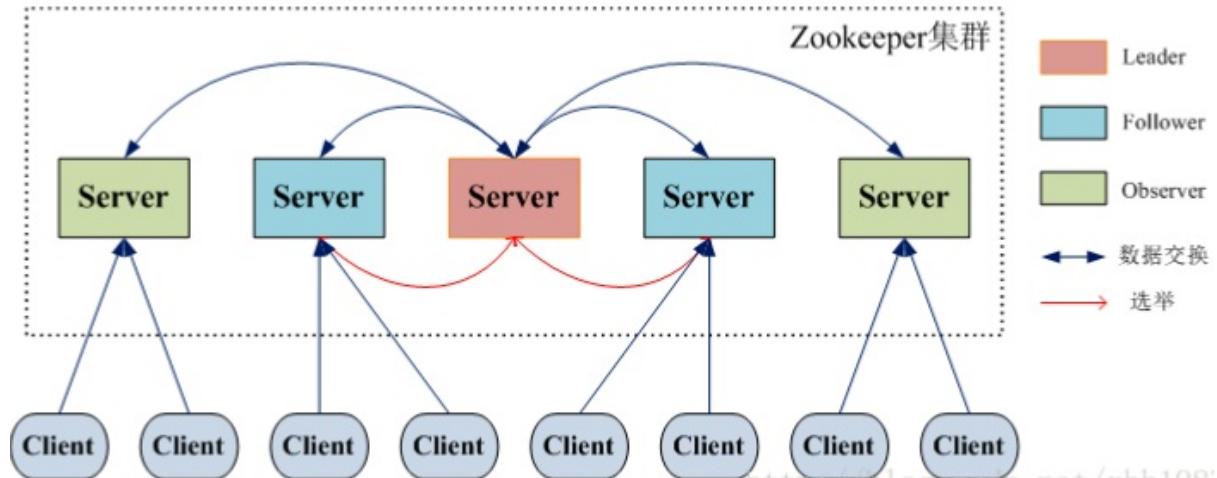
## 4.4 高性能

ZooKeeper 是高性能的。在“读”多于“写”的应用程序中尤其地高性能，因为“写”会导致所有的服务器间同步状态。（“读”多于“写”是协调服务的典型场景。）

# 五 ZooKeeper 集群角色介绍

最典型集群模式：**Master/Slave** 模式（主备模式）。在这种模式中，通常 Master 服务器作为主服务器提供写服务，其他的 Slave 服务器从服务器通过异步复制的方式获取 Master 服务器最新的数据提供读服务。

但是，在 ZooKeeper 中没有选择传统的 **Master/Slave** 概念，而是引入了 **Leader**、**Follower** 和 **Observer** 三种角色。如下图所示



ZooKeeper 集群中的所有机器通过一个 Leader 选举过程来选定一台称为“Leader”的机器，Leader 既可以为客户端提供写服务又能提供读服务。除了 Leader 外，Follower 和 Observer 都只能提供读服务。Follower 和 Observer 唯一的区别在于 Observer 机器不参与 Leader 的选举过程，也不参与写操作的“过半写成功”策略，因此 Observer 机器可以在不影响写性能的情况下提升集群的读性能。

| 角色            |                | 主要工作描述                                                                                                         |
|---------------|----------------|----------------------------------------------------------------------------------------------------------------|
| 学习者 (Learner) | 领导者            | 1. 事务请求的唯一调度和处理者，保证集群事务处理的顺序性；<br>2. 集群内部各服务器的调度者                                                              |
|               | 跟随者 (Follower) | 1. 处理客户端非事务请求，转发事务请求给 Leader 服务器<br>2. 参与事务请求 Proposal 的投票<br>3. 参与 Leader 选举的投票                               |
|               | 观察者 (Observer) | Follower 和 Observer 唯一的区别在于 Observer 机器不参与 Leader 的选举过程，也不参与写操作的“过半写成功”策略，因此 Observer 机器可以在不影响写性能的情况下提升集群的读性能。 |
| 客户端 (Client)  |                | 请求发起方                                                                                                          |

当 Leader 服务器出现网络中断、崩溃退出与重启等异常情况时，ZAB 协议就会进入恢复模式并选举产生新的 Leader 服务器。这个过程大致是这样的：

1. Leader election (选举阶段)：节点在一开始都处于选举阶段，只要有一个节点得到超半数节点的票数，它就可以当选准 leader。
2. Discovery (发现阶段)：在这个阶段，followers 跟准 leader 进行通信，同步 followers 最近接收的事务提议。
3. Synchronization (同步阶段)：同步阶段主要是利用 leader 前一阶段获得的最新提议历史，同步集群中所有的副本。同步完成之后准 leader 才会成为真正的 leader。
4. Broadcast (广播阶段) 到了这个阶段，Zookeeper 集群才能正式对外提供事务服务，并且 leader 可以进行消息广播。同时如果有新的节点加入，还需要对新节点进行同步。

## 六 ZooKeeper &ZAB 协议&Paxos算法

### 6.1 ZAB 协议&Paxos算法

Paxos 算法应该可以说是 ZooKeeper 的灵魂了。但是，ZooKeeper 并没有完全采用 Paxos 算法，而是使用 ZAB 协议作为其保证数据一致性的核心算法。另外，在 ZooKeeper 的官方文档中也指出，ZAB 协议并不像 Paxos 算法那样，是一种通用的分布式一致性算法，它是一种特别为 Zookeeper 设计的崩溃可恢复的原子消息广播算法。

### 6.2 ZAB 协议介绍

ZAB (ZooKeeper Atomic Broadcast 原子广播) 协议是为分布式协调服务 ZooKeeper 专门设计的一种支持崩溃恢复的原子广播协议。在 ZooKeeper 中，主要依赖 ZAB 协议来实现分布式数据一致性，基于该协议，ZooKeeper 实现了一种主备模式的系统架构来保持集群中各个副本之间的数据一致性。

### 6.3 ZAB 协议两种基本的模式：崩溃恢复和消息广播

ZAB 协议包括两种基本的模式，分别是 崩溃恢复和消息广播。当整个服务框架在启动过程中，或是当 Leader 服务器出现网络中断、崩溃退出与重启等异常情况时，ZAB 协议就会进入恢复模式并选举产生新的 Leader 服务器。当选举产生了新的 Leader 服务器，同时集群中已经有过半的机器与该 Leader 服务器完成了状态同步之后，ZAB 协议就会退出恢复模式。其中，所谓状态同步是指数据同步，用来保证集群中存在过半的机器能够和 Leader 服务器的数据状态保持一致。

当集群中已经有过半的 Follower 服务器完成了和 Leader 服务器的状态同步，那么整个服务框架就可以进入消息广播模式了。当一台同样遵守 ZAB 协议的服务器启动后加入到集群中时，如果此时集群中已经存在一个 Leader 服务器在负责进行消息广播，那么新加人的服务器就会自觉地进入数据恢复模式：找到 Leader 所在的服务器，并与其进行数据同步，然后一起参与到消息广播流程中去。正如上文介绍中所说的，ZooKeeper 设计成只允许唯一的一个 Leader 服务器来进行事务请求的处理。Leader 服务器在接收到客户端的事务请求后，会生成对应的事务提案并发起一轮广播协议；而如果集群中的其他机器接收到客户端的事务请求，那么这些非 Leader 服务器会首先将这个事务请求转发给 Leader 服务器。

关于 ZAB 协议&Paxos 算法 需要讲和理解的东西太多了，说实话，笔者到现在不太清楚这俩兄弟的具体原理和实现过程。推荐阅读下面两篇文章：

- 图解 Paxos 一致性协议
- Zookeeper ZAB 协议分析

关于如何使用 zookeeper 实现分布式锁，可以查看下面这篇文章：

- 10分钟看懂！基于Zookeeper的分布式锁

## 六 总结

通过阅读本文，想必大家已从 ①ZooKeeper 的由来。-> ②ZooKeeper 到底是什么。-> ③ZooKeeper 的一些重要概念（会话（Session）、Znode、版本、Watcher、ACL）-> ④ZooKeeper 的特点。-> ⑤ZooKeeper 的设计目标。-> ⑥ZooKeeper 集群角色介绍（Leader、Follower 和 Observer 三种角色）-> ⑦ZooKeeper &ZAB 协议&Paxos 算法。这七点了解了 ZooKeeper。

## 参考

- 《从 Paxos 到 Zookeeper》
- <https://cwiki.apache.org/confluence/display/ZOOKEEPER/ProjectDescription>
- <https://cwiki.apache.org/confluence/display/ZOOKEEPER/Index>
- <https://www.cnblogs.com/raphael5200/p/5285583.html>

- <https://zhuanlan.zhihu.com/p/30024403>

Java面试通关手册（Java学习指南，欢迎Star，会一直完善下去，欢迎建议和指导）：[https://github.com/Snailclimb/Java\\_Guide](https://github.com/Snailclimb/Java_Guide)

## 书籍推荐

《高性能MySQL：第3版》

## 文字教程推荐

MySQL 教程（菜鸟教程）

MySQL教程（易百教程）

## 视频教程推荐

基础入门：[与MySQL的零距离接触-慕课网](#)

Mysql开发技巧：[MySQL开发技巧（一）](#)    [MySQL开发技巧（二）](#)    [MySQL开发技巧（三）](#)

Mysql5.7新特性及相关优化技巧：[MySQL5.7版本新特性](#)    [性能优化之MySQL优化](#)

[MySQL集群（PXC）入门](#)    [MyCAT入门及应用](#)

## 常见问题总结

### • ①存储引擎

[MySQL常见的两种存储引擎：MyISAM与InnoDB的爱恨情仇](#)

### • ②字符集及校对规则

字符集指的是一种从二进制编码到某类字符符号的映射。校对规则则是指某种字符集下的排序规则。Mysql中每一种字符集都会对应一系列的校对规则。

Mysql采用的是类似继承的方式指定字符集的默认值，每个数据库以及每张数据表都有自己的默认值，他们逐层继承。比如：某个库中所有表的默认字符集将是该数据库所指定的字符集（这些表在没有指定字符集的情况下，才会采用默认字符集） PS：整理自《Java工程师修炼之道》

详细内容可以参考：[MySQL字符集及校对规则的理解](#)

### • ③索引相关的内容（数据库使用中非常关键的技术，合理正确的使用索引可以大大提高数据库的查询性能）

Mysql索引使用的数据结构主要有**BTree索引** 和 **哈希索引**。对于哈希索引来说，底层的数据结构就是哈希表，因此在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能最快；其余大部分场景，建议选择BTree索引。

Mysql的BTree索引使用的是B数中的B+Tree，但对于主要的两种存储引擎的实现方式是不同的。

**MyISAM：** B+Tree叶节点的data域存放的是数据记录的地址。在索引检索的时候，首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其 data 域的值，然后以 data 域的值为地址读取相应的数据记录。这被称为“非聚簇索引”。

**InnoDB:** 其数据文件本身就是索引文件。相比MyISAM，索引文件和数据文件是分离的，其表数据文件本身就是按B+Tree组织的一个索引结构，树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。这被称为“聚簇索引（或聚集索引）”。而其余的索引都作为辅助索引（非聚集索引），辅助索引的data域存储相应记录主键的值而不是地址，这也是和MyISAM不同的地方。在根据主索引搜索时，直接找到key所在的节点即可取出数据；在根据辅助索引查找时，则需要先取出主键的值，在走一遍主索引。因此，在设计表的时候，不建议使用过长的字段作为主键，也不建议使用非单调的字段作为主键，这样会造成主索引频繁分裂。PS：整理自《Java工程师修炼之道》

详细内容可以参考：

[干货：mysql索引的数据结构](#)

[MySQL优化系列（三）--索引的使用、原理和设计优化](#)

[数据库两大神器【索引和锁】](#)

## • ④查询缓存的使用

my.cnf加入以下配置，重启Mysql开启查询缓存

```
query_cache_type=1
query_cache_size=600000
```

Mysql执行以下命令也可以开启查询缓存

```
set global query_cache_type=1;
set global query_cache_size=600000;
```

如上，开启查询缓存后在同样的查询条件以及数据情况下，会直接在缓存中返回结果。这里的查询条件包括查询本身、当前要查询的数据库、客户端协议版本号等一些可能影响结果的信息。因此任何两个查询在任何字符上的不同都会导致缓存不命中。此外，如果查询中包含任何用户自定义函数、存储函数、用户变量、临时表、Mysql库中的系统表，其查询结果也不会被缓存。

缓存建立之后，Mysql的查询缓存系统会跟踪查询中涉及的每张表，如果这些表（数据或结构）发生变化，那么和这张表相关的所有缓存数据都将失效。

缓存虽然能够提升数据库的查询性能，但是缓存同时也带来了额外的开销，每次查询后都要做一次缓存操作，失效后还要销毁。因此，开启缓存查询要谨慎，尤其对于写密集的应用来说更是如此。如果开启，要注意合理控制缓存空间大小，一般来说其大小设置为几十MB比较合适。此外，还可以通过`sql_cache`和`sql_no_cache`来控制某个查询语句是否需要缓存：

```
select sql_no_cache count(*) from usr;
```

## • ⑤事务机制

关系性数据库需要遵循ACID规则，具体内容如下：



1. 原子性：事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
2. 一致性：执行事务前后，数据保持一致；
3. 隔离性：并发访问数据库时，一个用户的事物不被其他事物所干扰，各并发事务之间数据库是独立的；
4. 持久性：一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

为了达到上述事务特性，数据库定义了几种不同的事务隔离级别：

- **READ\_UNCOMMITTED**（未授权读取）：最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读
- **READ\_COMMITTED**（授权读取）：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
- **REPEATABLE\_READ**（可重复读）：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。
- **SERIALIZABLE**（串行）：最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

这里需要注意的是：**Mysql** 默认采用的 **REPEATABLE\_READ** 隔离级别 **Oracle** 默认采用的 **READ\_COMMITTED** 隔离级别。

事务隔离机制的实现基于锁机制和并发调度。其中并发调度使用的是MVVC（多版本并发控制），通过保存修改的旧版本信息来支持并发一致性读和回滚等特性。

详细内容可以参考：[可能是最漂亮的Spring事务管理详解](#)

## • ⑥锁机制与InnoDB锁算法

**MyISAM**和**InnoDB**存储引擎使用的锁：

- MyISAM采用表级锁(table-level locking)。
- InnoDB支持行级锁(row-level locking)和表级锁，默认为行级锁

表级锁和行级锁对比：

- **表级锁**：Mysql中锁定粒度最大的一种锁，对当前操作的整张表加锁，实现简单，资源消耗也比较少，加锁快，不会出现死锁。其锁定粒度最大，触发锁冲突的概率最高，并发度最低，MyISAM和 InnoDB引擎都支持表级锁。

- 行级锁：Mysql中锁定粒度最小的一种锁，只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，并发度高，但加锁的开销也最大，加锁慢，会出现死锁。

详细内容可以参考：[Mysql锁机制简单了解一下](#)

### InnoDB存储引擎的锁的算法有三种：

- Record lock：单个行记录上的锁
- Gap lock：间隙锁，锁定一个范围，不包括记录本身
- Next-key lock：record+gap 锁定一个范围，包含记录本身

相关知识点：

1. innodb对于行的查询使用next-key lock
2. Next-locking keying为了解决Phantom Problem幻读问题
3. 当查询的索引含有唯一属性时，将next-key lock降级为record key
4. Gap锁设计的目的是为了阻止多个事务将记录插入到同一范围内，而这会导致幻读问题的产生
5. 有两种方式显式关闭gap锁：（除了外键约束和唯一性检查外，其余情况仅使用record lock） A. 将事务隔离级别设置为RC B. 将参数innodb\_locks\_unsafe\_for\_binlog设置为1

## • ⑦大表优化

当MySQL单表记录数过大时，数据库的CRUD性能会明显下降，一些常见的优化措施如下：

1. 限定数据的范围：务必禁止不带任何限制数据范围条件的查询语句。比如：我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内。；
2. 读/写分离：经典的数据库拆分方案，主库负责写，从库负责读；
3. 缓存：使用MySQL的缓存，另外对重量级、更新少的数据可以考虑使用应用级别的缓存；
4. 垂直分区：

根据数据库里面数据表的相关性进行拆分。例如，用户表中既有用户的登录信息又有用户的基本信息，可以将用户表拆分成两个单独的表，甚至放到单独的库做分库。

简单来说垂直拆分是指数据表列的拆分，把一张列比较多的表拆分为多张表。如下图所示，这样来说大家应该就更容易理解了。



**垂直拆分的优点：**可以使得行数据变小，在查询时减少读取的Block数，减少I/O次数。此外，垂直分区可以简化表的结构，易于维护。

**垂直拆分的缺点：**主键会出现冗余，需要管理冗余列，并会引起Join操作，可以通过在应用层进行Join来解决。此外，垂直分区会让事务变得更加复杂；

### 5. 水平分区：

\*\*保持数据表结构不变，通过某种策略存储数据分片。这样每一片数据分散到不同的表或者库中，达到了分布式的目的。水平拆分可以支撑非常大的数据量。\*\*

水平拆分是指数据表行的拆分，表的行数超过200万行时，就会变慢，这时可以把一张的表的数据拆成多张表来存放。举个例子：我们可以将用户信息表拆分成多个用户信息表，这样就可以避免单一表数据量过大对性能造成影响。

! [数据库水平拆分] (<https://user-gold-cdn.xitu.io/2018/6/16/164084b7e9e423e3?w=690&h=271&f=jpeg&s=23119>)

水晶拆分可以支持非常大的数据量。需要注意的一点是：分表仅仅是解决了单一表数据过大的问题，但由于表的数据还是在同一台机器上，其实对于提升MySQL并发能力没有什么意义，所以 \*\*水晶拆分最好分库\*\*。

水平拆分能够 \*\*支持非常大的数据量存储，应用端改造也少\*\*，但 \*\*分片事务难以解决\*\*，跨界点Join性能较差，逻辑复杂。《Java工程师修炼之道》的作者推荐 \*\*尽量不要对数据进行分片，因为拆分会带来逻辑、部署、运维的各种复杂度\*\*，一般的数据表在优化得当的情况下支撑千万以下的数据量是没有太大问题的。如果实在要分片，尽量选择客户端分片架构，这样可以减少一次和中间件的网络I/O。

\*\*下面补充一下数据库分片的两种常见方案：\*\*

- \*\*客户端代理：\*\* \*\*分片逻辑在应用端，封装在jar包中，通过修改或者封装JDBC层来实现。\*\* 当当网的 \*\*Sharding-JDBC\*\*、阿里的TDDL是两种比较常用的实现。
- \*\*中间件代理：\*\* \*\*在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件服务中。\*\* 我们现在谈的 \*\*Mycat\*\*、360的Atlas、网易的DDB等等都是这种架构的实现。

详细内容可以参考：[MySQL大表优化方案](#)

Redis 是一个使用 C 语言写成的，开源的 key-value 数据库。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash (哈希类型)。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis支持各种不同方式的排序。与memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。目前，Vmware在资助着redis项目的开发和维护。

## 书籍推荐

《Redis实战》

《Redis设计与实现》

## 教程推荐

redis官方中文版教程：<http://www.redis.net.cn/tutorial/3501.html>

Redis 教程（菜鸟教程）：<http://www.runoob.com/redis/redis-tutorial.html>

## 常见问题总结

就我个人而言，我觉得Redis的基本使用是我们每个Java程序员都应该会的。另外，如果需要面试的话，一些关于Redis的理论知识也需要好好的学习一下。学完Redis之后，对照着下面8点看看自己还有那些不足的地方，同时，下面7点也是面试中经常会问到的。另外，《Redis实战》、《Redis设计与实现》是我比较推荐的两本学习Redis的书籍。

1. Redis的两种持久化操作以及如何保障数据安全（快照和AOF）
2. 如何防止数据出错（Redis事务）
3. 如何使用流水线来提升性能
4. Redis主从复制
5. Redis集群的搭建
6. Redis的几种淘汰策略
7. Redis集群宕机，数据迁移问题
8. Redis缓存使用有很多，怎么解决缓存雪崩和缓存穿透？

# Redis常见问题分析与好文Mark

## 什么是Redis?

Redis 是一个使用 C 语言写成的，开源的 key-value 数据库。。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash (哈希类型)。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis支持各种不同方式的排序。与memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。目前，Vmware在资助着redis项目的开发和维护。

## Redis与Memcached的区别与比较

1、Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，zset，hash等数据结构的存储。memcache支持简单的数据类型，String。

2、Redis支持数据的备份，即master-slave模式的数据备份。

3、Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用，而Memcached把数据全部存在内存之中

4、redis的速度比memcached快很多

5、Memcached是多线程，非阻塞IO复用的网络模型；Redis使用单线程的IO复用模型。

| 对比参数     | Redis                                                        | Memcached                            |
|----------|--------------------------------------------------------------|--------------------------------------|
| 类型       | 1、支持内存<br>2、非关系型数据库                                          | 1、支持内存<br>2、key-value键值对形式<br>3、缓存系统 |
| 数据存储类型   | 1、String<br>2、List<br>3、Set<br>4、Hash<br>5、Sort Set 【俗称ZSet】 | 1、文本型<br>2、二进制类型【新版增加】               |
| 查询【操作】类型 | 1、批量操作<br>2、事务支持【虽然是假的事务】<br>3、每个类型不同的CRUD                   | 1、CRUD<br>2、少量的其他命令                  |
| 附加功能     | 1、发布/订阅模式<br>2、主从分区<br>3、序列化支持<br>4、脚本支持【Lua脚本】              | 1、多线程服务支持                            |
| 网络IO模型   | 1、单进程模式                                                      | 2、多线程、非阻塞IO模式                        |
| 事件库      | 自封装简易事件库AeEvent                                              | 贵族血统的LibEvent事件库                     |
| 持久化支持    | 1、RDB<br>2、AOF                                               | 不支持                                  |

如果想要更详细了解的话，可以查看慕课网上的这篇手记（非常推荐）：《脚踏两只船的困惑 - Memcached与Redis》：<https://www.imooc.com/article/23549>

## Redis与Memcached的选择

终极策略：使用Redis的String类型做的事，都可以用Memcached替换，以此换取更好的性能提升；除此以外，优先考虑Redis：

### 使用redis有哪些好处？

(1) 速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)

(2) 支持丰富数据类型，支持string, list, set, sorted set, hash

(3) 支持事务：redis对事务是部分支持的，如果是在入队时报错，那么都不会执行；在非入队时报错，那么成功的就会成功执行。详细了解请参考：《Redis事务介绍（四）》：<https://blog.csdn.net/cuipeng0916/article/details/53698774>

redis监控：锁的介绍

(4) 丰富的特性：可用于缓存，消息，按key设置过期时间，过期后将会自动删除

## Redis常见数据结构使用场景

### 1. String

常用命令：set, get, decr, incr, mget 等。

String数据结构是简单的key-value类型，value其实不仅可以是String，也可以是数字。常规key-value缓存应用；常规计数：微博数，粉丝数等。

## 2.Hash

常用命令： hget,hset,hgetall 等。

Hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。比如我们可以Hash数据结构来存储用户信息，商品信息等等。

**举个例子：**最近做的一个电商网站项目的首页就使用了redis的hash数据结构进行缓存，因为一个网站的首页访问量是最大的，所以通常网站的首页可以通过redis缓存来提高性能和并发量。我用jedis客户端来连接和操作我搭建的redis集群或者单机redis，利用jedis可以很容易的对redis进行相关操作，总的来说从搭一个简单的集群到实现redis作为缓存的整个步骤不难。感兴趣的可以看我昨天写的这篇文章：

《一文轻松搞懂redis集群原理及搭建与使用》：<https://juejin.im/post/5ad54d76f265da23970759d3>

## 3.List

常用命令： lpush,rpush,lpop,rpop,range等

list就是链表，Redis list的应用场景非常多，也是Redis最重要的数据结构之一，比如微博的关注列表，粉丝列表，最新消息排行等功能都可以用Redis的list结构来实现。

Redis list的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销。

## 4.Set

常用命令： sadd,spop,smembers,sunion 等

set对外提供的功能与list类似是一个列表的功能，特殊之处在于set是可以自动排重的。当你需要存储一个列表数据，又不希望出现重复数据时，set是一个很好的选择，并且set提供了判断某个成员是否在一个set集合内的重要接口，这个也是list所不能提供的。

在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis可以非常方便的实现如共同关注、共同喜好、二度好友等功能。

## 5.Sorted Set

常用命令： zadd,zrange,zrem,zcard等

和set相比，sorted set增加了一个权重参数score，使得集合中的元素能够按score进行有序排列。

**举例：**在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维度的消息排行榜）等信息，适合使用Redis中的SortedSet结构进行存储。

**MySQL里有2000w数据，Redis中只存20w的数据，如何保证Redis中的数据都是热点数据（redis有哪些数据淘汰策略？？？）**

相关知识：redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略（回收策略）。redis 提供 6种数据淘汰策略：

1. **volatile-lru**: 从已设置过期时间的数据集 (server.db[i].expires) 中挑选最近最少使用的数据淘汰
2. **volatile-ttl**: 从已设置过期时间的数据集 (server.db[i].expires) 中挑选将要过期的数据淘汰
3. **volatile-random**: 从已设置过期时间的数据集 (server.db[i].expires) 中任意选择数据淘汰
4. **allkeys-lru**: 从数据集 (server.db[i].dict) 中挑选最近最少使用的数据淘汰

5. **allkeys-random**: 从数据集 (server.db[i].dict) 中任意选择数据淘汰
6. **no-eviction** (驱逐) : 禁止驱逐数据

## Redis的并发竞争问题如何解决?

Redis为单进程单线程模式，采用队列模式将并发访问变为串行访问。Redis本身没有锁的概念，Redis对于多个客户端连接并不存在竞争，但是在Jedis客户端对Redis进行并发访问时会发生连接超时、数据转换错误、阻塞、客户端关闭连接等问题，这些问题均是由于客户端连接混乱造成。对此有2种解决方法：

1.客户端角度，为保证每个客户端间正常有序与Redis进行通信，对连接进行池化，同时对客户端读写Redis操作采用内部锁synchronized。 2.服务器角度，利用setnx实现锁。

注：对于第一种，需要应用程序自己处理资源的同步，可以使用的方法比较通俗，可以使用synchronized也可以使用lock；第二种需要用到Redis的setnx命令，但是需要注意一些问题。

## Redis回收进程如何工作的? Redis回收使用的是什么算法?

Redis内存回收:LRU算法（写的很不错，推荐）：<https://www.cnblogs.com/WJ5888/p/4371647.html>

## Redis 大量数据插入

官方文档给的解释：<http://www.redis.cn/topics/mass-insert.html>

## Redis 分区的优势、不足以及分区类型

官方文档提供的讲解：<http://www.redis.net.cn/tutorial/3524.html>

## Redis持久化数据和缓存怎么做扩容?

《redis的持久化和缓存机制》：<https://github.com/Snailclimb/Java-Guide/blob/master/数据存储/R春夏秋冬又一春之Redis持久化.md>

扩容的话可以通过redis集群实现，之前做项目的时候用过自己搭的redis集群 然后写了一篇关于redis集群的文章：《一文轻松搞懂redis集群原理及搭建与使用》：<https://juejin.im/post/5ad54d76f265da23970759d3>

## Redis常见性能问题和解决方案:

1. Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件
2. 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次
3. 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内
4. 尽量避免在压力很大的主库上增加从库

## Redis与消息队列

作者：翁伟 链接：<https://www.zhihu.com/question/20795043/answer/345073457>

不要使用redis去做消息队列，这不是redis的设计目标。但实在太多人使用redis去做去消息队列，redis的作者看不下去，另外基于redis的核心代码，另外实现了一个消息队列disque： antirez/disque:<https://github.com/antirez/disque>部署、协议等方面都跟redis非常类似，并且支持集群，延迟消息等等。

我在做网站过程接触比较多的还是使用redis做缓存，比如秒杀系统，首页缓存等等。

## 好文Mark

非常非常推荐下面几篇文章。。。

《Redis深入之道：原理解析、场景使用以及视频解读》：<https://zhuanlan.zhihu.com/p/28073983>: 主要介绍了：Redis集群开源的方案、Redis协议简介及持久化Aof文件解析、Redis短连接性能优化等等内容，文章干货太大，容量很大，建议时间充裕可以看看。另外文章里面还提供了视频讲解，可以说是非常非常用心了。

《阿里云Redis混合存储典型场景：如何轻松搭建视频直播间系统》：[https://yq.aliyun.com/articles/582487?utm\\_content=m\\_46529](https://yq.aliyun.com/articles/582487?utm_content=m_46529): 主要介绍视频直播间系统，以及如何使用阿里云Redis混合存储实例方便快捷的构建大数据量，低延迟的视频直播间服务。还介绍到了我们之前提高过的redis的数据结构的使用场景

《美团在Redis上踩过的一些坑-5.redis cluster遇到的一些问》：<http://carlosfu.iteye.com/blog/2254573>: 主要介绍了redis集群的两个常见问题，然后分享了一些关于redis集群不错的文章。

参考：

<https://www.cnblogs.com/Survivalist/p/8119891.html>

<http://www.redis.net.cn/tutorial/3524.html>

<https://redis.io/>

这篇文章主要是对 Redis 官方网站刊登的 [Distributed locks with Redis](#) 部分内容的总结和翻译。

## 什么是 RedLock

Redis 官方站这篇文章提出了一种权威的基于 Redis 实现分布式锁的方式名叫 *Redlock*, 此种方式比原先的单节点的方法更安全。它可以保证以下特性：

1. 安全特性：互斥访问，即永远只有一个 client 能拿到锁
2. 避免死锁：最终 client 都可能拿到锁，不会出现死锁的情况，即使原本锁住某资源的 client crash 了或者出现了网络分区
3. 容错性：只要大部分 Redis 节点存活就可以正常提供服务

## 怎么在单节点上实现分布式锁

```
SET resource_name my_random_value NX PX 30000
```

主要依靠上述命令，该命令仅当 Key 不存在时（NX保证）set 值，并且设置过期时间 3000ms（PX保证），值 my\_random\_value 必须是所有 client 和所有锁请求发生期间唯一的，释放锁的逻辑是：

```
if redis.call("get",KEYS[1]) == ARGV[1] then
 return redis.call("del",KEYS[1])
else
 return 0
end
```

上述实现可以避免释放另一个client创建的锁，如果只有 del 命令的话，那么如果 client1 拿到 lock1 之后因为某些操作阻塞了很长时间，此时 Redis 端 lock1 已经过期了并且已经被重新分配给了 client2，那么 client1 此时再去释放这把锁就会造成 client2 原本获取到的锁被 client1 无故释放了，但现在为每个 client 分配一个 unique 的 string 值可以避免这个问题。至于如何去生成这个 unique string，方法很多随意选择一种就行了。

## Redlock 算法

算法很易懂，起 5 个 master 节点，分布在不同的机房尽量保证可用性。为了获得锁，client 会进行如下操作：

1. 得到当前的时间，微妙单位
2. 尝试顺序地在 5 个实例上申请锁，当然需要使用相同的 key 和 random value，这里一个 client 需要合理设置与 master 节点沟通的 timeout 大小，避免长时间和一个 fail 了的节点浪费时间
3. 当 client 在大于等于 3 个 master 上成功申请到锁的时候，且它会计算申请锁消耗了多少时间，这部分消耗的时间采用获得锁的当下时间减去第一步获得的时间戳得到，如果锁的持续时长（lock validity time）比流逝的时间多的话，那么锁就真正获取到了。
4. 如果锁申请到了，那么锁真正的 lock validity time 应该是 origin (lock validity time) - 申请锁期间流逝的时间
5. 如果 client 申请锁失败了，那么它就会在少部分申请成功锁的 master 节点上执行释放锁的操作，重置状态

## 失败重试

如果一个 client 申请锁失败了，那么它需要稍等一会在重试避免多个 client 同时申请锁的情况，最好的情况是一个 client 需要几乎同时向 5 个 master 发起锁申请。另外就是如果 client 申请锁失败了它需要尽快在它曾经申请到锁的 master 上执行 unlock 操作，便于其他 client 获得这把锁，避免这些锁过期造成的时间浪费，当然如果这时候网络分区使得 client 无法联系上这些 master，那么这种浪费就是不得不付出的代价了。

## 放锁

放锁操作很简单，就是依次释放所有节点上的锁就行了

## 性能、崩溃恢复和 fsync

如果我们的节点没有持久化机制，client 从 5 个 master 中的 3 个处获得了锁，然后其中一个重启了，这是注意 整个环境中又出现了 3 个 master 可供另一个 client 申请同一把锁！违反了互斥性。如果我们开启了 AOF 持久化那么情况会稍微好转一些，因为 Redis 的过期机制是语义层面实现的，所以在 server 挂了的时候时间依旧在流逝，重启之后锁状态不会受到污染。但是考虑断电之后呢，AOF部分命令没来得及刷回磁盘直接丢失了，除非我们配置刷回策略为 `fsync = always`，但这会损伤性能。解决这个问题的方法是，当一个节点重启之后，我们规定在 max TTL 期间它是不可用的，这样它就不会干扰原本已经申请到的锁，等到它 crash 前的那部分锁都过期了，环境不存在历史锁了，那么再把这个节点加进来正常工作。

本文是对 Martin Kleppmann 的文章 [How to do distributed locking](#) 部分内容的翻译和总结，上次写 Redlock 的原因就是看到了 Martin 的这篇文章，写得很好，特此翻译和总结。感兴趣的同学可以翻看原文，相信会收获良多。

开篇作者认为现在 Redis 逐渐被使用到数据管理领域，这个领域需要更强的数据一致性和耐久性，这使得他感到担心，因为这不是 Redis 最初设计的初衷（事实上这也是很多业界程序员的误区，越来越把 Redis 当成数据库在使用），其中基于 Redis 的分布式锁就是令人担心的其一。

Martin 指出首先你要明确你为什么使用分布式锁，为了性能还是正确性？为了帮你区分这二者，在这把锁 fail 了的时候你可以询问自己以下问题：

1. **要性能的：** 拥有这把锁使得你不会重复劳动（例如一个 job 做了两次），如果这把锁 fail 了，两个节点同时做了这个 Job，那么这个 Job 增加了你的成本。
2. **要正确性的：** 拥有锁可以防止并发操作污染你的系统或者数据，如果这把锁 fail 了两个节点同时操作了一份数据，结果可能是数据不一致、数据丢失、file 冲突等，会导致严重的后果。

上述二者都是需求锁的正确场景，但是你必须清楚自己是因为什么原因需要分布式锁。

如果你只是为了性能，那没必要用 Redlock，它成本高且复杂，你只用一个 Redis 实例就够了，最多加个从防止主挂了。当然，你使用单节点的 Redis 那么断电或者一些情况下，你会丢失锁，但是你的目的只是加速性能且断电这种事情不会经常发生，这并不是什么大问题。并且如果你使用了单节点 Redis，那么很显然你这个应用需要的锁粒度是很模糊粗糙的，也不会是什么重要的服务。

那么是否 Redlock 对于要求正确性的场景就合适呢？Martin 列举了若干场景证明 Redlock 这种算法是不可靠的。

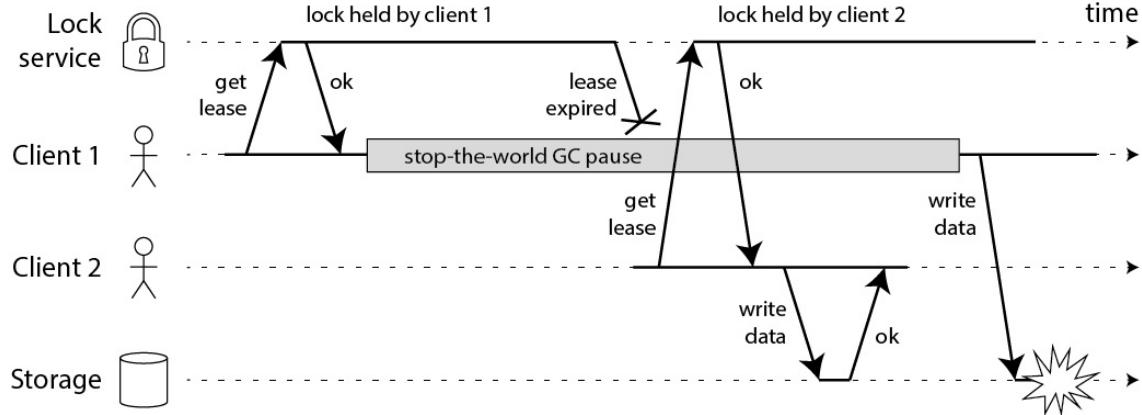
## 用锁保护资源

这节里 Martin 先将 Redlock 放在了一边而是仅讨论总体上一个分布式锁是怎么工作的。在分布式环境下，锁比 mutex 这类复杂，因为涉及到不同节点、网络通信并且他们随时可能无征兆的 fail。Martin 假设了一个场景，一个 client 要修改一个文件，它先申请得到锁，然后修改文件写回，放锁。另一个 client 再申请锁 ... 代码流程如下：

```
// THIS CODE IS BROKEN
function writeData(filename, data) {
 var lock = lockService.acquireLock(filename);
 if (!lock) {
 throw 'Failed to acquire lock';
 }

 try {
 var file = storage.readFile(filename);
 var updated = updateContents(file, data);
 storage.writeFile(filename, updated);
 } finally {
 lock.release();
 }
}
```

可惜即使你的锁服务非常完美，上述代码还是可能跪，下面的流程图会告诉你为什么：

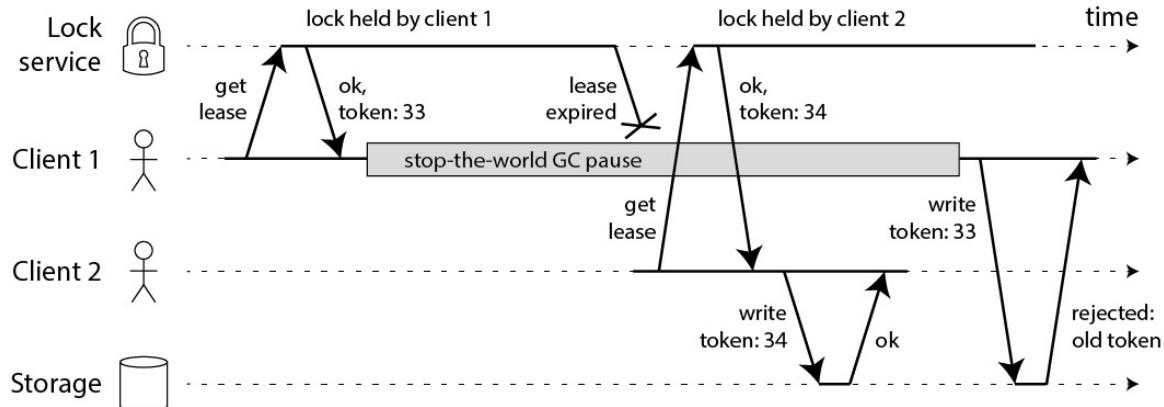


上述图中，得到锁的 client1 在持有锁的期间 pause 了一段时间，例如 GC 停顿。锁有过期时间（一般叫租约，为了防止某个 client 崩溃之后一直占有锁），但是如果 GC 停顿太长超过了锁租约时间，此时锁已经被另一个 client2 所得到，原先的 client1 还没有感知到锁过期，那么奇怪的结果就会发生，曾经 HBase 就发生过这种 Bug。即使你在 client1 写回之前检查一下锁是否过期也无助于解决这个问题，因为 GC 可能在任何时候发生，即使是你非常不便的时候（在最后的检查与写操作期间）。如果你认为自己的程序不会有长时间的 GC 停顿，还有其他原因会导致你的进程 pause。例如进程可能读取尚未进入内存的数据，所以它得到一个 page fault 并且等待 page 被加载进缓存；还有可能你依赖于网络服务；或者其他进程占用 CPU；或者其他意外发生 SIGSTOP 等。

.... 这里 Martin 又增加了一节列举各种进程 pause 的例子，为了证明上面的代码是不安全的，无论你的锁服务多完美。

## 使用 Fencing (栅栏) 使得锁变安全

修复问题的方法也很简单：你需要在每次写操作时加入一个 fencing token。这个场景下，fencing token 可以是一个递增的数字（lock service 可以做到），每次有 client 申请锁就递增一次：



client1 申请锁同时拿到 token33，然后它进入长时间的停顿锁也过期了。client2 得到锁和 token34 写入数据，紧接着 client1 活过来之后尝试写入数据，自身 token33 比 34 小因此写入操作被拒绝。注意这需要存储层来检查 token，但这并不难实现。如果你使用 Zookeeper 作为 lock service 的话那么你可以使用 zxid 作为递增数字。但是对于 Redlock 你要知道，没什么生成 fencing token 的方式，并且怎么修改 Redlock 算法使其能产生 fencing token 呢？好像并不那么显而易见。因为产生 token 需要单调递增，除非在单节点 Redis 上完成但是这又没有高可靠性，你好像需要引进一致性协议来让 Redlock 产生可靠的 fencing token。

## 使用时间来解决一致性

Redlock 无法产生 fencing token 早该成为在需求正确性的场景下弃用它的理由，但还有一些值得讨论的地方。

学术界有个说法，算法对时间不做假设：因为进程可能 pause 一段时间、数据包可能因为网络延迟延后到达、时钟可能根本就是错的。而可靠的算法依旧要在上述假设下做正确的事情。

对于 failure detector 来说，timeout 只能作为猜测某个节点 fail 的依据，因为网络延迟、本地时钟不正确等其他原因的限制。考虑到 Redis 使用 gettimeofday，而不是单调的时钟，会受到系统时间的影响，可能会突然前进或者后退一段时间，这会导致一个 key 更快或更慢地过期。

可见，Redlock 依赖于许多时间假设，它假设所有 Redis 节点都能对同一个 Key 在其过期前持有差不多的时间、跟过期时间相比网络延迟很小、跟过期时间相比进程 pause 很短。

## 用不可靠的时间打破 Redlock

这节 Martin 举了个因为时间问题，Redlock 不可靠的例子。

1. client1 从 ABC 三个节点处申请到锁，DE 由于网络原因请求没有到达
2. C 节点的时钟往前推了，导致 lock 过期
3. client2 在 CDE 处获得了锁，AB 由于网络原因请求未到达
4. 此时 client1 和 client2 都获得了锁

在 Redlock 官方文档中也提到了这个情况，不过是 C 崩溃的时候，Redlock 官方本身也是知道 Redlock 算法不是完全可靠的，官方为了解决这种问题建议使用延时启动，相关内容可以看之前的[这篇文章](#)。但是 Martin 这里分析得更加全面，指出延时启动不也是依赖于时钟的正确性的么？

接下来 Martin 又列举了进程 Pause 时而不是时钟不可靠时会发生的问题：

1. client1 从 ABCDE 处获得了锁
2. 当获得锁的 response 还没到达 client1 时 client1 进入 GC 停顿
3. 停顿期间锁已经过期了
4. client2 在 ABCDE 处获得了锁
5. client1 GC 完成收到了获得锁的 response，此时两个 client 又拿到了同一把锁

同时长时间的网络延迟也有可能导致同样的问题。

## Redlock 的同步性假设

这些例子说明了，仅有在你假设了一个同步性系统模型的基础上，Redlock 才能正常工作，也就是系统能满足以下属性：

1. 网络延时边界，即假设数据包一定能在某个最大延时之内到达
2. 进程停顿边界，即进程停顿一定在某个最大时间之内
3. 时钟错误边界，即不会从一个坏的 NTP 服务器处取得时间

## 结论

Martin 认为 Redlock 实在不是一个好的选择，对于需求性能的分布式锁应用它太重了且成本高；对于需求正确性的应用来说它不够安全。因为它对高危的时钟或者说其他上述列举的情况进行了不可靠的假设，如果你的应用只需要高性能的分布式锁不要求多高的正确性，那么单节点 Redis 够了；如果你的应用想要保住正确性，那么不建议 Redlock，建议使用一个合适的一致性协调系统，例如 Zookeeper，且保证存在 fencing token。



非常感谢《redis实战》真本书，本文大多内容也参考了书中的内容。非常推荐大家看一下《redis实战》这本书，感觉书中的很多理论性东西还是很不错的。

为什么本文的名字要加上春夏秋冬又一春，哈哈，这是一部韩国的电影，我感觉电影不错，所以就用在文章名字上了，没有什么特别的含义，然后下面的有些配图也是电影相关镜头。



很多时候我们需要持久化数据也就是将内存中的数据写入到硬盘里面，大部分原因是为了之后重用数据（比如重启机器、机器故障之后回复数据），或者是为了防止系统故障而将数据备份到一个远程位置。

Redis不同于Memcached的很重一点就是，**Redis支持持久化**，而且支持两种不同的持久化操作。Redis的一种持久化方式叫快照（snapshotting, RDB）,另一种方式是只追加文件（append-only file,AOF）。这两种方法各有千秋，下面我会详细这两种持久化方法是什么，怎么用，如何选择适合自己的持久化方法。

## 快照（snapshotting）持久化

Redis可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。Redis创建快照之后，可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis主从结构，主要用来提高Redis性能），还可以将快照留在原地以便重启服务器的时候使用。



快照持久化是Redis默认采用的持久化方式，在redis.conf配置文件中默认有此下配置：

```
save 900 1 #在900秒(15分钟)之后, 如果至少有1个key发生变化, Redis就会自动触发BGSAVE命令创建快照。
save 300 10 #在300秒(5分钟)之后, 如果至少有10个key发生变化, Redis就会自动触发BGSAVE命令创建快照。
save 60 10000 #在60秒(1分钟)之后, 如果至少有10000个key发生变化, Redis就会自动触发BGSAVE命令创建快照。
```

根据配置，快照将被写入dbfilename选项指定的文件里面，并存储在dir选项指定的路径上面。如果在新的快照文件创建完毕之前，Redis、系统或者硬件这三者中的任意一个崩溃了，那么Redis将丢失最近一次创建快照写入的所有数据。

举个例子：假设Redis的上一个快照是2: 35开始创建的，并且已经创建成功。下午3: 06时，Redis又开始创建新的快照，并且在下午3: 08快照创建完毕之前，有35个键进行了更新。如果在下午3: 06到3: 08期间，系统发生了崩溃，导致Redis无法完成新快照的创建工作，那么Redis将丢失下午2: 35之后写入的所有数据。另一方面，如果系统恰好在新的快照文件创建完毕之后崩溃，那么Redis将丢失35个键的更新数据。

创建快照的办法有如下几种：

- **BGSAVE命令：**客户端向Redis发送 **BGSAVE**命令 来创建一个快照。对于支持BGSAVE命令的平台来说（基本上所有平台支持，除了Windows平台），Redis会调用fork来创建一个子进程，然后子进程负责将快照写入硬盘，而父进程则继续处理命令请求。
- **SAVE命令：**客户端还可以向Redis发送 **SAVE**命令 来创建一个快照，接到SAVE命令的Redis服务器在快照创建完毕之前不会再响应任何其他命令。SAVE命令不常用，我们通常只会在没有足够内存去执行BGSAVE命令的情况下，又或者即使等待持久化操作执行完毕也无所谓的情况下，才会使用这个命令。
- **save选项：**如果用户设置了save选项（一般会默认设置），比如 **save 60 10000**，那么从Redis最近一次创建快照之后开始算起，当“60秒之内有10000次写入”这个条件被满足时，Redis就会自动触发BGSAVE命令。
- **SHUTDOWN命令：**当Redis通过SHUTDOWN命令接收到关闭服务器的请求时，或者接收到标准TERM信号时，会执行一个SAVE命令，阻塞所有客户端，不再执行客户端发送的任何命令，并在SAVE命令执行完毕之后关闭服务器。
- **一个Redis服务器连接到另一个Redis服务器：**当一个Redis服务器连接到另一个Redis服务器，并向对方发送SYNC命令来开始一次复制操作的时候，如果主服务器目前没有执行BGSAVE操作，或者主服务器并非刚刚执行完BGSAVE操作，那么主服务器就会执行BGSAVE命令

如果系统真的发生崩溃，用户将丢失最近一次生成快照之后更改的所有数据。因此，快照持久化只适用于即使丢失一部分数据也不会造成一些大问题的应用程序。不能接受这个缺点的话，可以考虑AOF持久化。

## AOF (append-only file) 持久化

与快照持久化相比，AOF持久化 的实时性更好，因此已成为主流的持久化方案。默认情况下Redis没有开启AOF (append only file) 方式的持久化，可以通过appendonly参数开启：

```
appendonly yes
```

开启AOF持久化后每执行一条会更改Redis中的数据的命令，Redis就会将该命令写入硬盘中的AOF文件。AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的，默认的文件名是appendonly.aof。



在Redis的配置文件中存在三种同步方式，它们分别是：

```
appendfsync always #每次有数据修改发生时都会写入AOF文件,这样会严重降低Redis的速度
appendfsync everysec #每秒钟同步一次,显示地将多个写命令同步到硬盘
appendfsync no #让操作系统决定何时进行同步
```

**appendfsync always** 可以实现将数据丢失减到最少，不过这种方式需要对硬盘进行大量的写入而且每次只写入一个命令，十分影响Redis的速度。另外使用固态硬盘的用户谨慎使用appendfsync always选项，因为这会明显降低固态硬盘的使用寿命。

为了兼顾数据和写入性能，用户可以考虑 **appendfsync everysec**选项，让Redis每秒同步一次AOF文件，Redis性能几乎没受到任何影响。而且这样即使出现系统崩溃，用户最多只会丢失一秒之内产生的数据。当硬盘忙于执行写入操作的时候，Redis还会优雅的放慢自己的速度以便适应硬盘的最大写入速度。

**appendfsync no** 选项一般不推荐，这种方案会使Redis丢失不定量的数据而且如果用户的硬盘处理写入操作的速度不够的话，那么当缓冲区被等待写入的数据填满时，Redis的写入操作将被阻塞，这会导致Redis的请求速度变慢。

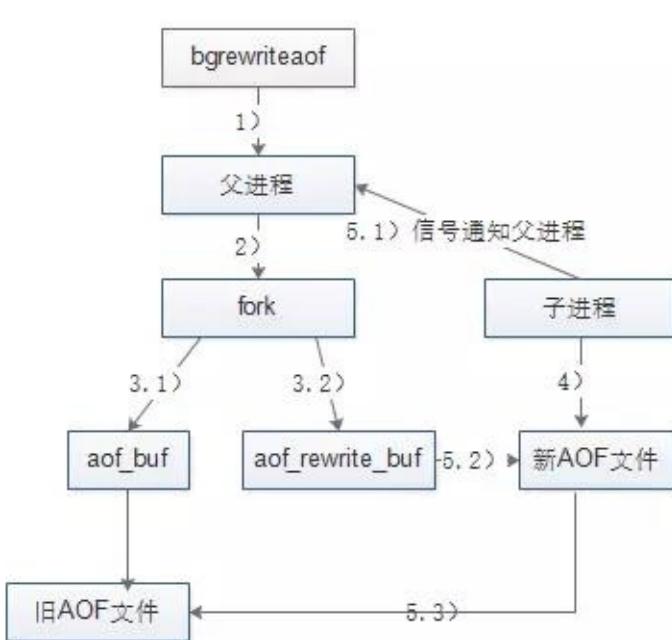
虽然AOF持久化非常灵活地提供了多种不同的选项来满足不同应用程序对数据安全的不同要求，但AOF持久化也有缺陷——AOF文件的体积太大。

## 重写/压缩AOF

AOF虽然在某个角度可以将数据丢失降低到最小而且对性能影响也很小，但是极端的情况下，体积不断增大的AOF文件很可能用完硬盘空间。另外，如果AOF体积过大，那么还原操作执行时间就可能会非常长。

为了解决AOF体积过大的问题，用户可以向Redis发送 **BGREWRITEAOF**命令，这个命令会通过移除AOF文件中的冗余命令来重写 (rewrite) AOF文件来减小AOF文件的体积。BGREWRITEAOF命令和BGSAVE创建快照原理十分相似，所以AOF文件重写也需要用到子进程，这样会导致性能问题和内存占用问题，和快照持久化一样。更糟糕的是，如果不加以控制的话，AOF文件的体积可能会比快照文件大好几倍。

### 文件重写流程：



| 和快照持久化可以通过设置 `save` 选项来自动执行

行BGSAVE一样， AOF持久化也可以通过设置

```
auto-aof-rewrite-percentage
```

选项和

```
auto-aof-rewrite-min-size
```

选项自动执行BGREWRITEAOF命令。举例：假设用户对Redis设置了如下配置选项并且启用了AOF持久化。那么当AOF文件体积大于64mb，并且AOF的体积比上一次重写之后的体积大了至少一倍（100%）的时候，Redis将执行BGREWRITEAOF命令。

```
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

无论是AOF持久化还是快照持久化，将数据持久化到硬盘上都是非常有必要的，但除了进行持久化外，用户还必须对持久化得到的文件进行备份（最好是备份到不同的地方），这样才能尽量避免数据丢失事故发生。如果条件允许的话，最好能将快照文件和重新重写的AOF文件备份到不同的服务器上面。

随着负载量的上升，或者数据的完整性变得越来越重要时，用户可能需要使用到复制特性。

参考：

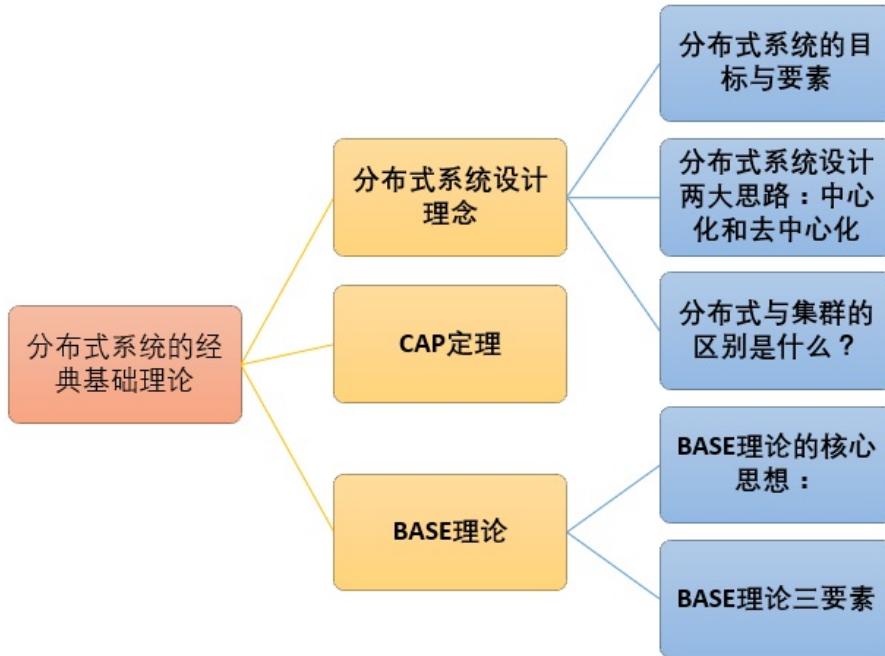
《Redis实战》

[深入学习Redis（2）：持久化](#)

## • 一 分布式系统的经典基础理论

### 分布式系统的经典基础理论

本文主要是简单的介绍了三个常见的概念： 分布式系统设计理念、 CAP定理、 BASE理论， 关于分布式系统的还有很多很多东西。



## • 二 分布式事务

分布式事务就是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。以上是百度百科的解释，简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

- 深入理解分布式事务
- 分布式事务？No, 最终一致性
- 聊聊分布式事务，再说说解决方案

## • 三 分布式系统一致性

### 分布式服务化系统一致性的“最佳实干”

## ◦ 四 一致性协议/算法

早在1898年就诞生了著名的 Paxos经典算法（Zookeeper就采用了Paxos算法的近亲兄弟Zab算法），但由于Paxos算法非常难以理解、实现、排错。所以不断有人尝试简化这一算法，直到2013年才有了重大突破：斯坦福的Diego Ongaro、John Ousterhout以易懂性为目标设计了新的一致性算法—— Raft算法，并发布了对应的论文《In Search of an Understandable Consensus Algorithm》，到现在有十多种语言实现的Raft算法框架，较为出名的有以Go语言实现的Etcd，它的功能类似于Zookeeper，但采用了更为主流的Rest接口。

- 图解 Paxos 一致性协议
- 图解分布式协议-RAFT
- Zookeeper ZAB 协议分析

## • 五 分布式存储

分布式存储系统将数据分散存储在多台独立的设备上。传统的网络存储系统采用集中的存储服务器存放所有数据，存储服务器成为系统性能的瓶颈，也是可靠性和安全性的焦点，不能满足大规模存储应用的需要。分布式网络存储系统采用可扩展的系统结构，利用多台存储服务器分担存储负荷，利用位置服务器定位存储信息，它不但提高了系统的可靠性、可用性和存取效率，还易于扩展。

- [分布式存储系统概要](#)

## • 六 分布式计算

所谓分布式计算是一门计算机科学，它研究如何把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分，然后把这些部分分配给许多计算机进行处理，最后把这些计算结果综合起来得到最终的结果。分布式网络存储技术是将数据分散的存储于多台独立的机器设备上。分布式网络存储系统采用可扩展的系统结构，利用多台存储服务器分担存储负荷，利用位置服务器定位存储信息，不但解决了传统集中式存储系统中单存储服务器的瓶颈问题，还提高了系统的可靠性、可用性和扩展性。

- [关于分布式计算的一些概念](#)

## 何谓悲观锁与乐观锁

乐观锁对应于生活中乐观的人总是想着事情往好的方向发展，悲观锁对应于生活中悲观的人总是想着事情往坏的方向发展。这两种人各有优缺点，不能不以场景而定说一种人好于另外一种人。

### 悲观锁

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中 `synchronized` 和 `ReentrantLock` 等独占锁就是悲观锁思想的实现。

### 乐观锁

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 `write_condition` 机制，其实都是提供的乐观锁。在 Java 中 `java.util.concurrent.atomic` 包下面的原子变量类就是使用了乐观锁的一种实现方式 **CAS** 实现的。

## 两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行 `retry`，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

## 乐观锁常见的两种实现方式

乐观锁一般会使用版本号机制或 **CAS** 算法实现。

### 1. 版本号机制

一般是在数据表中加上一个数据版本号 `version` 字段，表示数据被修改的次数，当数据被修改时，`version` 值会加一。当线程A要更新数据值时，在读取数据的同时也会读取 `version` 值，在提交更新时，若刚才读取到的 `version` 值为当前数据库中的 `version` 值相等时才更新，否则重试更新操作，直到更新成功。

举一个简单的例子：假设数据库中帐户信息表中有一个 `version` 字段，当前值为 1；而当前帐户余额字段（`balance`）为 \$100。

1. 操作员 A 此时将其读出（`version=1`），并从其帐户余额中扣除 \$50（\$100-\$50）。
2. 在操作员 A 操作的过程中，操作员 B 也读入此用户信息（`version=1`），并从其帐户余额中扣除 \$20（\$100-\$20）。
3. 操作员 A 完成了修改工作，将数据版本号加一（`version=2`），连同帐户扣除后余额（`balance=$50`），提交至数据库更新，此时由于提交数据版本大于数据库记录当前版本，数据被更新，数据库记录 `version` 更新为 2。
4. 操作员 B 完成了操作，也将版本号加一（`version=2`）试图向数据库提交数据（`balance=$80`），但此时比对数据库记录版本时发现，操作员 B 提交的数据版本号为 2，数据库记录当前版本也为 2，不满足“提交版本必须大于记录当前版本才能执行更新”的乐观锁策略，因此，操作员 B 的提交被驳回。

这样，就避免了操作员 B 用基于 `version=1` 的旧数据修改的结果覆盖操作员 A 的操作结果的可能。

### 2. CAS算法

即**compare and swap**（比较与交换），是一种有名的无锁算法。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。**CAS算法**涉及到三个操作数

- 需要读写的内存值 V
- 进行比较的值 A
- 拟写入的新值 B

当且仅当 V 的值等于 A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。

关于自旋锁，大家可以看一下这篇文章，非常不错：[《面试必备之深入理解自旋锁》](#)

## 乐观锁的缺点

ABA 问题是乐观锁一个常见的问题

### 1 ABA 问题

如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 "**ABA**"问题。

JDK 1.5 以后的 `AtomicStampedReference` 类 就提供了此种能力，其中的 `compareAndSet` 方法 就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

### 2 循环时间长开销大

**自旋CAS**（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline），使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

### 3 只能保证一个共享变量的原子操作

CAS只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5开始，提供了 `AtomicReference` 类 来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作.所以我们可以使用锁或者利用 `AtomicReference`类 把多个共享变量合并成一个共享变量来操作。

### CAS与synchronized的使用情景

简单的来说**CAS**适用于写比较少的情况下（多读场景，冲突一般较少），**synchronized**适用于写比较多的情况下（多写场景，冲突一般较多）

1. 对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。
2. 对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

补充：Java并发编程这个领域中synchronized关键字一直都是元老级的角色，很久之前很多人都会称它为“重量级锁”。但是，在JavaSE 1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的**偏向锁** 和 **轻量级锁** 以及其它各种优化之后变得在某些情况下并不是那么重了。synchronized的底层实现主要依靠**Lock-Free** 的队列，基本思路是自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。在线程冲突较少的情况下，可以获得和CAS类似的性能；而线程冲突严重的情况下，性能远高于CAS。

# 一 Java中的值传递和引用传递 (非常重要)

首先要明确的是：“对象传递（数组、类、接口）是引用传递，原始类型数据（整型、浮点型、字符型、布尔型）传递是值传递。”

## 那么什么是值传递和应用传递呢？

值传递是指对象被值传递，意味着传递了对象的一个副本，即使副本被改变，也不会影响源对象。（因为值传递的时候，实际上是将实参的值复制一份给形参。）

引用传递是指对象被引用传递，意味着传递的并不是实际的对象，而是对象的引用。因此，外部对引用对象的改变会反映到所有的对象上。（因为引用传递的时候，实际上是将实参的地址值复制一份给形参。）

有时候面试官不是单纯问你“Java中是值传递还是引用传递”是什么啊，骚年？而是给出一个例子，然后让你写出答案，这种也常见在笔试题目中！所以，非常重要的，请看下面的例子：

## 值传递和应用传递实例

### 1. 值传递

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

 swap(num1, num2);

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

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

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

结果：

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

解析：

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

### 2. 引用传递

```
public static void main(String[] args) {
 int[] arr = {1,2,3,4,5};
```

```

 change(arr);

 System.out.println(arr[0]);
}

public static void change(int[] array) {
//将数组的第一个元素变为0
 array[0] = 0;
}

```

**结果：**

```

1
0

```

**解析：**

无论是主函数，还是change方法，操作的都是同一个地址值对应的数组。。因此，外部对引用对象的改变会反映到所有的对象上。

## 一些特殊的例子

### 1. StringBuffer类型传递

```

// 测试引用传递: StringBuffer
@org.junit.Test
public void method1() {
 StringBuffer str = new StringBuffer("公众号: Java面试通关手册");
 System.out.println(str);
 change1(str);
 System.out.println(str);
}

public static void change1(StringBuffer str) {
 str = new StringBuffer("abc");//输出: "公众号: Java面试通关手册"
 //str.append("欢迎大家关注");//输出: 公众号: Java面试通关手册欢迎大家关注
 //str.insert(3, "(编程)");//输出: 公众号(编程): Java面试通关手册

}

```

**结果：**

```

公众号: Java面试通关手册
公众号: Java面试通关手册

```

**解析：**

很多要这个时候要问了：StringBuffer创建的明明也是对象，那为什么输出结果依然是原来的值呢？

因为在 change1 方法内部我们是新建了一个StringBuffer对象，所以 str 指向了另外一个地址，相应的操作也同样是指向另外的地址的。

那么，如果将 change1 方法改成如下图所示，想必大家应该知道输出什么了，如果你还不知道，那可能就是我讲的有问题了，我反思（开个玩笑，上面程序中已经给出答案）：

```

public static void change1(StringBuffer str) {

 str.append("欢迎大家关注");
 str.insert(3, "(编程)");
}

```

```
 }
```

## 2. String类型传递

```
// 测试引用传递: String
@org.junit.Test
public void method2() {
 String str = new String("公众号: Java面试通关手册");
 System.out.println(str);
 change2(str);
 System.out.println(str);
}

public static void change2(String str) {
 // str="abc"; //输出: 公众号: Java面试通关手册
 str = new String("abc"); //输出: 公众号: Java面试通关手册
}
```

结果:

```
公众号: Java面试通关手册
公众号: Java面试通关手册
```

可以看到不论是执行 `str="abc;"` 还是 `str = new String("abc");` `str` 的输出的值都不变。按照我们上面讲“`StringBuffer`类型传递”的时候说的，`str="abc;"` 应该会让 `str` 的输出的值都不变。为什么呢？因为 `String` 在创建之后是不可变的。

## 3. 一道类似的题目

下面的程序输出是什么？

```
public class Demo {
 public static void main(String[] args) {
 Person p = new Person("张三");

 change(p);

 System.out.println(p.name);
 }

 public static void change(Person p) {
 Person person = new Person("李四");
 p = person;
 }
}

class Person {
 String name;

 public Person(String name) {
 this.name = name;
 }
}
```

很明显仍然会输出 `张三`。因为 `change` 方法中重新创建了一个 `Person` 对象。

那么，如果把 `change` 方法改为下图所示，输出结果又是什么呢？

```
public static void change(Person p) {
 p.name="李四";
```

```
}
```

答案我就不说了，我觉得大家如果认真看完上面的内容之后应该很清楚了。

## 二 ==与equals(重要)

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

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

- 情况1：类没有覆盖equals()方法。则通过equals()比较该类的两个对象时，等价于通过“==”比较这两个对象。
- 情况2：类覆盖了equals()方法。一般，我们都覆盖equals()方法来两个对象的内容相等；若它们的内容相等，则返回true(即，认为这两个对象相等)。

举个例子：

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

说明：

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

## 三 hashCode与equals（重要）

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

### hashCode () 介绍

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个int整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义在JDK的Object.java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。

```
/**
 * Returns a hash code value for the object. This method is
 * supported for the benefit of hash tables such as those provided by
```

```

* {@link java.util.HashMap}.
* <p>
* As much as is reasonably practical, the hashCode method defined by
* class {@code Object} does return distinct integers for distinct
* objects. (This is typically implemented by converting the internal
* address of the object into an integer, but this implementation
* technique is not required by the
* Java™ programming language.)
*
* @return a hash code value for this object.
* @see java.lang.Object#equals(java.lang.Object)
* @see java.lang.System#identityHashCode
*/
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：

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

## hashCode () 与equals () 的相关规定

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

## 为什么两个对象有相同的hashcode值，它们也不一定是相等的？

在这里解释一位小伙伴的问题。以下内容摘自《Head Fist Java》。

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

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

参考：

<https://blog.csdn.net/zhzhao999/article/details/53449504>

<https://www.cnblogs.com/skywang12345/p/3324958.html>

<https://www.cnblogs.com/skywang12345/p/3324958.html>

<https://www.cnblogs.com/Eason-S/p/5524837.html>



## String和StringBuffer、StringBuilder的区别是什么？String为什么是不可变的？

### String和StringBuffer、StringBuilder的区别

**可变性** String类中使用字符数组: `private final char value[]` 保存字符串，所以String对象是不可变的。StringBuilder与StringBuffer都继承自AbstractStringBuilder类，在AbstractStringBuilder中也是使用字符数组保存字符串，`char[]value`，这两种对象都是可变的。

#### 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：

- 如果要操作少量的数据用 = String
- 单线程操作字符串缓冲区下操作大量数据 = StringBuilder
- 多线程操作字符串缓冲区下操作大量数据 = StringBuffer

## String为什么是不可变的吗？

简单来说就是String类利用了final修饰的char类型数组存储字符，源码如下图所示：

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

## String真的是不可变的吗？

我觉得如果别人问这个问题的话，回答不可变就可以了。下面只是给大家看两个有代表性的例子：

### 1) String不可变但不代表引用不可以变

```
String str = "Hello";
str = str + " World";
System.out.println("str=" + str);
```

结果：

```
str=Hello World
```

解析：

实际上，原来String的内容是不变的，只是str由原来指向"Hello"的内存地址转为指向"Hello World"的内存地址而已，也就是说多开辟了一块内存区域给"Hello World"字符串。

### 2) 通过反射是可以修改所谓的“不可变”对象

```

// 创建字符串"Hello World", 并赋给引用s
String s = "Hello World";

System.out.println("s = " + s); // Hello World

// 获取String类中的value字段
Field valueFieldOfString = String.class.getDeclaredField("value");

// 改变value属性的访问权限
valueFieldOfString.setAccessible(true);

// 获取s对象上的value属性的值
char[] value = (char[]) valueFieldOfString.get(s);

// 改变value所引用的数组中的第5个字符
value[5] = '_';

System.out.println("s = " + s); // Hello_World

```

结果：

```

s = Hello World
s = Hello_World

```

解析：

用反射可以访问私有成员，然后反射出String对象中的value属性，进而改变通过获得的value引用改变数组的结构。但是一般我们不会这么做，这里只是简单提一下有这个东西。

## 什么是反射机制？反射机制的应用场景有哪些？

### 反射机制介绍

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

### 静态编译和动态编译

- 静态编译：在编译时确定类型，绑定对象
- 动态编译：运行时确定类型，绑定对象

### 反射机制优缺点

- 优点：运行期类型的判断，动态加载类，提高代码灵活度。
- 缺点：性能瓶颈：反射相当于一系列解释操作，通知JVM要做的事情，性能比直接的java代码要慢很多。

### 反射的应用场景

反射是框架设计的灵魂。

在我们平时的项目开发过程中，基本上很少会直接使用到反射机制，但这不能说明反射机制没有用，实际上有很多设计、开发都与反射机制有关，例如模块化的开发，通过反射去调用对应的字节码；动态代理设计模式也采用了反射机制，还有我们日常使用的Spring / Hibernate等框架也大量使用到了反射机制。

举例：①我们在使用JDBC连接数据库时使用Class.forName()通过反射加载数据库的驱动程序；②Spring框架也用到很多反射机制，最经典的就是xml的配置模式。Spring通过XML配置模式装载Bean的过程：1)将程序内所有XML或Properties配置文件加载入内存中；2)Java类里面解析xml或properties里面的内容，得到对应实体类的字节码字符串以

及相关的属性信息; 3)使用反射机制, 根据这个字符串获得某个类的Class实例; 4)动态配置实例的属性

推荐阅读:

- [Reflection: Java反射机制的应用场景](#)
- [Java基础之一反射 \(非常重要\)](#)

## 什么是JDK?什么是JRE? 什么是JVM? 三者之间的联系与区别

这几个是Java中很基本很基本的东西, 但是我相信一定还有很多人搞不清楚! 为什么呢? 因为我们大多数时候在使用现成的编译工具以及环境的时候, 并没有去考虑这些东西。

**JDK:** 顾名思义它是给开发者提供的开发工具箱, 是给程序开发者用的。它除了包括完整的JRE (Java Runtime Environment), Java运行环境, 还包含了其他供开发者使用的工具包。

**JRE:** 普通用户只需要安装JRE (Java Runtime Environment) 来运行Java程序。而程序开发者必须安装JDK来编译、调试程序。

**JVM:** 当我们运行一个程序时, JVM负责将字节码转换为特定机器代码, JVM提供了内存管理/垃圾回收和安全机制等。这种独立于硬件和操作系统, 正是java程序可以一次编写多处执行的原因。

区别与联系:

1. JDK用于开发, JRE用于运行java程序 ;
2. JDK和JRE中都包含JVM ;
3. JVM是java编程语言的核心并且具有平台独立性。

## 什么是字节码? 采用字节码的最大好处是什么?

先看下java中的编译器和解释器:

Java中引入了虚拟机的概念, 即在机器和编译程序之间加入了一层抽象的虚拟的机器。这台虚拟的机器在任何平台上都提供给编译程序一个的共同的接口。编译程序只需要面向虚拟机, 生成虚拟机能够理解的代码, 然后由解释器来将虚拟机代码转换为特定系统的机器码执行。在Java中, 这种供虚拟机理解的代码叫做 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。每一种平台的解释器是不同的, 但是实现的虚拟机是相同的。Java源程序经过编译器编译后变成字节码, 字节码由虚拟机解释执行, 虚拟机将每一条要执行的字节码送给解释器, 解释器将其翻译成特定机器上的机器码, 然后在特定的机器上运行。这也就是解释了Java的编译与解释并存的特点。

Java源代码---->编译器---->jvm可执行的Java字节码(即虚拟指令)--->jvm--->jvm中解释器----->机器可执行的二进制机器码--->程序运行。

采用字节码的好处:

Java语言通过字节码的方式, 在一定程度上解决了传统解释型语言执行效率低的问题, 同时又保留了解释型语言可移植的特点。所以Java程序运行时比较高效, 而且, 由于字节码并不专对一种特定的机器, 因此, Java程序无须重新编译便可可在多种不同的计算机上运行。

## Java和C++的区别

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

- 都是面向对象的语言, 都支持封装、继承和多态
- Java不提供指针来直接访问内存, 程序内存更加安全
- Java的类是单继承的, C++支持多重继承; 虽然Java的类不可以多继承, 但是接口可以多继承。
- Java有自动内存管理机制, 不需要程序员手动释放无用内存

## 接口和抽象类的区别是什么？

1. 接口的方法默认是public，所有方法在接口中不能有实现，抽象类可以有非抽象的方法
2. 接口中的实例变量默认是final类型的，而抽象类中则不一定
3. 一个类可以实现多个接口，但最多只能实现一个抽象类
4. 一个类实现接口的话要实现接口的所有方法，而抽象类不一定
5. 接口不能用new实例化，但可以声明，但是必须引用一个实现该接口的对象 从设计层面来说，抽象是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

## 成员变量与局部变量的区别有那些？

1. 从语法形式上，看成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被public,private,static等修饰符所修饰，而局部变量不能被访问控制修饰符及static所修饰；但是，成员变量和局部变量都能被final所修饰；
2. 从变量在内存中的存储方式来看，成员变量是对象的一部分，而对象存在于堆内存，局部变量存在于栈内存
3. 从变量在内存中的生存时间上看，成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动消失。
4. 成员变量如果没有被赋初值，则会自动以类型的默认值而赋值（一种情况例外被final修饰但没有被static修饰的成员变量必须显示地赋值）；而局部变量则不会自动赋值。

## 重载和重写的区别

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

**重写：**发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，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；如果父类方法访问修饰符为private则子类就不能重写该方法。

## 字符型常量和字符串常量的区别

1) 形式上：字符常量是单引号引起的一个字符 字符串常量是双引号引起的若干个字符 2) 含义上：字符常量相当于一个整形值(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 字符常量只占一个字节 字符串常量占若干个字节(至少一个字符结束标志)

## 1. 简述线程、程序、进程的基本概念。以及他们之间关系是什么？

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

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

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

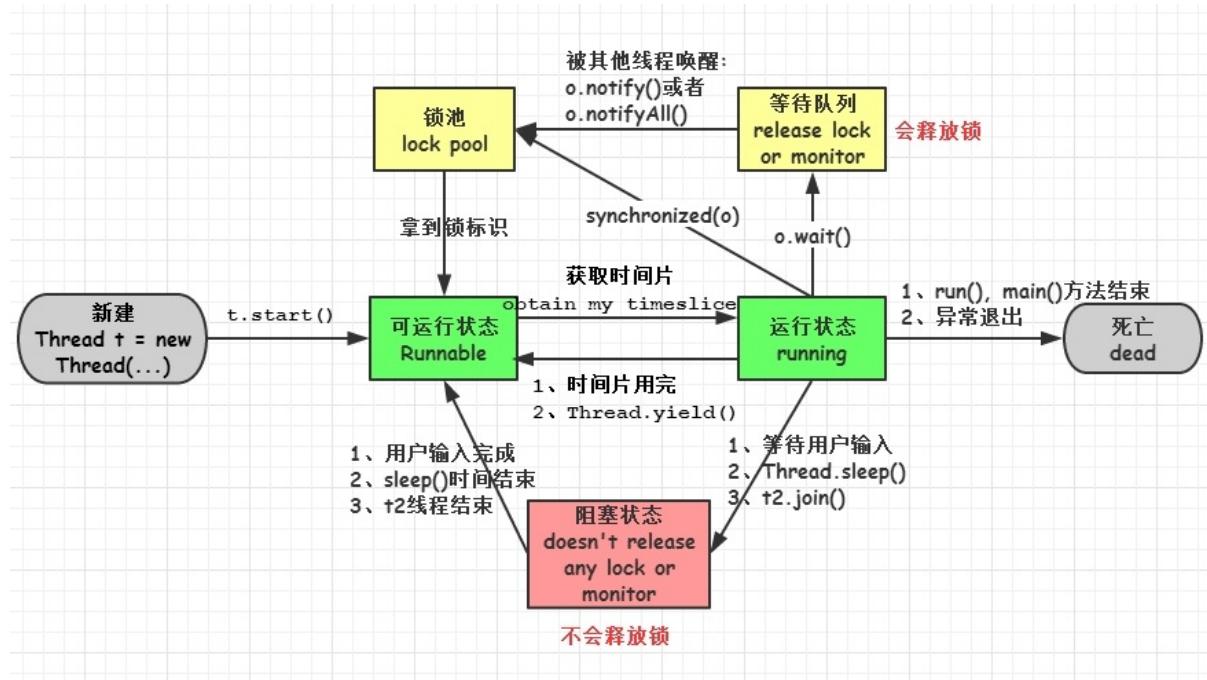
线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。从另一角度来说，进程属于操作系统的范畴，主要是同一段时间内，可以同时执行一个以上的程序，而线程则是在同一程序内几乎同时执行一个以上的程序段。

线程上下文的切换比进程上下文切换要快很多

- 进程切换时，涉及到当前进程的CPU环境的保存和新被调度运行进程的CPU环境的设置。
- 线程切换仅需要保存和设置少量的寄存器内容，不涉及存储管理方面的操作。

## 2. 线程有哪些基本状态？这些状态是如何定义的？

1. 新建(new): 新创建了一个线程对象。
2. 可运行(runnable): 线程对象创建后，其他线程(比如main线程)调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu的使用权。
3. 运行(running): 可运行状态(runnable)的线程获得了cpu时间片(timeslice)，执行程序代码。
4. 阻塞(block): 阻塞状态是指线程因为某种原因放弃了cpu使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice转到运行(running)状态。阻塞的情况分三种：
  - (一). 等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waiting queue)中。
  - (二). 同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。
  - (三). 其他阻塞：运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。
5. 死亡(dead): 线程run()、main()方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。



备注：可以用早起坐地铁来比喻这个过程（下面参考自牛客网某位同学的回答）：

1. 还没起床：sleeping
2. 起床收拾好了，随时可以坐地铁出发：Runnable
3. 等地铁来：Waiting
4. 地铁来了，但要排队上地铁：I/O阻塞
5. 上了地铁，发现暂时没座位：synchronized阻塞
6. 地铁上找到座位：Running
7. 到达目的地：Dead

### 3. 何为多线程？

多线程就是多个线程同时运行或交替运行。单核CPU的话是顺序执行，也就是交替运行。多核CPU的话，因为每个CPU有自己的运算器，所以在多个CPU中可以同时运行。

### 4. 为什么多线程是必要的？

1. 使用线程可以把占据长时间的程序中的任务放到后台去处理。
2. 用户界面可以更加吸引人，这样比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度。
3. 程序的运行速度可能加快。

### 5 使用多线程常见的三种方式

#### ①继承Thread类

MyThread.java

```
public class MyThread extends Thread {
 @Override
 public void run() {
```

```
 super.run();
 System.out.println("MyThread");
 }
}
```

### Run.java

```
public class Run {
 public static void main(String[] args) {
 MyThread mythread = new MyThread();
 mythread.start();
 System.out.println("运行结束");
 }
}
```

运行结果：

**MyThread**

从上面的运行结果可以看出：线程是一个子任务，CPU以不确定的方式，或者说是以随机的时间来调用线程中的run方法。

## ②实现Runnable接口

推荐实现Runnable接口方式开发多线程，因为Java单继承但是可以实现多个接口。

### <MyRunnable.java

```
public class MyRunnable implements Runnable {
 @Override
 public void run() {
 System.out.println("MyRunnable");
 }
}
```

### Run.java

```
public class Run {
 public static void main(String[] args) {
 Runnable runnable=new MyRunnable();
 Thread thread=new Thread(runnable);
 thread.start();
 System.out.println("运行结束! ");
 }
}
```

运行结果：

**MyRunnable**

从上面的运行结果可以看出：线程是一个子任务，CPU以不确定的方式，或者说是以随机的时间来调用线程中的run方法。

## ③使用线程池

在《阿里巴巴Java开发手册》“并发处理”这一章节，明确指出线程资源必须通过线程池提供，不允许在应用中自行显示创建线程。

为什么呢？

使用线程池的好处是减少在创建和销毁线程上所消耗的时间以及系统资源开销，解决资源不足的问题。如果不使用线程池，有可能会造成系统创建大量同类线程而导致消耗完内存或者“过度切换”的问题。

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

**Executors** 返回线程池对象的弊端如下：

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

对于线程池感兴趣的可以查看我的这篇文章：[《Java多线程学习（八）线程池与Executor 框架》](#) 点击阅读原文即可查看到该文章的最新版。

## 6 线程的优先级

每个线程都具有各自的优先级，线程的优先级可以在程序中表明该线程的重要性，如果有很多线程处于就绪状态，系统会根据优先级来决定首先使哪个线程进入运行状态。但这个并不意味着低优先级的线程得不到运行，而只是它运行的几率比较小，如垃圾回收机制线程的优先级就比较低。所以很多垃圾得不到及时的回收处理。

线程优先级具有继承特性。比如A线程启动B线程，则B线程的优先级和A是一样的。

线程优先级具有随机性。也就是说线程优先级高的不一定每一次都先执行完。

Thread类中包含的成员变量代表了线程的某些优先级。如**Thread.MIN\_PRIORITY**（常数1），**Thread.NORM\_PRIORITY**（常数5），**Thread.MAX\_PRIORITY**（常数10）。其中每个线程的优先级都在**Thread.MIN\_PRIORITY**（常数1）到**Thread.MAX\_PRIORITY**（常数10）之间，在默认情况下优先级都是**Thread.NORM\_PRIORITY**（常数5）。

学过操作系统这门课程的话，我们可以发现多线程优先级或多或少借鉴了操作系统对进程的管理。

## 7 Java多线程分类

### 用户线程

运行在前台，执行具体的任务，如程序的主线程、连接网络的子线程等都是用户线程

### 守护线程

运行在后台，为其他前台线程服务.也可以说守护线程是JVM中非守护线程的“佣人”。

- **特点：**一旦所有用户线程都结束运行，守护线程会随JVM一起结束工作
- **应用：**数据库连接池中的检测线程，JVM虚拟机启动后的检测线程
- **最常见的守护线程：**垃圾回收线程

#### 如何设置守护线程？

可以通过调用 Thread 类的 `setDaemon(true)` 方法设置当前的线程为守护线程>。

注意事项：

1. `setDaemon(true)`必须在`start ()` 方法前执行，否则会抛出`IllegalThreadStateException`异常
2. 在守护线程中产生的新线程也是守护线程
3. 不是所有的任务都可以分配给守护线程来执行，比如读写操作或者计算逻辑

## 8 sleep()方法和wait()方法简单对比

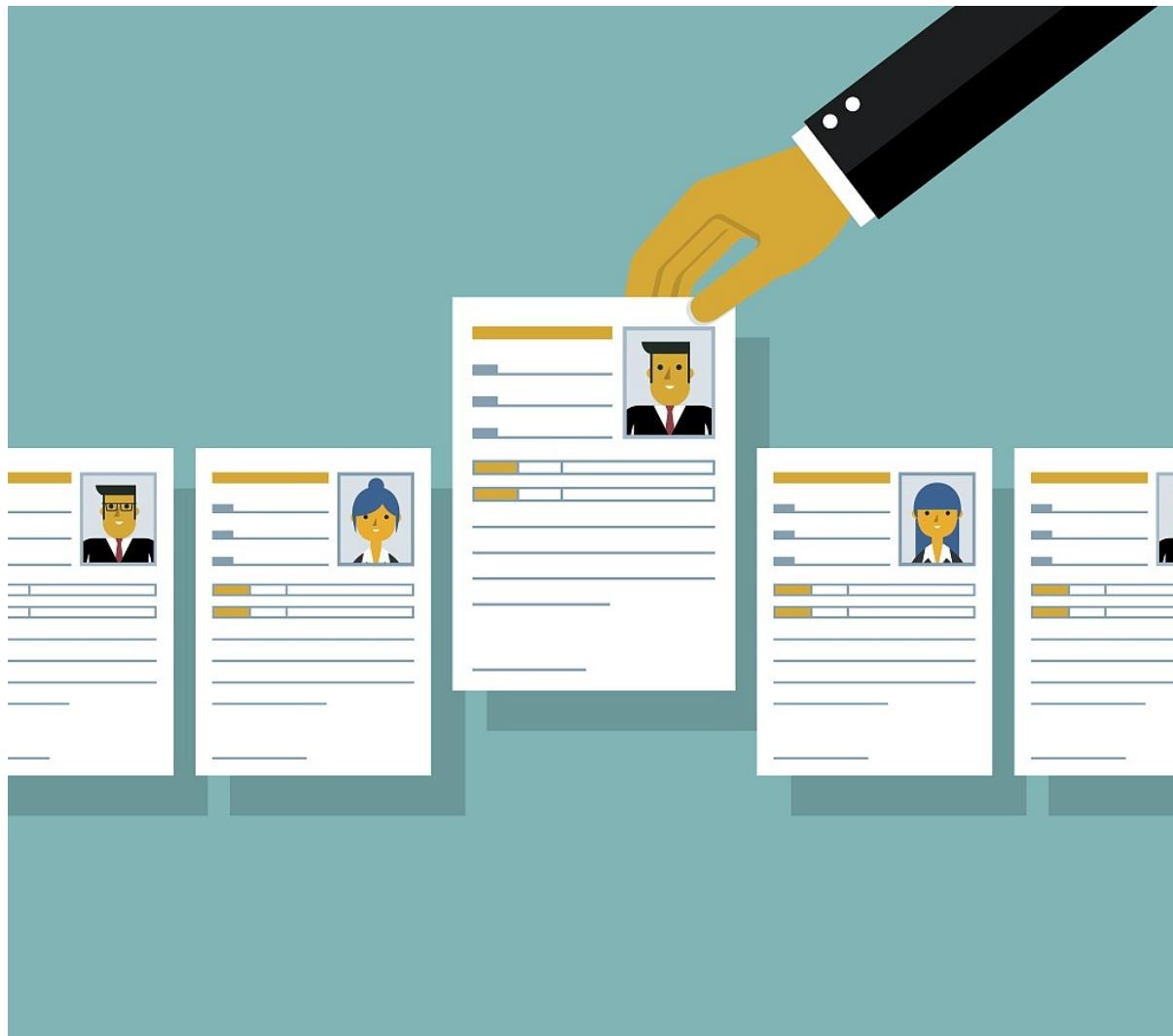
- 两者最主要的区别在于：sleep方法没有释放锁，而wait方法释放了锁。
- 两者都可以暂停线程的执行。
- Wait通常被用于线程间交互/通信，sleep通常被用于暂停执行。
- wait()方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的notify()或者notifyAll()方法。sleep()方法执行完成后，线程会自动苏醒。

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

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

new一个Thread，线程进入了新建状态；调用start()方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。start()会执行线程的相应准备工作，然后自动执行run()方法的内容，这是真正的多线程工作。而直接执行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，而run方法只是thread的一个普通方法调用，还是在主线程里执行。



俗话说的好：“工欲善其事，必先利其器”。准备一份好的简历对于能不能找到一份好工作起到了至关重要的作用。

## 六 如何写自己的简历？

一份好的简历可以在整个申请面试以及面试过程中起到非常好的作用。

### 6.1 为什么说简历很重要？

假如你是网申，你的简历必然会经过HR的筛选，一张简历HR可能也就花费10秒钟看一下，然后HR就会决定你这一关是Fail还是Pass。

假如你是内推，如果你的简历没有什么优势的话，就算是内推你的人再用心，也无能为力。

另外，就算你通过了筛选，后面的面试中，面试官也会根据你的简历来判断你究竟是否值得他花费很多时间去面试。

### 6.2 这3点你必须知道

1. 大部分应届生找工作的硬伤是没有工作经验或实习经历；
2. 写在简历上的东西一定要慎重，这可能是面试官大量提问的地方；
3. 将自己的项目经历完美的展示出来非常重要。

## 6.3 两大法则了解一下

目前写简历的方式有两种普遍被认可，一种是 STAR，一种是 FAB。

**STAR法则 (Situation Task Action Result) :**

- **Situation:** 事情是在什么情况下发生；
- **Task:** 你是如何明确你的任务的；
- **Action:** 针对这样的情况分析，你采用了什么行动方式；
- **Result:** 结果怎样，在这样的情况下你学习到了什么。

**FAB 法则 (Feature Advantage Benefit) :**

- **Feature:** 是什么；
- **Advantage:** 比别人好在哪些地方；
- **Benefit:** 如果雇佣你，招聘方会得到什么好处。

## 6.4 项目经历怎么写？

简历上有一两个项目经历很正常，但是真正能把项目经历很好的展示给面试官的非常少。对于项目经历大家可以考虑从如下几点来写：

1. 对项目整体设计的一个感受
2. 在这个项目中你负责了什么、做了什么、担任了什么角色
3. 从这个项目中你学会了那些东西，使用到了那些技术，学会了那些新技术的使用
4. 另外项目描述中，最好可以体现自己的综合素质，比如你是如何协调项目组成员协同开发的或者在遇到某一个棘手的问题的时候你是如何解决的。

## 6.5 专业技能该怎么写？

先问一下你自己会什么，然后看看你意向的公司需要什么。一般HR可能并不太懂技术，所以他在筛选简历的时候可能就盯着你专业技能的关键词来看。对于公司有要求而你不会的技能，你可以花几天时间学习一下，然后在简历上可以写上自己了解这个技能。比如你可以这样写：

- Dubbo: 精通
- Spring: 精通
- Docker: 掌握
- SOA分布式开发：掌握
- Spring Cloud: 了解

## 6.6 开源程序员简历模板分享

分享一个Github上开源的程序员简历模板。包括PHP程序员简历模板、iOS程序员简历模板、Android程序员简历模板、Web前端程序员简历模板、Java程序员简历模板、C/C++程序员简历模板、NodeJS程序员简历模板、架构师简历模板以及通用程序员简历模板。Github地址：<https://github.com/geekcompany/ResumeSample>

## 6.7 其他的一些小tips

1. 尽量避免主观表述，少一点语义模糊的形容词，尽量要简洁明了，逻辑结构清晰。
2. 注意排版（不需要花花绿绿的），尽量使用Markdown语法。
3. 如果自己有博客或者个人技术栈点的话，写上去会为你加分很多。
4. 如果自己的Github比较活跃的话，写上去也会为你加分很多。
5. 注意简历真实性，一定不要写自己不会的东西，或者带有欺骗性的内容
6. 项目经历建议以时间倒序排序，另外项目经历不在于多，而在于有亮点。

7. 如果内容过多的话，不需要非把内容压缩到一页，保持排版干净整洁就可以了。
8. 简历最后最好能加上：“感谢您花时间阅读我的简历，期待能有机会和您共事。”这句话，显的你会很有礼貌。

本文摘自我的Gitchat: [《从应届程序员角度分析如何备战大厂面试》](#)。

## Markdown 简历模板样式一览

## 联系方式

---

- 手机：
- Email：
- 微信：

## 个人信息

---

- 姓名/性别/出生日期
- 本科/xxx计算机系xxx专业/英语六级
- 技术博客：<http://snailclimb.top/>
- 荣誉奖励：获得了什么奖（获奖时间）
- Github：<https://github.com/Snailclimb>
- Github Resume：<http://resume.github.io/?Snailclimb>
- 期望职位：Java 研发程序员/大数据工程师(Java后台开发为首选)
- 期望城市：xxd城市

## 项目经历

---

### xxx项目

#### 项目描述

介绍该项目是做什么的、使用到了什么技术以及你对项目整体设计的一个感受

#### 责任描述

主要可以从下面三点来写：

1. 在这个项目中你负责了什么、做了什么、担任了什么角色
2. 从这个项目中你学会了那些东西，使用到了那些技术，学会了那些新技术的使用
3. 另外项目描述中，最好可以体现自己的综合素质，比如你是如何协调项目组成员协同开发的或者在遇到某一个棘手的问题的时候你是如何解决的。

## 开源项目和技术文章

---

### 开源项目

- [Java-Guide](#)：一份涵盖大部分Java程序员所需要掌握的核心知识。Star:3.9K Fork:0.9k。

### 技术文章推荐

- 可能是把Java内存区域讲的最清楚的一篇文章
- 搞定JVM垃圾回收就是这么简单
- 前端&后端程序员必备的Linux基础知识
- 可能是把Docker的概念讲的最清楚的一篇文章

## 校园经历（可选）

---

### 2016-2017

担任学校社团-致深社副会长，主要负责团队每周活动的组建以及每周例会的主持。

### 2017-2018

担任学校传媒组织：“长江大学在线信息传媒”的副站长以及安卓组成员。主要负责每周例会主持、活动策划以及学校校园通APP的研发工作。

## 技能清单

---

以下均为我熟练使用的技能

- Web开发：PHP/Hack/Node
- Web框架：ThinkPHP/Yaf/Yii/Laravel/LazyPHP
- 前端框架：Bootstrap/AngularJS/EmberJS/HTML5/Cocos2dJS/ionic
- 前端工具：Bower/Gulp/Sass/LeSS/PhoneGap
- 数据库相关：MySQL/PgSQL/PDO/SQLite
- 版本管理、文档和自动化部署工具：Svn/Git/PHPDoc/Phing/Composer
- 单元测试：PHPUnit/SimpleTest/Qunit
- 云和开放平台：SAE/BAE/AWS/微博开放平台/微信应用开发

## 自我评价（可选）

---

自我发挥。切记不要过度自夸！！！

感谢您花时间阅读我的简历，期待能有机会和您共事。

可以看到我把联系方式放在第一位，因为公司一般会与你联系，所以把联系方式放在第一位也是为了方便联系考虑。

## 为什么要用 Markdown 写简历？

Markdown 语法简单，易于上手。使用正确的 Markdown 语言写出来的简历不论是在排版还是格式上都比较干净，易于阅读。另外，使用 Markdown 写简历也会给面试官一种你比较专业的感觉。

除了这些，我觉得使用 Markdown 写简历可以很方便将其与 PDF、HTML、PNG 格式之间转换。后面我会介绍到转换方法，只需要一条命令你就可以实现 Markdown 到 PDF、HTML 与 PNG 之间的无缝切换。

下面的一些内容我在之前的一篇文章中已经提到过，这里再说一遍，最后会分享如何实现 Markdown 到 PDF、HTML、PNG 格式之间转换。

## 为什么说简历很重要？

假如你是网申，你的简历必然会经过 HR 的筛选，一张简历 HR 可能也就花费 10 秒钟看一下，然后 HR 就会决定你这一关是 Fail 还是 Pass。

假如你是内推，如果你的简历没有什么优势的话，就算是内推你的人再用心，也无能为力。

另外，就算你通过了筛选，后面的面试中，面试官也会根据你的简历来判断你究竟是否值得他花费很多时间去面试。

## 写简历的两大法则

目前写简历的方式有两种普遍被认可，一种是 STAR，一种是 FAB。

**STAR 法则（Situation Task Action Result）：**

- **Situation:** 事情是在什么情况下发生；
- **Task:** 你是如何明确你的任务的；
- **Action:** 针对这样的情况分析，你采用了什么行动方式；
- **Result:** 结果怎样，在这样的情况下你学习到了什么。

**FAB 法则（Feature Advantage Benefit）：**

- **Feature:** 是什么；
- **Advantage:** 比别人好在哪些地方；
- **Benefit:** 如果雇佣你，招聘方会得到什么好处。

## 项目经历怎么写？

简历上有一两个项目经历很正常，但是真正能把项目经历很好的展示给面试官的非常少。对于项目经历大家可以考虑从如下几点来写：

1. 对项目整体设计的一个感受
2. 在这个项目中你负责了什么、做了什么、担任了什么角色
3. 从这个项目中你学会了那些东西，使用到了那些技术，学会了那些新技术的使用
4. 另外项目描述中，最好可以体现自己的综合素质，比如你是如何协调项目组成员协同开发的或者在遇到某一个棘手的问题的时候你是如何解决的。

## 专业技能该怎么写？

先问一下你自己会什么，然后看看你意向的公司需要什么。一般HR可能并不太懂技术，所以他在筛选简历的时候可能就盯着你专业技能的关键词来看。对于公司有要求而你不会的技能，你可以花几天时间学习一下，然后在简历上可以写上自己了解这个技能。比如你可以这样写：

- Dubbo: 精通
- Spring: 精通
- Docker: 掌握
- SOA分布式开发：掌握
- Spring Cloud: 了解

## 简历模板分享

开源程序员简历模板：<https://github.com/geekcompany/ResumeSample>（包括PHP程序员简历模板、iOS程序员简历模板、Android程序员简历模板、Web前端程序员简历模板、Java程序员简历模板、C/C++程序员简历模板、NodeJS程序员简历模板、架构师简历模板以及通用程序员简历模板）

上述简历模板的改进版本：<https://github.com/Snailclimb/Java-Guide/blob/master/面试必备/简历模板.md>

## 其他的一些小tips

1. 尽量避免主观表述，少一点语义模糊的形容词，尽量要简洁明了，逻辑结构清晰。
2. 注意排版（不需要花花绿绿的），尽量使用Markdown语法。
3. 如果自己有博客或者个人技术栈点的话，写上去会为你加分很多。
4. 如果自己的Github比较活跃的话，写上去也会为你加分很多。
5. 注意简历真实性，一定不要写自己不会的东西，或者带有欺骗性的内容
6. 项目经历建议以时间倒序排序，另外项目经历不在于多，而在于有亮点。
7. 如果内容过多的话，不需要非把内容压缩到一页，保持排版干净整洁就可以了。
8. 简历最后最好能加上：“感谢您花时间阅读我的简历，期待能有机会和您共事。”这句话，显的你会很有礼貌。

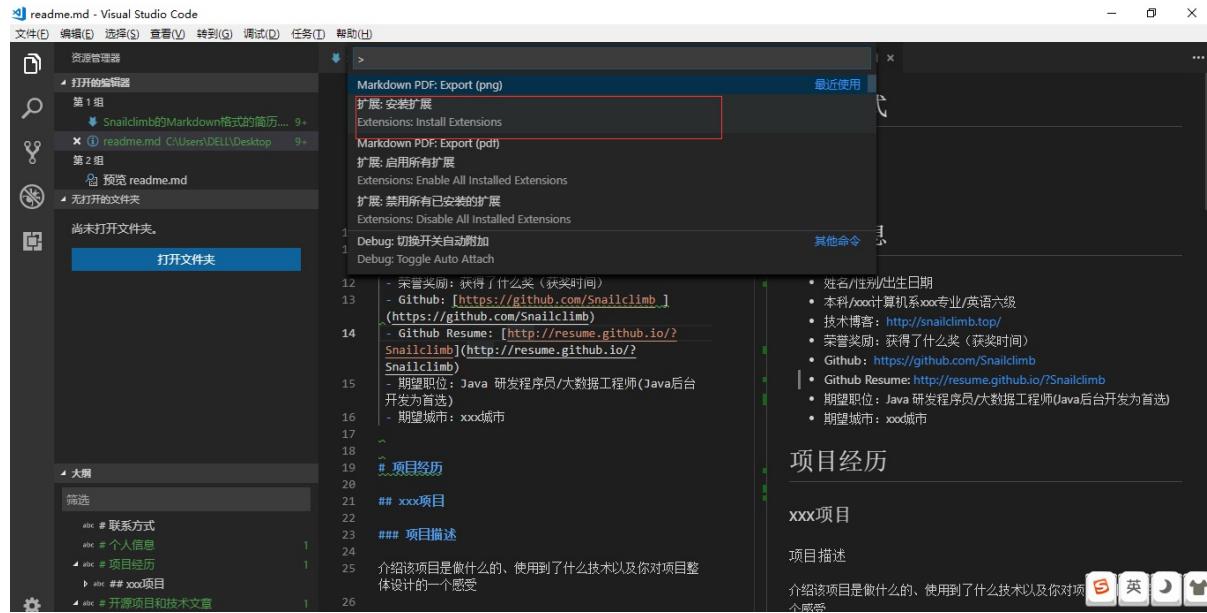
我们刚刚讲了很多关于如何写简历的内容并且分享了一份 Markdown 格式的简历文档。下面我们来看看如何实现 Markdown 到 HTML格式、PNG格式之间转换。

## Markdown 到 HTML格式、PNG格式之间转换

网上很难找到一个比较方便并且效果好的转换方法，最后我是通过 Visual Studio Code 的 Markdown PDF 插件完美解决了这个问题！

### 安装 Markdown PDF 插件

① 打开Visual Studio Code，按快捷键 F1，选择安装扩展选项



## ② 搜索“Markdown PDF”插件并安装，然后重启

This screenshot shows the VS Code Extension Marketplace. On the left, a sidebar lists various extensions like 'Markdown All in One', 'markdownlint', and 'Markdown Preview Enhanced'. In the center, the 'Markdown PDF' extension by 'yzane' is displayed. It has a download count of 330,552, a five-star rating, and is marked as '已安装' (Installed). The extension details page shows it converts Markdown files to PDF, HTML, PNG, or JPEG. A red arrow points from the extension's page back to the search results in the sidebar.

## 使用方法

随便打开一份 Markdown 文件 点击F1，然后输入export即可！

This screenshot shows the Visual Studio Code interface again. The command palette shows the search term 'export' and the 'Markdown PDF: Export (png)' command is highlighted with a red box. Below the search bar, other export options like 'Markdown PDF: Export (pdf)', 'Markdown PDF: Export (all: pdf, html, png, jpeg)', and 'Markdown PDF: Export (html)' are listed. The main code editor area shows the same resume document. A red arrow points from the export options in the command palette down to the list of export formats in the editor.

