
title: 面试总结

tags:

- 面试

categories:

- 面试

copyright: true

comments: true

lang: zh-Hans

updated:

sticky: 0

本文档收录了在面试期间对Java及其他计算机知识的总结，希望秋招能斩获一个好OFFER！



<!--more-->

Java基础

谈谈你对Java的理解/Java有哪些优点？

1. 纯面向对象语言
2. 平台无关性
3. Java提供了许多内置类库
4. Java提供了对Web应用开发的支持
5. 具有较好的安全性和健壮性
6. 去除了C/C++语言中难以理解、容易混淆的特性：如头文件、指针、结构体、单元运算符、虚拟基础类、多重继承等

Java与C++有什么异同？

1. 都是是一门纯面向对象语言
2. Java没有C++中的指针、结构体、文件、单元运算、虚拟基础类、多重继承等概念。
3. 由于没有指针等概念，Java的内存管理将由JVM自动完成，使得开发人员从中解放出来。
4. Java是一门平台无关的解释型语言，Java可以一次编译，然后在不同平台的JVM上运行。而C++与平台有关，是编译型语言，跨平台运行需要重新编译。
5. Java不支持自动强制类型转换，需要开发人员显示声明。

解释一下面向对象的三个特性？

1. **封装：**即将客观事物封装成抽象类，它有一组数据变量以及方法，并可以对它们的访问性进行控制，可以将对不可信信息进行隐藏。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。
2. **继承：**是指使得一种对象获得另一种对象的数据和方法，它支持按级分类的概念。它是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。继承的过程，就是从一般到特殊的过程。要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。继承概念的实现方式有二类：实现继承与接口继承。实现继承是指直接使用基类的属性和方法而无需额外编码的能力；接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力。
3. **多态：**通常是指同名方法中有多种实现方式。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。

面向对象的六大基本原则？

1. 单一职责原则SRP(Single Responsibility Principle):

指一个类的功能尽可能单一，不要保罗万象，否则效率不高。

2. 开放封闭原则OCP(Open – Close Principle):

指模块设计的时候应该对拓展开放，对修改内部代码封闭。

3. 里式替换原则LSP(the Liskov Substitution Principle LSP):

指子类对象应该可以替换父类对象，并可以出现在任何父类对象可以出现的地方。

4. 依赖倒置原则DIP(the Dependency Inversion Principle DIP):

指具体依赖抽象，上层依赖下层。假设B模块（下层）依赖于A模块（上层），但B需要使用到A的功能，这个时候，B不应当直接使用A中的具体类：而应当由B定义一抽象接口，并由A来实现这个抽象接口，B只使用这个抽象接口：这样就达到了依赖倒置的目的，B也解除了对A的依赖，反过来是A依赖于B定义的抽象接口。通过上层模块难以避免依赖下层模块，假如B也直接依赖A的实现，那么就可能造成循环依赖。一个常见的问题就是编译A模块时需要直接包含到B模块的cpp文件，而编译B时同样要直接包含到A的cpp文件。

5. 接口分离原则ISP(the Interface Segregation Principle ISP):

模块间要通过抽象接口隔离开，而不是通过具体的类强耦合起来。

6. 最小知道原则 DP (Demeter Principle):

设计一个系统，不管是任何对象，都要注意它所交互的类有哪些，并注意它们是如何交互的。如何才能避免违反该原则呢？我们应该尽可能做到只调用属于以下范围的方法：

- 该对象本身的方法
- 被当成方法的参数而传递进来的对象的方法
- 此方法创建或实例化的任何对象的方法
- 对象的任何组件的方法

说说你对面向对象编程的理解？

面向对象编程，是指将现实世界的事物通过封装抽象成为一个个对象，它们可以有自己的数据与方法操作，并对不可信信息进行隐藏，从而达到模拟现实世界的效果。面向对象可以理解成对待每一个问题，都是首先要确定这个问题由几个部分组成，而每一个部分其实就是一个对象。然后再分别设计这些对象，最后得到整个程序。

介绍一下Java多态？

多态的优点？

1. 消除类型之间的耦合关系
2. 可替换性
3. 可扩充性
4. 接口性
5. 灵活性
6. 简化性

多态的必要条件?

1. 继承
2. 重写
3. 父类引用指向子类: E.g. Parent p = new Child();
Java多态的实现方式?
4. 重写
5. 接口实现
6. 抽象类和抽象方法

介绍一下Java基本数据类型及其长度?

TYPE	BITS	Maximum	Minimum	NOTE
boolean	1	-	-	-
byte	8	-128	127	首位符号位
char	16	Unicode 0	Unicode $2^{16}-1$	-
short	16	-2^{15}	$2^{15}-1$	首位符号位
int	32	-2^{31}	$2^{31}-1$	首位符号位
float	32	IEEE754 \u0000	IEEE754 \uffff	单精度、不能表示精确值
double	64	IEEE754	IEEE754	双精度、不能表示精确值
long	64	-2^{65}	$2^{65}-1$	首位符号位

基本类型和包装类型哪个占用资源多? 为什么?

基本类型的值就是一个数字, 一个字符, 或者一个布尔值, 存放在栈空间中。

包装类型是一个对象的引用, 值是指向内存空间的引用, 就是地址。所指向的内存中保存着

变量所表示的一个值或一组值，存放在堆空间中。

由于包装类型中还有一组方法函数，因此包装类型所占的资源多。

介绍一下Java装箱和拆箱机制？

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。

在装箱过程中，注意Java会对一定范围内（例如int 在[-128, 128]）会有缓存机制，在缓存范围内的装箱不会创建新的对象，而是直接引用相同对象。

装箱操作会创建对象，频繁的装箱操作会消耗许多内存，影响性能，所以可以避免装箱的时候应该尽量避免。

Integer有符号吗？

Java没有无符号类型，都是有符号类型的数据类型。

0.1*3 == 0.3 结果是什么？ 1*0.3 == 0.3结果是什么？为什么？

$3*0.1 == 0.3$ 返回值 false

$1*0.3 == 0.3$ 返回值 true

因为对于十进制数值系统（就是我们现实中使用的），它只能表示以进制数的质因子为分母的分数。10 的质因子有 2 和 5。因此 $1/2$ 、 $1/4$ 、 $1/5$ 、 $1/8$ 和 $1/10$ 都可以精确表示，因为这些分母只使用了10的质因子。相反， $1/3$ 、 $1/6$ 和 $1/7$ 都是循环小数，因为它们的分母使用了质因子 3 或者 7。二进制下（进制数为2），只有一个质因子，即2。因此只能精确表示分母质因子是2的分数。

二进制中， $1/2$ 、 $1/4$ 和 $1/8$ 都可以被精确表示。但是， $1/5$ 或者 $1/10$ 就变成了循环小数。所以，在十进制中能够精确表示的 0.1 与 0.2 ($1/10$ 与 $1/5$)，到了计算机所使用的二进制数值系统中，就变成了循环小数。当你对这些循环小数进行数学运算时，并将二进制数据转换成人类可读的十进制数据时，会对小数尾部进行截断处理。

Java中如何精确表示浮点数？

Java通过Decimal和BigDecimal类来精确表示浮点数。

Java char 编码？

Java 中 `char` 是16位 UTF16 编码的

实例方法和静态方法加载区别？

首先解类编译的class文件中字节码的方法调用指令

1. `invokestatic`: 调用静态方法
2. `invokespecial`: 调用实例构造器方法，私有方法。
3. `invokevirtual`: 调用所有的虚方法，说明运行期间会到方法表中去调用真实指向的方法。
4. `invokeinterface`: 调用接口方法，会在运行时再确定一个实现此接口的对象。
5. `invokedynamic`: 先在运行时动态解析出调用点限定符所引用的方法，然后再执行该方法。

对于**实例方法**，在编译阶段会调用`invokevirtual`指令，它会根据实例的具体类型查找对应的方法表并进行方法绑定（方法重写发生在此时）；

对于**静态方法**，在编译阶段会调用`invokestatic`指令，它不会查找方法表，而是直接绑定方法，即静态方法的执行只看静态类型，而与实际类型无关，这也是为什么静态方法不能被重写。

重写和重载区别

重写：发生在父子类中，方法名、参数列表必须相同，子类同名同参函数的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。**只有实例方法可以被重写，静态方法不能被重写！**因为静态方法的调用的是通过在编译器静态绑定的，而实例方法的调用是在运行时动态绑定的，2者的调用的方式不同，所以二者只能存在其一，否则会存在歧义！但当父类和子类都存在同名同参函数时，因为调用方式一致，不会像上面造成歧义，虽然父类和子类都定义了同样的函数，但是编译器会根据对象的静态类型激活对应的静态方法的引用，造成了重写的假象，实则不是重写！

重载：放在同一个类中，方法名相同，但参数类型、个数、顺序不同的函数，同时，它们的返回值类型和访问控制符可以不同。**重载发生在编译阶段**。

super与this异同

super: 指向当前对象的父类，它是一个引用但是却没办法输出。

super(): 父类构造方法。如果子类构造方法没有显示地调用父类的构造方法，那么编译器会自动为它加上一个默认的super()方法调用。如果父类没有默认的无参构造方法，编译器就会报错。super()语句必须在构造方法第一行调用，否则编译器会报错。

this: this是一个引用类型，在堆中的每一个Java对象上都有this，this保存内存地址指向自身。

this(): 当前类构造方法。通过this()调用其他构造方法时，必须放在第一行，否则编译器会报错。且在构造方法中，只能通过this()调用一次其他构造方法。

注意：

1. super()和this()不能共存。
2. super();调用了父类中的构造方法，虚拟机会创建父类对象，但不是new出来的，而是虚拟机在执行字节码时在初始化方法 init（该方法在字节码中）中创建的对象！
3. 在Java语言中只要是创建Java对象，那么Object中的无参数构造方法一定会执行。

匿名内部类有构造器吗？

1. 匿名内部类是唯一一种没有构造器的类。
2. 一般使用匿名内部类的方法来编写事件监听代码。
3. 匿名内部类不能有访问修饰符和static修饰符。
4. 匿名内部类用于继承其他类或是实现接口，并不需要增加额外的方法，只是对继承方法的实现或是重写。

介绍一下 Error 和 Exception ?

Error 和 Exception 都共同继承于 Throwable 类。

异常和错误的区别:异常能被程序本身可以处理，错误是无法处理。

Throwable 类常用方法?

1. `public string getMessage()` : 返回异常发生时的详细信息
2. `public string toString()` : 返回异常发生时的简要描述
3. `public string getLocalizedMessage()` : 返回异常对象的本地化信息。使用 Throwable的子类覆盖这个方法，可以声称本地化信息。如果子类没有覆盖该方法，则该方法返回的信息与getMessage()返回的结果相同

4. `public void printStackTrace()`：在控制台上打印Throwable对象封装的异常信息

Error

是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM 出现的问题。例如，Java 虚拟机运行错误(Virtual MachineError)，当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java 虚拟机(JVM)一般会选择线程终止。

这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如 Java 虚拟机运行错误(Virtual MachineError)、类定义错误(NoClassDefFoundError)等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在 Java 中，错误通过 Error 的子类描述。

Exception

是程序本身可以处理的异常。Exception 类有一个重要的子类 RuntimeException。RuntimeException 异常由 Java 虚拟机抛出。

在异常处理的时候应该注意：

1. **不要在finally代码块中处理返回值**：按照我们的惯性认知：当遇到 return 语句的时候，执行函数会立刻返回。但是，在 Java 语言中，如果存在 finally 就会有例外。除了 return 语句，try 代码块中的 break 或 continue 语句也可能使控制权进入 finally 代码块。尽量不要在 try 代码块中调用 return、break 或 continue 语句，否则就要确保 finally 不会改变函数的返回值。
2. **当一个try后跟多个catch需要先捕获小异常在捕获大异常**
3. finally 中的代码始终是执行的，目的是为了清理资源（除非线程或者程序被中断）。

说出几种常见的错误和运行时异常？

错误

1. **StackOverflowError**：当前栈深度超过最大 JVM 栈深度，常常由无限递归造成。
2. **OutOfMemoryError**：没有足够的内存进行分配，一是改 JVM 内存大小（JVM 使用 `-XX:PermSize` 设置非堆内存初始值，默认是物理内存的 1/64；用 `-XX:MaxPermSize` 设置最大非堆内存的大小，默认是物理内存的 1/4）。
3. **VirtualMachineError**：JVM 运行时错误。

4. **NoClassDefFoundError**: 类定义错误。发生在类加载的**初始阶段**, 通常是如类依赖的 class文件或者jar不存在、类文件存在, 但存在不同的域中或者大小写问题 (javac编译时是无视大小写的, 很有可能编译出来的class文件与想要的不一样)。

运行时异常

1. **NullPointerException**: 空指针异常。
2. **ClassNotFoundException**: 找不到类异常。找不到指定的class, 发生在**加载阶段**。
常见的场景: 调用class的forName方法, 找不到指定的类、ClassLoader 中 findSystemClass, 找不到指定的类、ClassLoader中loadClass, 找不到指定的类等。
3. **ArithmaticException**: 算术异常。如除0。
4. **ArrayIndexOutOfBoundsException**: 数组下标越界异常。
5. **RuntimeException**: 运行时异常。由JVM抛出。
6. **InterruptedException**: 当线程处于阻塞状态时, 调用了 interrupted() 方法。
7. **IllegalMonitorStateException**: 在同步语块以外调用 wait()、notify() 和 notifyAll() 方法引起。

说一下final finally finalize有什么不同?

final

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。此外, 对于一个类中所有的private方法, 都会被隐式地指定为final。

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)。

finally

finally块用于try{...}catch{...}语块后, 无论是否捕获或处理异常, finally块里的语句都会被执行。当在try块或catch块中遇到return语句 时, finally语句块将在方法返回之前被执行。
finally中的代码始终是执行的, 目的是为了清理资源 (除非线程或者程序被中断) 。

finalize

finalize用于在GC回收内存前，给予即将被回收的对象一次“复活”的机会。对于GC可达性分析法中被认定为不可达的对象，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：

1. 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。
2. 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。

String 底层实现的 char[] 为什么要用 final 修饰？

Java 将 String 设成不可变最大的原因是效率和安全：

1. 不可变的 String 可以提高 String Pool(字符串常量池)的效率和安全性。在拷贝的 String 对象的内容时，用于它时不可变的，因此不用复制它本身，而只需要复制它的地址，而复制地址(通常一个指针的大小)只需要很小的内存，且效率也很好。对于其他引用同一个对象的其他变量也不会造成影响。
2. 可不变的 String 对于多线程是安全的，因为在多线程并发进行的情况下，一个可变对象的值很可能被其他线程改变这样会造成不可预期的结果，使用不可变对象就可以避免这种情况出现。

说一下 String 添加字符实现机制？

由于 String 是不可变类型，因此对 String 添加字符实际上是创建一个新的字符对象，并将当前变量引用指向新创建的对象上。其底层过程如下：

```
String a = "abd";
a = a + "d";
```

对于 Java 8 有：

1. 首先调用 String.valueOf(a) 获取a的值，并返回一个String类型对象。
2. 创建一个StringBuilder对象 StringBuilder sb = new
 StringBuilder(String.valueOf(a))
3. 将要拼接的字符通过调用StringBuilder的 append() 方法与原字符串拼接。

4. 调用StringBuilder的 `toString()` 方法返回结果。

因此，会出现如下现象：

```
String a = null;  
a = a + "d";  
System.out.println(a);  
  
// Output: null
```

值得注意的是，通过比较使用 `+` 和转成 `StringBuilder` 对象再调用其 `append()` 方法，可以发现前者的字节码行数多于后者，并且随着拼接字符串操作的数量的增加而急剧增长。同时，通过 `+` 拼接也会导致产生大量无引用的对象，会引发 GC。因此为了执行效率考虑，字符串拼接的时候，建议使用 `StringBuilder` 进行拼接。

Java 9 开始使用了 `invokeDynamic` 指令，可以动态指定要调用的方法，而不是一开始就编译好的。JDK9以拼接会创建新的对象，JDK9之后拼接静态字符串则不会。

介绍一下Java 9 中 String 的改动？

Java 8 中数组底层是用 `char[]` 来存储的。而在 Java 9 中，则是改由 `byte[] + coder` 来存储的。

但 Java 中 `char` 是16位UTF16 编码的，那么 `byte[]` 是如何存储 `char[]` 的呢？

```
public String(char value[]) {  
    this(value, 0, value.length, null);  
}  
*/  
String(char[] value, int off, int len, Void sig) {  
    if (len == 0) {  
        this.value = "".value;  
        this.coder = "".coder;  
        return;  
    }  
    if (COMPACT_STRINGS) {  
        byte[] val = StringUTF16.compress(value, off, len);  
        if (val != null) {  
            this.value = val;  
            this.coder = LATIN1;  
            return;  
        }  
    }  
    this.coder = UTF16;  
    this.value = StringUTF16.toBytes(value, off, len);  
}
```

默认初始化为true

这里是通过 `StringUTF16.compress(value, off, len)` 来判断，如果 `char[]` 存在 `value > 0xFF` 的值时，就返回null。

```
public static byte[] compress(char[] val, int off, int len) {
    byte[] ret = new byte[len];
    if (compress(val, off, ret, 0, len) == len) {
        return ret;
    }
    return null;
}

public static byte[] compress(byte[] val, int off, int len) {...}

// compressedCopy char[] -> byte[]
@HotSpotIntrinsicCandidate
public static int compress(char[] src, int srcOff, byte[] dst, int dstOff, int len) {
    for (int i = 0; i < len; i++) {
        char c = src[srcOff];
        if (c > 0xFF) {判断char的value是否大于0xFF
            len = 0;
            break;
        }
        dst[dstOff] = (byte)c;
        srcOff++;
        dstOff++;
    }
    return len;
}
```

如果 char[] 所有的字符都是小于 0xFF，那么正好让一个 byte 对应一个 char，String 构造到此结束。

只要存在一个 char > 0xFF，那么将会把 byte 数组的长度改为两倍 char 数组的长度，用两个字节存放一个 char

```

@HotSpotIntrinsicCandidate
public static byte[] toBytes(char[] value, int off, int len) {
    byte[] val = newBytesFor(len);
    for (int i = 0; i < len; i++) {
        putChar(val, i, value[off]);
        off++;
    }
    return val;
}

public static byte[] newBytesFor(int len) {
    if (len < 0) {
        throw new NegativeArraySizeException();
    }
    if (len > MAX_LENGTH) {
        throw new OutOfMemoryError("UTF16 String size is " + len +
                                   ", should be less than " + MAX_LENGTH);
    }
    return new byte[len << 1];
}

@HotSpotIntrinsicCandidate
// intrinsic performs no bounds checks
static void putChar(byte[] val, int index, int c) {
    assert index >= 0 && index < length(val) : "Trusted caller missed bounds check";
    index <<= 1;           ← 下标 X2
    val[index++] = (byte)(c >> HI_BYTE_SHIFT);   ← 高位
    val[index]   = (byte)(c >> LO_BYTE_SHIFT);   ← 低位
}

```

由于实现机制的变动，所有的String方法都重新实现了一遍，但对外的接口还是保持一致的。重构带来的最大好处就是在字符串中所有的字符都小于 0xFF 的情况下，会节省一半的内存。

StringBuilder 与 StringBuffer 的区别

两者都是用于字符串拼接的。但 StringBuilder 是线程不安全的， StringBuffer 是线程安全的。但由此也导致 StringBuilder 的效率要高于 StringBuffer。

== 与 equals() 的区别？

==

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：

1. 类没有重写 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。
2. 类重写了 equals() 方法。一般，我们都重写 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)。

为什么 equals() 和 hashCode() 要一起重写？

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. 如果 equals() 方法不重写，则它与 == 的效果相同，即判断两个对象是不是同一个对象，因此需要重写该方法。因此 equals() 方法被重写过，则 hashCode() 方法也必须被重写
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？

因为 hashCode() 所使用的杂凑算法可能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）。

如同 HashMap，如果 HashMap 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，而不是唯一标记一个对象。

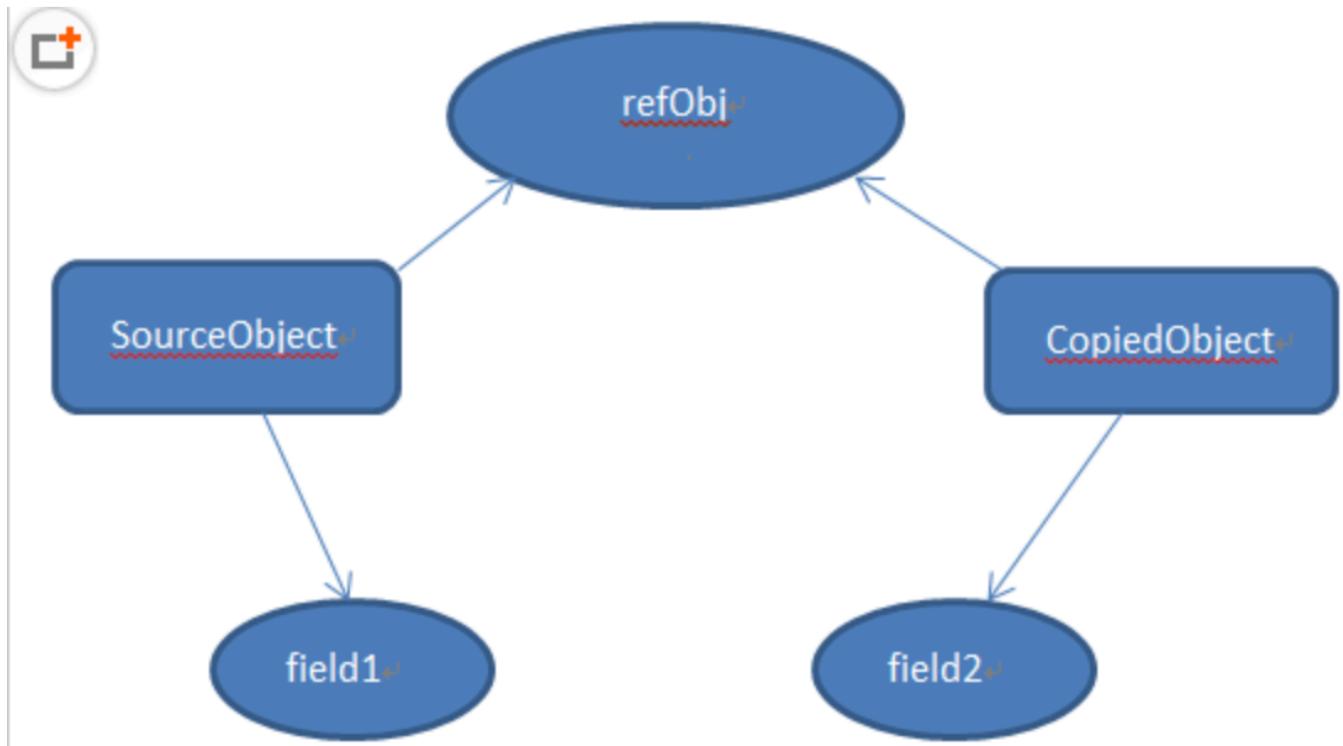
什么是 Java 浅拷贝与深拷贝？

浅拷贝

1. 对于数据类型是基本数据类型的成员变量，浅拷贝会直接进行值传递，也就是将该属性值复制一份给新的对象。因为是两份不同的数据，所以对其中一个对象的该成员变量值进行修改，不会影响另一个对象拷贝得到的数据。
2. 对于数据类型是引用数据类型的成员变量，比如说成员变量是某个数组、某个类的对象等，那么浅拷贝会进行引用传递，也就是只是将该成员变量的引用值（内存地址）复制

一份给新的对象。因为实际上两个对象的该成员变量都指向同一个实例。在这种情况下，在一个对象中修改该成员变量会影响到另一个对象的该成员变量值。

具体模型如图所示：可以看到基本数据类型的成员变量，对其值创建了新的拷贝。而引用数据类型的成员变量的实例仍然是只有一份，两个对象的该成员变量都指向同一个实例。



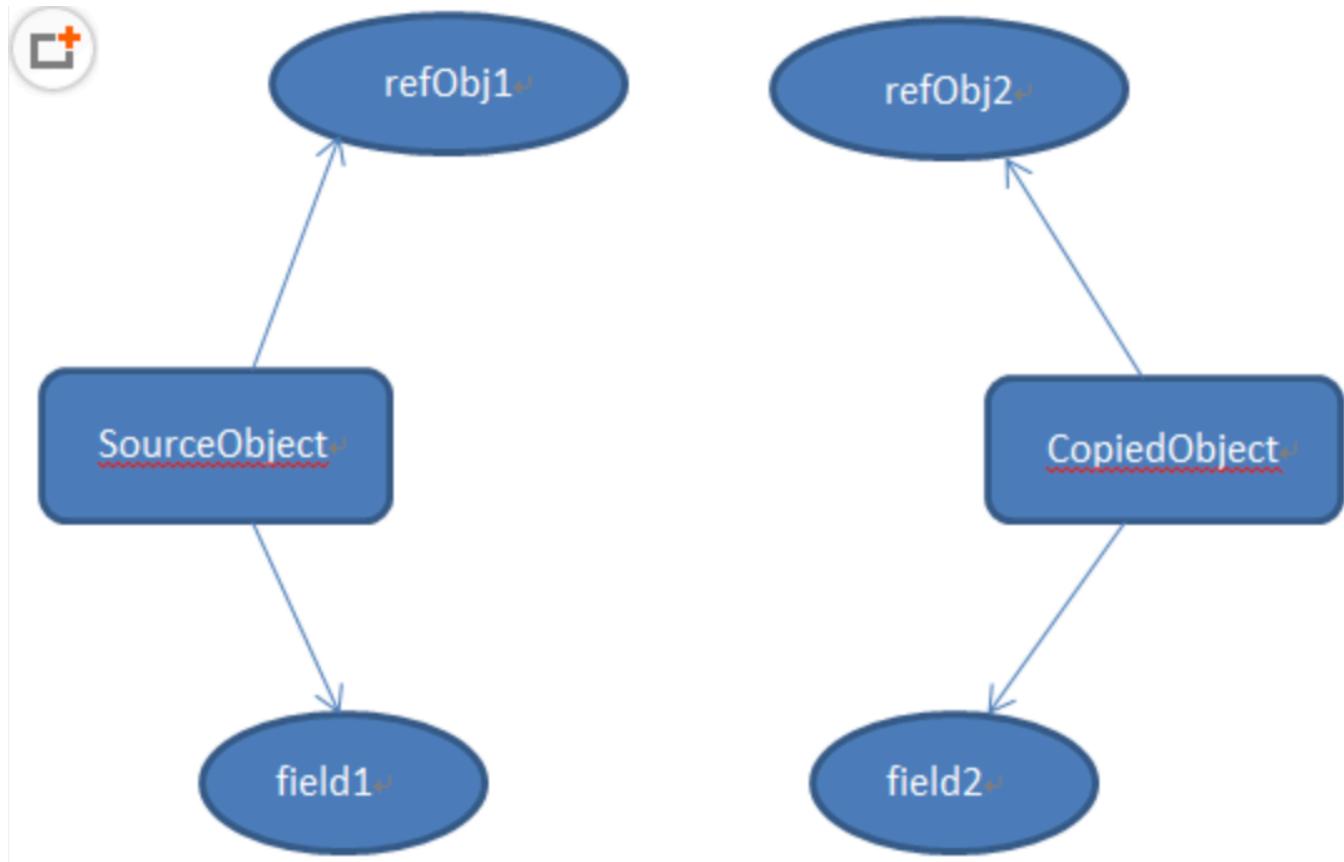
浅拷贝的主要实现方式：

1. **通过拷贝构造方法实现浅拷贝**：拷贝构造方法指的是该类的构造方法参数为该类的对象。使用拷贝构造方法可以很好地完成浅拷贝，直接通过一个现有的对象创建出与该对象属性相同的新的对象。
2. **通过重写 `clone()` 方法进行浅拷贝**：Object 类是类结构的根类，其中有一个方法为
`protected Object clone() throws CloneNotSupportedException`，这个方法就是进行的浅拷贝。有了这个浅拷贝模板，我们可以通过调用 `clone()` 方法来实现对象的浅拷贝。但是需要注意：1、Object 类虽然有这个方法，但是这个方法是受保护的（被 `protected` 修饰），所以我们无法直接使用。2、使用 `clone()` 方法的类必须实现 `Cloneable` 接口，否则会抛出异常 `CloneNotSupportedException`。对于这两点，我们的解决方法是，在要使用 `clone()` 方法的类中重写 `clone()` 方法，通过 `super.clone()` 调用 `Object` 类中的原 `clone()` 方法。

深拷贝

对于深拷贝来说，不仅要复制对象的所有基本数据类型的成员变量值，还要为所有引用数据类型的成员变量申请存储空间，并复制每个引用数据类型成员变量所引用的对象，直到该对

象可达的所有对象。也就是说，对象进行深拷贝要对整个对象图进行拷贝！简单地说，深拷贝对引用数据类型的成员变量的对象图中所有的对象都开辟了内存空间；而浅拷贝只是传递地址指向，新的对象并没有对引用数据类型创建内存空间。深拷贝模型如图所示：可以看到所有的成员变量都进行了复制。



深拷贝的主要实现方式：

1. **通过重写clone方法来实现深拷贝**：与通过重写 `clone()` 方法实现浅拷贝的基本思路一样，只需要为对象图的每一层的每一个对象都实现 `Cloneable` 接口并重写 `clone()` 方法，最后在最顶层的类的重写的 `clone()` 方法中调用所有的 `clone()` 方法即可实现深拷贝。简单的说就是：**每一层的每个对象都进行浅拷贝=深拷贝**。
2. **通过对象序列化实现深拷贝**：虽然层次调用 `clone()` 方法可以实现深拷贝，但是显然代码量实在太大。特别对于属性数量比较多、层次比较深的类而言，每个类都要重写 `clone()` 方法太过繁琐。将对象序列化为字节序列后，默认会将该对象的整个对象图进行序列化，再通过反序列即可完美地实现深拷贝。

介绍一下内存泄露？

如果忘记“释放”先前分配的内存，就可能造成内存泄漏。如果程序保留对永远不再使用的对象的引用，这些对象将会占用并耗尽内存，这是因为自动化的垃圾收集器无法证明这些对象将不再使用。如果存在一个对对象的引用，对象就被定义为活动的，因此不能删除。概括地说，内存泄漏产生的主要原因：**保留下来自原不再使用的对象引用**。

为了确保能回收对象占用的内存，编程人员必须确保该对象不能到达。这通常是通过将对象字段设置为 null 或者从集合(collection)中移除对象而完成的。但是，注意，当局部变量不再使用时，没有必要将其显式地设置为 null。对这些变量的引用将随着方法的退出而自动清除。

内存泄露例子？

1. **全局集合**: 在大的应用程序中有某种全局的数据储存库是很常见的，例如一个JNDI树或一个会话表。在这些情况下，必须注意管理储存库的大小。必须有某种机制从储存库中移除不再需要的数据。这可能有多种方法，但是最常见的一种是周期性运行的某种清除任务。该任务将验证储存库中的数据，并移除任何不再需要的数据。另一种管理储存库的方法是使用反向链接referrer计数。然后集合负责统计集合中每个入口的反向链接的数目。这要求反向链接告诉集合何时会退出入口。当反向链接数目为零时，该元素就可以从集合中移除了。
2. **缓存**: 没有正确地选用合适的缓存过期淘汰算法。最好的办法是对缓存的大小进行限制。
3. **HashMap 元素没有重写 equals() 或者 hashCode() 方法导致插入大量重复元素**。
4. **ThreadLocal 在线程池中没有及时清除**: 在不使用 ThreadLocal 后，要手动设为 null 并在 `finally{...}` 语句中调用 `remove()` 方法清除。
5. **数据库连接没有及时关闭**。

内存泄露检测工具？

1. **详细输出**: 有许多监控垃圾收集器活动的方法。而其中使用最广泛的可能是使用 `-Xverbose:gc` 选项启动 JVM，并观察输出，观察垃圾收集所使用的堆的容量。
2. **JRockit Management Console 控制台**: JRockit Management Console 可以显示堆使用量的图示。借助于该图，可以很容易地看出堆使用量是否随时间增加。
3. **JRockit Memory Leak Detector**: JRockit Memory Leak Detector 可以用来查看内存泄漏，并可以更深入地查出泄漏的根源。这个强大的工具是紧密集成到 JRockit JVM 中的，其开销非常小，对虚拟机的堆的访问也很容易。JRockit Memory Leak Detector 是通过在每次垃圾收集时计算每个类的现有对象的数目来实现这一步的。如果特定类的对象数目随时间而增长（“增长率”），就可能发生了内存泄漏。

列举 Object 类方法？

```
public final native Class<?> getClass()
```

//native方法，用于返回当前运行时对象的Class对象，使用了 final 关键字修饰，故不允许子类重写。

```
public native int hashCode()  
//native方法，用于返回对象的哈希码，主要使用在哈希表中，比如 JDK 中的 HashMap。  
  
public boolean equals(Object obj)  
//用于比较2个对象的内存地址是否相等，String 类对该方法进行了重写用户 比较字符串的值是否相等。  
  
protected native Object clone() throws CloneNotSupportedException  
// native 方法，用于创建并返回 当前对象的一份拷贝。一般情况下，对于任何对象 x，表达式 x.clone() != x 为true，x.clone().getClass() == x.getClass() 为 true 。 Object 本身没有实现 Cloneable 接口，所以不重写 clone 方法并且进行调用的话会发生 CloneNotSupportedException 异常。  
  
public String toString()  
// 返回类的名字@实例的哈希码的16进制的字符串。建议 Object 所有的子类都重写这个方法。  
  
public final native void notify()  
// native方法，并且不能重写。唤醒一个在此对象监视器上等待的线程(监视 器相当于就是锁的概念)。如果有多个线程在等待只会任意唤醒一个。  
  
public final native void notifyAll()  
// native方法，并且不能重写。跟notify一样，唯一的区别就是会唤醒 在此对象监视器上等待的所有线程，而不是一个线程。  
  
public final native void wait(long timeout) throws InterruptedException  
// native 方法，并且不能重写。暂停线程的执行。注意：sleep 方法没有释放锁，而 wait() 方法释放了锁。timeout 是等待时间。  
  
public final void wait(long timeout, int nanos) throws InterruptedException  
//多了 nanos 参数， 这个参数表示额外时间(以毫微秒为单位，范围是 0-999999)。 所以超时的时间还需要加上 nanos 毫秒。  
  
public final void wait() throws InterruptedException  
//跟之前的2个 wait 方法一样，只不过该方法一直等待，没有超时时间这个概念。  
  
protected void finalize() throws Throwable { }  
//实例被垃圾回收器回收的时候触发的操作
```

接口和抽象类的区别是什么？

1. 接口的方法默认是 public，所有方法在接口中不能有实现（Java 8 开始接口方法可以用 default 关键字修饰进行默认实现），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。

备注：在 Java 8 中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，必须重写，不然会报错。

什么是反射？

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方法和属性；这种动态获取信息以及动态调用对象方法的功能称为 Java 语言的反射机制。

反射的用途

在日常的第三方应用开发过程中，经常会遇到某个类的某个成员变量、方法或是属性是私有的或是只对系统应用开放，这时候就可以利用 Java 的反射机制通过反射来获取所需的私有成员或是方法。当然，也不是所有的都适合反射，之前就遇到一个案例，通过反射得到的结果与预期不符。阅读源码发现，经过层层调用后在最终返回结果的地方对应用的权限进行了校验，对于没有权限的应用返回值是没有意义的缺省值，否则返回实际值起到保护用户的隐私目的。

反射相关的类

类	用途
Class	代表类的实体，在运行的Java应用程序中表示类和接口。Class 对象是在加载类时由 Java 虚拟机自动构造的。也就是说我们不需要创建，JVM已经帮我们创建了。
Field	代表类的成员变量（成员变量也称为类的属性）
Method	代表类的方法
Constructor	代表类的构造方法

数据库驱动为什么使用反射调用不直接 new

JDBC 只是 JDK 提出的一种 Java 连接数据库的规范，它提供了一些接口和抽象方法，并没有提供到某个具体数据库的实现，而是由各个数据库厂家来实现 JDBC，比如上面提到的

mysql.Driver 就是一种具体实现。

因为反射是运行时根据全类名动态生成的 Class 对象，完全可以把这个全类名写在 xml 或者 properties 中去，不仅从代码上解耦和，而且需要更换数据库时，不需要进行代码的重新编译，只需要在配置文件中，更改相应的驱动和 URL 即可。

JDBC 连接查询过程

数据库连接过程：

```
Class.forName("com.mysql.jdbc.Driver");
String url="jdbc:mysql://localhost/test";
String user='root';
String password='root';
Connection conn = DriverManager.getConnection(url,user,password);
```

1、`Class.forName()` 加载 Driver 类，Driver 通过静态代码块把自己注册到 `DriverManager` 中，这也是下一步 `Connection conn = DriverManager.getConnection(url, username, password)` 能够获取连接的原因。而 `registerDriver()` 的作用，就是如果 `DriverManager` 中没有这个 Driver 类，就将这个类添加到到 `DriverManager` 的 list 中。

JDBC 的 DriverManager 是一个工厂类，我们通过它来创建数据库连接。当 JDBC 的 Driver 类被加载进来时，它会自己注册到 `DriverManager` 类里面。然后我们会把数据库配置信息传成 `DriverManager.getConnection()` 方法，`DriverManager` 会使用注册到它里面的驱动来获取数据库连接，并返回给调用的程序。

在实际的使用过程中，我们完全可以用 `java.sql.DriverManager.registerDriver(new Driver())` 这一句代码替换以反射加载驱动的方式，但这样会对具体的类产生依赖，导致后续更改不便。

```
static {
    try {
        // 放入到一个CopyOnWriteArrayList中
        java.sql.DriverManager.registerDriver(new Driver());
    } catch (SQLException E) {
        throw new RuntimeException("Can't register driver!");
    }
}

/**
 * Construct a new driver and register it with DriverManager
 *
```

```
* @throws SQLException
* if a database error occurs.
*/
public Driver() throws SQLException {
    // Required for Class.forName().newInstance()
}
```

在加载完成后，一般还要调用 Class 下的 `newInstance()` 静态方法来实例化对象以便操作。因为仅使用 `Class.forName()` 是动态加载类是没有用的，其最终目的是为了实例化对象。

这里有必要提一下就是 Class 下的 `newInstance()` 和 `new` 有什么区别？，首先，`newInstance()` 是一个方法，而 `new` 是一个关键字，其次，Class 下的 `newInstance()` 的使用有局限，因为它生成对象只能调用无参的构造函数，而使用 `new` 关键字生成对象没有这个限制。

那为什么在加载数据库驱动包的时候有的却没有调用 `newInstance()` 方法呢？即有的 JDBC 连接数据库的写法里是 `Class.forName(xxx.xx.xx)`；而有一些：`Class.forName(xxx.xx.xx).newInstance()`？刚才提到，`Class.forName()` 的作用是要求 JVM 查找并加载（查找 -> 加载 -> 连接（验证 -> 准备 -> 解析） -> 初始化）指定的类，如果在类中有静态初始化代码的话，JVM 必然会执行该类的静态代码段。而在 JDBC 规范中明确要求这个 Driver 类必须向 DriverManager 注册自己，即任何一个 JDBC Driver 的 Driver 类的代码都必须类似如下：

```
public class MyJDBCDriver implements Driver {
    static {
        DriverManager.registerDriver(new MyJDBCDriver());
    }
}
```

既然在静态初始化器的中已经进行了注册，所以我们在使用 JDBC 时只需要 `Class.forName(XXX.XXX)` 加载就可以了。

JDBC 4.0以后新增了新特性：JDBC 4.0不再需要显式调用 `Class.forName()` 注册驱动，`DriverManager` 初始化中会通过 `ServiceLoader` 类，在我们 `classpath` 中 `jar`（数据库驱动包）中查找，使用 `META-INF\services\java.sql.Driver` 文本中的类名称去注册。也就是说，在启动的时候，通过 `jar` 包下面的 `java.sql.Driver` 里的文本内容，帮你把驱动给加载了。

2、通过 `DriverManager`，使用 `URL`，用户名和密码建立连接(`Connection`)。本质上是调用了 `mysql.Driver` 的 `connect`方法，通过建立到数据库的socket连接，来完成接下来 SQL 的

执行。

```
for(DriverInfo aDriver : registeredDrivers) {  
    // If the caller does not have permission to load the driver then skip it.  
    if(isDriverAllowed(aDriver.driver, callerCL)) {  
        try {  
            System.out.println("trying " + aDriver.driver.getClass().getName());  
            // 通过具体注册的driver的connect方法获取连接  
            Connection con = aDriver.driver.connect(url, info);  
            if (con != null) {  
                // Success!  
                System.out.println("getConnection returning " + aDriver.driver.getClass().getName());  
                return (con);  
            }  
        } catch (SQLException ex) {  
            if (reason == null) {  
                reason = ex;  
            }  
        }  
    } else {  
        System.out.println("skipping: " + aDriver.getClass().getName());  
    }  
}
```

- 3、通过 Connection，使用 SQL 语句打开 Statement 对象。
- 4、执行语句，将结果返回 resultSet。
- 5、对结果 resultSet 进行处理。
- 6、倒序释放资源 resultSet -> preparedStatement -> connection。

JNI底层实现

JNI 是 Java Native Interface 的缩写，通过使用 Java 本地接口书写程序，可以确保代码在不同的平台上方便移植。JNI 一开始是为了本地已编译语言，尤其是 C 和 C++ 而设计的，但是它并不妨碍你使用其他编程语言，只要调用约定受支持就可以了。使用 Java 与本地已编译的代码交互，通常会丧失平台可移植性。但是，有些情况下这样做是可以接受的，甚至是必须的。例如，使用一些旧的库，与硬件、操作系统进行交互，或者为了提高程序的性能。JNI 标准至少要保证本地代码能工作在任何 Java 虚拟机环境。

本地代码与 Java 虚拟机之间是通过 JNI 函数实现相互操作的。JNI 函数通过接口指针来获得，本地方法将 JNI 接口指针当作参数来接受。虚拟机保证在从相同的 Java 线程中对本地

方法进行多次调用时，传递给本地方法的接口指针是相同的，本地方法被不同的 Java 线程调用时，它接受不同的 JNI 接口指针。

创建 native 方法步骤：

1、编写java程序，声明native方法：如果你想将一个方法做为一个本地方法的话，那么你就必须声明该方法为 `native` 的，并且不能实现。其中方法的参数和返回值在后面讲述。

Load动态库： `System.loadLibrary("hello");` 加载动态库（我们可以这样理解：我们的方法 `displayHelloWorld()` 没有实现，但是我们在下面就直接使用了，所以必须在使用之前对它进行初始化）这里一般是以 static 块进行加载的。同时需要注意的是 `System.loadLibrary();` 的参数“hello”是动态库的名字。

```
public class HelloWorld {  
    public native void displayHelloWorld(); //所有native关键词修饰的都是对本地的声明  
    static {  
        System.loadLibrary("hello"); //载入本地库  
    }  
    public static void main(String[] args) {  
        new HelloWorld().displayHelloWorld();  
    }  
}
```

2、编译：`javac`。

3、生成扩展名为h的头文件：新版 jdk-10.0.2 已经移除了 `javah` 命令工具，使用 `javac HelloWorld.java -h JniH` 代替 `javah HelloWorld` 命令生成扩展名为 h 的头文件。

4、编写本地方法实现和由 `javah` 命令生成的头文件里面声明的方法名相同的方法。

5、生成动态库。

6、运行程序。

泛型

(...)

协程

协程，又称微线程，纤程。协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。注意，在一个子程序中中断，去执行其他子程序，不是函数调用，有点类似 CPU 的中断。协程的特点在于是一个线程执行。

那么，协程的优势在哪？

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

因为协程是一个线程执行，那怎么利用多核CPU呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

Java 8 的新特性？

1. Lambda表达式和函数式接口
2. 接口的默认方法 `default` 和静态方法
3. 方法引用
4. 重复注解 `@Repeatable`
5. 更好的类型推断：Java 8编译器在类型推断方面有很大的提升，在很多场景下编译器可以推导出某个参数的数据类型，从而使得代码更为简洁。
6. 拓宽注解的应用场景：Java 8拓宽了注解的应用场景。现在，注解几乎可以使用在任何元素上：局部变量、接口类型、超类和接口实现类，甚至可以用在函数的异常定义上。
7. 引入 Optional：解决`NullPointerException`，从而避免源码被各种null检查污染，以便开发者写出更加整洁的代码。Optional仅仅是一个容易：存放T类型的值或者null。它提供了一些有用的接口来避免显式的null检查。
 - i. 新增的 Stream API (`java.util.stream`) 将生成环境的函数式编程引入了Java库中。这是目前为止最大的一次对Java库的完善，以便开发者能够写出更加有效、更加简洁和紧凑的代码。
 - ii. Java 8引入了新的Date-Time API(JSR 310)来改进时间、日期的处理。
8. 还有其他更多特性.....

Java 9 的新特性？

1. HTTP 2 客户端：HTTP/2标准是HTTP协议的最新版本，新的 `HttpClient API` 支持 `WebSocket` 和 `HTTP2` 流以及服务器推送特性。
2. 多版本兼容 JAR 包：多版本兼容 JAR 功能能让你创建仅在特定版本的 Java 环境中运行库程序时选择使用的 class 版本。
3. 私有接口方法：在接口中使用`private`私有方法。我们可以使用 `private` 访问修饰符在接口中编写私有方法。
4. 集合工厂方法：`List`, `Set` 和 `Map` 接口中，新的静态工厂方法可以创建这些集合的不可变实例。

5. 模块系统：模块是一个包的容器，Java 9 最大的变化之一是引入了模块系统（Jigsaw 项目）。
6. 还有其他更多特性.....

最新的 Java 版本是多少？

截至2019年9月，最新版本为 Java 12

Java 12 的新特性？

1. Switch 表达式
2. G1的可中断 mixed GC
3. G1归还不使用的内存
4. 还有其他更多特性.....

Java集合类

讲讲Java集合框架，把你知道的都讲出来

Collections用过什么

Collections.sort()底层用的什么排序方法（归并排序）

Comparator和Comparable

List遍历方式的选择

String的底层hashcode怎么写的

treemap底层数据结构，扩容，插入删除效率？

介绍一下HashMap？

结构上

在JDK 1.8之前底层是数组+链表形式实现的。HashMap的key的hashCode经过扰动函数处理后得到hash值，然后通过 $(n-1) \& hash$ (n 指数组长度) 判断当前元素的存放的下标位置。如果当前位置上已经存放了元素了，则先判断已经存放的元素的hash值与key值是否与将要插入的元素的相等，如果不相等，则通过拉链法（用链表）解决冲突；如果一样则直接覆盖。

从JDK 1.8开始，HashMap在解决hash冲突时发生了变化。当链表长度大于阈值（默认是8）时，将会自动将链表转换为红黑树进行存储。

hash()函数实现上

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。

JDK 1.8之前：

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

JDK 1.8开始：

```
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);
}
```

HashMap链表转红黑树是怎么实现的

(...)

HashMap内存溢出处理？

HashMap内存溢出，说明插入了过多的Entry。HashMap中判断key是否相同是根据 `e.hash == hash && ((k = e.key) == key || key.equals(k))` 判断的。引发内存溢出问题的原

因一般是插入Entry的Key类的equals()和hashCode()方法没有重写，导致插入了大量重复的元素，进而导致内存泄露、内存溢出问题。解决方法就是重写equals()和hashCode()方法。

HashMap的key可以是基本类型吗？如果一定要定义为基本类型怎么做？

不可以，因为HashMap中判断key是否相同是根据 `e.hash == hash && ((k = e.key) == key || key.equals(k))` 判断的，在计算hash值的时候需要调用hashCode()方法，而基本类型不存在此方法，因此不可以使用基本类型。

如果一定要使用基本类型作为key，那就需要重写HashMap。

HashMap的长度为什么2的幂次？

因为虽然Hash值的范围值-2147483648到2147483647，前后加起来大概40亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个40亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的，用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。因此Entry在插入HashMap之前需要通过`(n-1) & hash`计算下标。

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的扩容，HashMap并发操作下会有什么问题

HashMap怎么使用才是线程安全的

HashMap的链表为什么将长度为8是转化为红黑色的临界点？

手写 hashmap 的 get 方法 / set 方法？

ConcurrentHashMap介绍？

ConcurrentHashMap怎么实现线程安全的？

List集合中有哪些线程安全的类？

Java并发

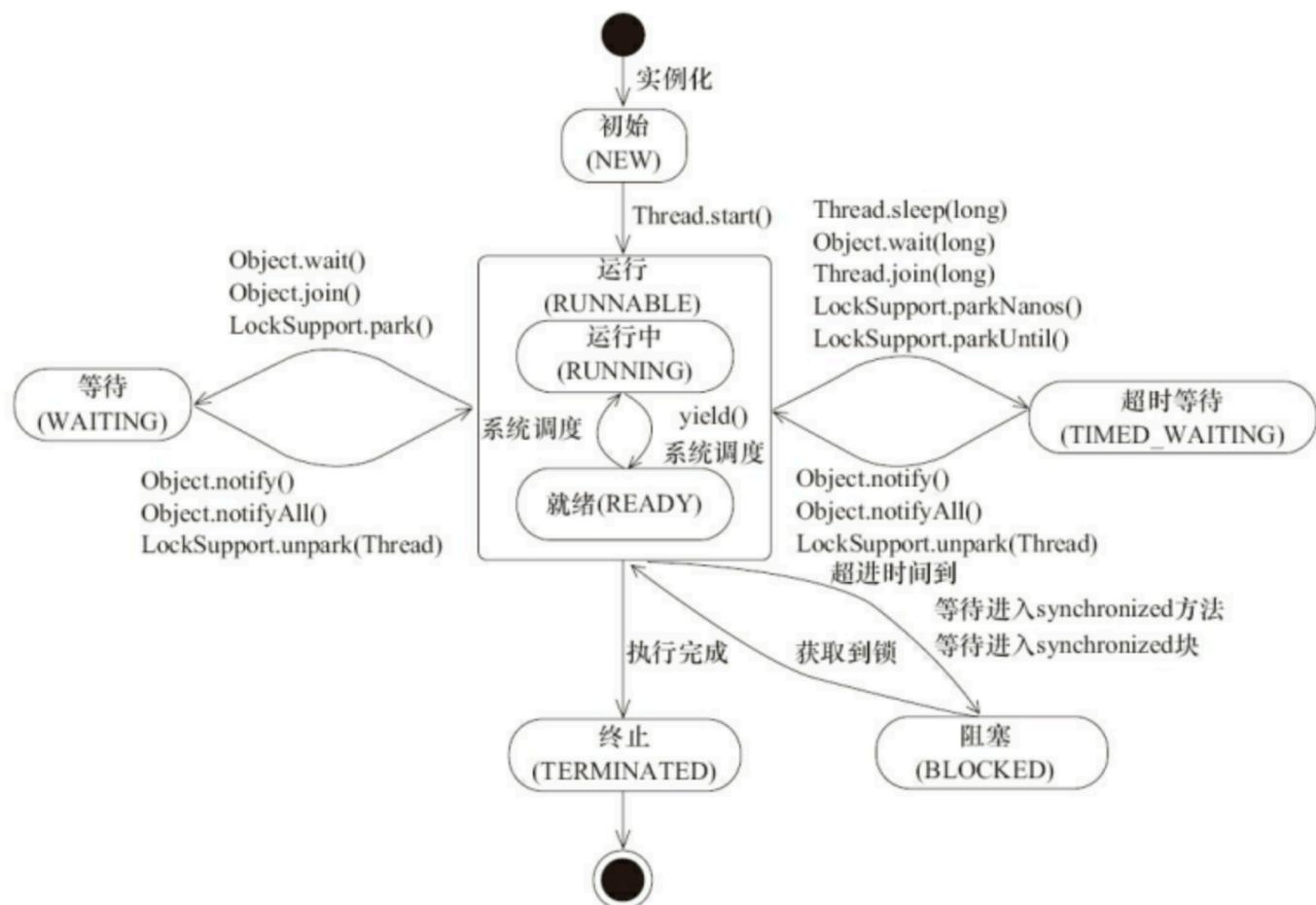
多线程三大特性？

原子性: 即一个或多个操作要么全部不做，要么全部都做。

可见性: 即多线程访问一个变量时，当一个线程改变了该变量值后，其他线程能立刻获得最新值。

有序性: 即结果顺序符合程序代码逻辑执行顺序。一般来说处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

线程的生命周期？（手画！！！）



阻塞状态和等待状态区别?

阻塞状态：当一个线程试图获取一个内部的对象锁（不是 JUC 库中的 Lock），而该锁又被其他线程所持有，则该线程进入阻塞状态。通常会在 synchronized 同步语块中出现这种情况。

等待状态：当线程等待另一线程通知调度器一个条件时，它自己进入等待状态。通常在调用 Object.wait() 和 Thread.join() 方法时，或者 JUC 库中的 Lock 或 Condition 时，就会出现这种情况。

什么是上下文切换?

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

说说并发与并行的区别?

并发：指宏观上多个线程是同时在工作的，实际上可能是多个线程真的在同时工作；也可能是多个线程快速切换运行，在同一时间只有一个线程在工作，但只是因为切换较快，感觉上就是多个线程在同时工作。即同一时间段，多个任务都在执行（单位时间内不一定同时执行）。

并行：指在微观上同时有多个线程在同时工作，这就要求必须有至少两个 CPU 在运行线程。即单位时间内，多个任务同时执行。

为什么要使用多线程呢?

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

使用多线程可能带来什么问题？

脏读、死锁、内存泄漏、OOM

单CPU的情况下适合用多线程吗？

单核开多线程只是为了编程方便，属于类似语法糖的范畴。它在特定领域可以提供较大的编程便利。

JDK 中消费者生产者应用场景？

Java 实现的经典的方法是使用 `wait()` 和 `notify()` 方法来协调生产者消费者的同步合作，实现生产者消费者模式最方便的方法是使用 JUC 中的阻塞队列（`BlockingQueue`），因为它结构更简单，更容易编程控制。只需要编写业务代码，而同步的问题，扔给了阻塞队列来管理。

当队列满的时候生产者调用的 `put()` 方法将阻塞，同理当队列为空的时候消费者 `take()` 方法将阻塞，等待生产者生产。

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Main {
    public static void main(String[] args){
        BlockingQueue<String> blockingQueue = new LinkedBlockingDeque<>();
        Thread producerThread = new Thread(new Producer(blockingQueue));
        Thread consumerThread = new Thread(new Consumer(blockingQueue));
        producerThread.start();
        consumerThread.start();
    }
}

class Producer implements Runnable{

    private BlockingQueue<String> bufQueue;

    public Producer(BlockingQueue<String> bufQueue){
        this.bufQueue = bufQueue;
    }
}
```

```

@Override
public void run(){
    try{
        for(int i = 0; i < 10; i++){
            bufQueue.put(String.valueOf(i));
            System.out.println("Producer: " + i);
        }
        bufQueue.put("OVER");
    }
    catch (Exception e){
        Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, e);
    }
}

class Consumer implements Runnable{

    private BlockingQueue<String> bufQueue;

    public Consumer(BlockingQueue<String> bufQueue){
        this.bufQueue = bufQueue;
    }

    @Override
    public void run(){
        try{
            String curr = null;
            while(!(curr = bufQueue.take()).equals("OVER")){
                System.out.println("Consumer: " + curr);
            }
        }
        catch (Exception e){
            Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, e);
        }
    }
}

```

但现实中生产者和消费者的数量通常不是1:1的，因此，可以通过使用AtomicInteger对生产者和消费者数量进行记录。

```

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingDeque;

```

```
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Main {
    public static void main(String[] args){
        BlockingQueue<String> blockingQueue = new LinkedBlockingDeque<>();
        final int PRODUCT_TARGET_NUM = 50;
        final int PROCUER_NUM = 2;
        final int CONSUMER_NUM = 10;
        AtomicInteger producerCount = new AtomicInteger(PROCUER_NUM);
        AtomicInteger productCount = new AtomicInteger(0);
        AtomicBoolean soldOut = new AtomicBoolean(false);

        for(int i = 0; i < PROCUER_NUM; i++){
            Thread producerThread = new Thread(new Producer(blockingQueue, producerCount, productCount, PRODUCT_TARGET_NUM, i, soldOut));
            producerThread.start();
        }
        for(int i = 0; i < CONSUMER_NUM; i++){
            Thread consumerThread = new Thread(new Consumer(blockingQueue, producerCount, i, soldOut));
            consumerThread.start();
        }
    }
}
```

```
class Producer implements Runnable{

    private BlockingQueue<String> bufQueue;
    private AtomicInteger producerCount;
    private AtomicInteger productCount;
    private AtomicBoolean soldOut;
    private int targetNum;
    private int num;

    public Producer(BlockingQueue<String> bufQueue, AtomicInteger producerCount, AtomicInteger productCount, int targetNum, int num, AtomicBoolean soldOut){
        this.bufQueue = bufQueue;
        this.producerCount = producerCount;
        this.productCount = productCount;
        this.targetNum = targetNum;
        this.num = num;
        this.soldOut = soldOut;
    }
}
```

```

@Override
public void run(){
    if(!soldOut.get()){
        try{
            int i = productCount.getAndIncrement();
            while (i < targetNum){
                bufQueue.put(String.valueOf(i));
                System.out.println("Producer" + num + ":" + i);
                i = productCount.getAndIncrement();
            }
            soldOut.set(true);
        }
        catch (Exception e){
            Logger.getLogger(Producer.class.getName()).log(Level.SEVERE,
null, e);
        }
        finally {
            producerCount.getAndDecrement();
        }
    }
}

```

```

class Consumer implements Runnable{

    private BlockingQueue<String> bufQueue;
    private AtomicInteger producerCount;
    private AtomicBoolean soldOut;
    private int num;

    public Consumer(BlockingQueue<String> bufQueue, AtomicInteger producerCo
unt, int num, AtomicBoolean soldOut){
        this.bufQueue = bufQueue;
        this.producerCount = producerCount;
        this.num = num;
        this.soldOut = soldOut;
    }

    @Override
    public void run(){
        try{
            while(!soldOut.get() || !bufQueue.isEmpty()){
                if(!bufQueue.isEmpty()){
                    System.out.println("Consumer" + num + ":" + bufQueue.ta
ke());
                }
            }
        }
    }
}

```

```

        else {
            try{
                System.out.println("Consumer" + num + ": WAITING 50ms");
                Thread.sleep(50);
            }
            catch (Exception e){
                Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, e);
            }
        }
    }
    catch (Exception e){
        Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, e);
    }
}
}
}
=====
```

OUTPUT:

```

Producer0: 0
Producer1: 1
Producer0: 2
Producer1: 3
Producer0: 4
Producer1: 5
Producer0: 6
Producer1: 7
Producer1: 9
Consumer0: 0
Producer0: 8
Consumer0: 2
Consumer2: 3
Producer1: 10
Consumer1: 1
Consumer4: 7
Producer1: 12
Consumer3: 6
Consumer2: 5
Consumer0: 4
Producer0: 11
Consumer7: WAITING 50ms
Consumer0: WAITING 50ms
Consumer6: 13
```

```
Consumer2: 12
Consumer5: 11
Consumer3: 10
Consumer3: WAITING 50ms
Producer1: 13
Producer1: 15
Consumer4: 9
Consumer4: 15
Consumer1: 8
Consumer1: WAITING 50ms
Consumer4: 16
Producer1: 16
Producer1: 17
Producer1: 18
Consumer5: WAITING 50ms
Consumer2: WAITING 50ms
Consumer9: WAITING 50ms
Consumer6: WAITING 50ms
Consumer8: 14
Consumer8: 17
Consumer8: 18
Consumer8: 19
Producer0: 14
Consumer8: WAITING 50ms
Producer1: 19
Producer1: 21
Consumer4: WAITING 50ms
Producer1: 22
Producer0: 20
Producer1: 23
Producer0: 24
Producer1: 25
Producer0: 26
Producer1: 27
Producer0: 28
Producer0: 30
Producer1: 29
Producer1: 32
Producer1: 33
Producer0: 31
Producer1: 34
Producer0: 35
Producer1: 36
Producer0: 37
Producer1: 38
Producer0: 39
Producer1: 40
Producer1: 42
```

```
Producer1: 43
Producer0: 41
Producer0: 45
Producer1: 44
Producer0: 46
Producer1: 47
Producer0: 48
Producer1: 49
Consumer3: 21
Consumer7: 20
Consumer7: 23
Consumer3: 22
Consumer7: 24
Consumer3: 25
Consumer7: 26
Consumer3: 27
Consumer5: 28
Consumer3: 29
Consumer5: 30
Consumer2: 34
Consumer6: 35
Consumer0: 33
Consumer8: 41
Consumer3: 32
Consumer1: 31
Consumer3: 44
Consumer1: 45
Consumer8: 43
Consumer0: 42
Consumer7: 40
Consumer9: 38
Consumer6: 39
Consumer5: 37
Consumer2: 36
Consumer8: 49
Consumer1: 48
Consumer3: 47
Consumer4: 46
```

如何保证线程执行顺序？

1. 使用 `join()` 方法等待当前线程执行完毕再执行下一个线程。
2. 使用最大工作线程为1的线程池来执行线程。

两个线程如何顺序输出0-100？

```
public class Main{

    static int num = 0;
    static int max = 100;
    static volatile boolean isOdd = false;

    public static void main(String[] args){

        Thread t1 = new Thread(() -> {
            while(num < max){
                if(!isOdd && (num == 0 || ++num % 2 == 0)){
                    System.out.println(Thread.currentThread().getName() + " " + num);
                    isOdd = true;
                }
            }
        }, "Even");

        Thread t2 = new Thread(() -> {
            while(num < max){
                if(isOdd && ++num % 2 != 0){
                    System.out.println(Thread.currentThread().getName() + " " + num);
                    isOdd = false;
                }
            }
        }, " Odd");

        t1.start();
        t2.start();
    }
}
```

Java 实现了线程的调度吗？

实现了，Java 是抢占式调度，线程调度具有一定的随机性，可以通过 `setPriority()` 方法对线程设置优先级进行调度，但优先级并不是很靠谱，因为Java线程是通过映射到系统的原生线程上来实现的，所以线程调度最终还是取决于操作系统。

Java 里为什么有线程安全问题？如何解决？

因为 JVM 内存中的堆、方法区两个区域是多线程主要的数据共享区域，有数据共享就意味

着有可能出现脏读和死锁等线程安全问题。可以通过对线程加锁或使用 synchronized 关键字以及避免共享变量进行解决。

多线程的实现方式？

1. 继承 Thread 类 实例化之后调用 start() 方法
2. 实现 Runnable 接口 实例化一个 Thread 传入该类实例，重写并运行 run()
3. 实现 Callable 接口，重写并调用 call()，通过 FutureTask 包装器来创建线程
4. 使用 ExecutorService、Callable、Future 实现有返回结果的线程

Thread、Runnable 和 Callable 区别？哪一种比较好，哪一种比较安全？

1. Thread 是类，需要继承，而 Java 只允许单继承，因此通过继承 Thread 类来创建线程的方式降低了类的可拓展性。
2. Runnable 是接口，Java 允许实现多个接口，因此使用 Runnable 可以保证程序的可拓展性。同时，它增加了程序的健壮性，代码可以被多个线程共享，代码和数据独立，线程安全，适合于多个有相同代码的线程区处理同一个资源。它没有返回值。
3. Callable 是接口，与 Runnable 相比，Callable 有一个 call() 方法有返回值。同时，call() 方法可抛出异常，而 run() 方法是不能抛出异常的。运行 Callable 任务可拿到一个 Future 对象，Future 表示异步计算的结果。通过 Future 对象可了解任务执行情况，可取消任务的执行，还可获取任务执行的结果。

线程在 JVM 中 start() 一定会启动吗？

不一定，start() 方法只是使得新建的线程进入就绪状态（READY），只有当当前线程被系统分配了时间片后，才会通过系统调用进入运行状态（RUNNING）。

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？

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。

wait() 与 sleep() 与 yield() 的区别？

wait(): 是 Object 类中的方法，他可以选传一个毫秒时间参数。它不是随便可以调用的，必须包含在对应的 synchronized 同步语块中。在它工作时，首先必须获得目标对象的一个监视器（锁），而 wait() 方法执行后，会释放这个监视器（锁）。这样做的目的是使得其他等待同一个对象的线程不会因为当前线程的休眠而全部无法执行。

sleep(): 是 Thread 类中的方法，它需要传入一个毫秒参数，指定线程“睡眠”时间。它可以在任何地方调用，调用后会使得当前工作线程立刻进入 TIMED-WAITING 状态，并且不会释放当前线程所持有的锁。

yield(): 是 Thread 类中的方法，用来暂停当前执行的线程并使之回到就绪（READY）状态，此时所有线程（包括当前线程）将会根据优先级（如果设了）来进行调用。因此有可能当前线程刚进入就绪（READY）状态，又立刻被调用，回到运行（RUNNING）状态。

notifyAll()怎么个全部法，全部唤醒都开始执行不就不安全了吗？

调用 notifyAll() 方法能够唤醒所有正在等待这个对象的 monitor （监视器，即锁）的线程。

如何安全地停止线程

1、**使用共享变量**：之所以引入共享变量，是因为该变量可以被多个执行相同任务的线程用来作为是否中断的信号，通知中断线程的执行。

```
private volatile Thread blinker;

public void stop() {
    blinker = null;
}

public void run() {
    Thread thisThread = Thread.currentThread();
    while (blinker == thisThread) {
        try {
            thisThread.sleep(interval);
        } catch (InterruptedException e){
        }
    }
}
```

```
        repaint();
    }
}
```

2、使用 **Thread.interrupted()** 方法：如果一个线程由于等待某些事件的发生而被阻塞，又该怎样停止该线程呢？这种情况经常会发生，比如当一个线程由于需要等候键盘输入而被阻塞，或者调用 `Thread.join()` 方法，或者 `Thread.sleep()` 方法，在网络中调用 `ServerSocket.accept()` 方法，或者调用了 `DatagramSocket.receive()` 方法时，都有可能导致线程阻塞，使线程处于不可运行状态时，即使主程序中将该线程的共享变量设置为 `true`，但该线程此时根本无法检查循环标志，当然也就无法立即中断。我们可以使用 **Thread.interrupt()** 方法，因为该方法虽然不会中断一个正在运行的线程，但是它可以使一个被阻塞的线程抛出一个中断异常 `InterruptedException`，从而使线程提前结束阻塞状态，退出堵塞代码。

interrupt() 和 isInterrupted() 的区别？

interrupt()：是一个静态方法，它用于检测**当前线程**是否被中断。

isInterrupted()：是一个实例方法，可以用来检测是否有线程被中断，调用这个方法不会改变中断状态。

为什么stop()不安全？

`stop()` 方法在结束线程时，会直接终止线程，并立刻释放这个线程所持有的锁，可能线程数据写到一半的时候被强行终止，导致数据不一致，对象被写坏

怎么让一个线程等另一个线程执行结束？

1. **Thread 的 join()**: `thread.join()` 让当前线程等待线程 `thread` 结束后再继续执行。
`join()`让线程存活的原理是：不断检查调用 `join()` 的线程是否存活，如果调用 `join()` 的线程存活，则让当前线程永远等待，其中，`wait(0)` 表示永远等待下去。直到调用 `join()` 的线程结束，JVM 会调用 `this.notifyAll()`（这段代码在 JVM 源码里，因此 JDK 里面看不见）使得当前线程恢复运行。
2. **JUC 的 CountDownLatch 类**: `CountDownLatch` 构造函数接收一个 int 参数 N，表示计数器。当调用 `CountDownLatch.countDown()` 时，计数减1，直至减到0为止。这里的 N 个点，可以是 N 个线程，也可以是 N 个执行步骤。
3. **JUC 的 CyclicBarrier 类**: 它的作用是让一组线程到达一个“屏障”（一个同步点）时被阻塞，直到本组最后一个线程到达“屏障”时，所有被拦截的线程才会继续运行。

说一下Java 线程间通信方式？ / Java 线程之间如何互操作？

1. **wait()、notify()、notifyAll()**：每次交互是都需要握手。同时，通常 wait() 操作还要用一个 while{} 循环包围，因为有可能多个任务出于相同原因在等待一个锁，而第一个被唤醒的任务会改变这种状况。因此，当有任务改变这种状况时，当前任务应该再次被挂起，直到其感兴趣的条件发生变化。
2. **Lock 和 Condition 对象**：Condition.await() 、Condition.signal() 和 Condition.signalAll() 方法。
3. **阻塞队列 / 同步队列（BlockingQueue）**：阻塞队列任何时候都只允许一个任务插入或者移除元素。当队列为空时，当任务试图从中获取元素对象时，那么这个队列可以把这个任务挂起，并且当用更多元素可用时恢复该任务。队列的自动挂起和恢复任务的机制，将显示调用 wait()、notify()、notifyAll() 时类和类之间的耦合消除，因为每个类只喝它的阻塞队列通信。
4. **任务间使用管道进行输入/输出**：PipedWriter、PipedReader 类（允许不同任务从同一个管道中读取）。实际上，管道基本上就是一个阻塞队列。在 shutdownNow() 被调用时，PipedReader 与普通的 I/O 之间最重要的差异就是前者可以被中断（interrupted()），而后者不可以。

JDK 同步机制是怎样？

1. **synchronized** 关键字。
2. **Lock** 接口及其实现类。
3. **信号量（Semaphore）**：是一种计数器，用来保护一个或者多个共享资源的访问，它是并发编程的一种基础工具，大多数编程语言都提供这个机制，这也是操作系统中经常提到的。
4. **CountDownLatch**：Java 语言提供的同步辅助类，在完成一组正在其他线程中执行的操作之前，他允许线程一直等待。
5. **CyclicBarrier**：Java 语言提供的同步辅助类，它允许多个线程在某一个集合点处进行相互等待。
6. **Phaser**：Java 语言提供的同步辅助类，它把并发任务分成多个阶段运行，在开始下一阶段之前，当前阶段中所有的线程都必须执行完成，JAVA7 才有的特性。
7. **Exchanger**：他提供了两个线程之间的数据交换点。

Java多线程会占满CPU吗？

一个线程死循环的情况下会占满一个CPU核

死锁的4个必要条件？

1. **互斥资源**: 即一个资源在任何一个时刻只能被一个线程所拥有。 (这个条件是无法被破坏的)
2. **请求与保持**: 即一个线程一旦获得一个资源后, 即使当前线程被阻塞, 也不会释放资源。
3. **不可剥夺**: 即一个线程一旦获得一个资源后, 其他线程不能从这个线程上抢夺这个资源。
4. **循环等待**: 即多个线程互相等待对方释放资源才能继续完成自己的任务。

如何避免死锁? 如何打破死锁的条件?

死锁预防是计算机操作系统, 在设计时确定资源分配算法, 为保证不发生死锁, 而破坏产生死锁的必要条件的行为过程。

1. **破坏请求与保持条件**: 一次性申请所有的资源。
2. **破坏不可剥夺条件**: 占用部分资源的线程进一步申请其他资源时, 如果申请不到, 可以主动释放它占有的资源。
3. **破坏循环等待条件**: 靠按序申请资源来预防。按某一顺序申请资源, 释放资源则反序释放。破坏循环等待条件。

写个死锁算法?

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Main {

    public static int[] resource = {0, 0};

    public static void main(String[] args){
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new Runner(resource, 1));
        exec.execute(new Runner(resource, 2));
        try{
            TimeUnit.SECONDS.sleep(1);
        } catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```

    }
    catch (InterruptedException e){
        e.printStackTrace();
    }
    exec.shutdownNow();
}
}

class Runner implements Runnable{

private int[] resourse;
private int id;

Runner(int[] resourse, int id){
    this.resourse = resourse;
    this.id = id;
}

@Override
public void run(){

try{
    while (!Thread.interrupted()){
        synchronized (resourse) {
            while (resourse[0] != 0 || resourse[1] != 0) {
                System.out.println("Thread " + id + " is waiting for
resources!");
                resourse.wait();
            }
            resourse[0] = id;
        }
        Thread.currentThread().sleep(10);
        synchronized (resourse){
            while (resourse[0] != 0 || resourse[1] != 0) {
                System.out.println("Thread " + id + " is waiting for
resources!");
                resourse.wait();
            }
            resourse[1] = id;
            System.out.println("Thread " + id + " gets all resources
!");
            resourse[0] = 0;
            resourse[1] = 0;
            System.out.println("Thread " + id + " releases all resou
rces!");
            resourse.notifyAll();
        }
    }
}
}

```

```
        Thread.currentThread().interrupt();
    }
}
catch (InterruptedException e){
    e.printStackTrace();
}
}
}
```

不用锁如何实现线程同步

基于 **Happens-Before (HB)** 规则，可以实现不使用 volatile 和锁实现共享变量的同步操作。

该规则定义了 Java 多线程操作的有序性和可见性，防止了编译器重排序对程序结果的影响。当一个变量被多个线程读取并且至少被一个线程写入时，如果读操作和写操作没有 HB 关系，则会产生数据竞争问题。要想保证操作 B 的线程看到操作 A 的结果（无论 A 和 B 是否在一个线程），那么在 A 和 B 之间必须满足 HB 原则，如果没有，将有可能导致重排序。

Happens-Before 原则是 JMM 的核心所在，只有满足了 HB 原则才能保证有序性和可见性，否则编译器将会对代码重排序。HB 甚至为 lock 和 volatile 也定义了规则。

HB 规则：

1. **程序次序规则**：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作。
2. **锁定规则**：在监视器锁上的解锁操作必须在同一个监视器上的加锁操作之前执行。
3. **volatile 变量规则**：对一个变量的写操作先行发生于后面对这个变量的读操作。
4. **传递规则**：如果操作 A 先行发生于操作 B，而操作 B 又先行发生于操作 C，则可以得出操作 A 先行发生于操作 C。
5. **线程启动规则**：Thread对象的start()方法先行发生于此线程的每一个动作。
6. **线程中断规则**：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生。
7. **线程终结规则**：线程中所有的操作都先行发生于线程的终止检测，我们可以通过 Thread.join()方法结束、Thread.isAlive() 的返回值手段检测到线程已经终止执行。
8. **对象终结规则**：一个对象的初始化完成先行发生于他的 finalize() 方法的开始。

实例：

一、 程序次序规则 + volatile 变量规则 + 传递规则

```

/* 两个线程交替打印 0-100 */

public class Main{

    static int num = 0;
    static int max = 100;
    static volatile boolean isOdd = false;

    public static void main(String[] args){

        Thread t1 = new Thread(() -> {
            while(num < max){
                if(!isOdd && (num == 0 || ++num % 2 == 0)){
                    System.out.println(Thread.currentThread().getName() + " : " + num);
                    isOdd = true;
                }
            }
        }, "Even");

        Thread t2 = new Thread(() -> {
            while(num < max){
                if(isOdd && ++num % 2 != 0){
                    System.out.println(Thread.currentThread().getName() + " : " + num);
                    isOdd = false;
                }
            }
        }, "Odd");

        t1.start();
        t2.start();
    }
}

```

二、线程终结规则

```

static int a = 1;

public static void main(String[] args) {
    Thread tb = new Thread(() -> {
        a = 2;
    });
    Thread ta = new Thread(() -> {
        try {

```

```
        tb.join();
    } catch (InterruptedException e) {
        //NO
    }
    System.out.println(a);
});

ta.start();
tb.start();
}
```

三、线程启动规则

```
static int a = 1;

public static void main(String[] args) {
    Thread tb = new Thread(() -> {
        System.out.println(a);
    });
    Thread ta = new Thread(() -> {
        a = 2;
        tb.start();
    });
    ta.start();
}
```

介绍一下线程池？

线程池的意义？

多线程的软件设计方法可以最大限度的发挥多核处理器的计算能力，提高CPU的使用率，提高生产系统的吞吐量和性能。但是，如果不对线程的数量加以控制，却会适得其反。

1. **降低资源消耗**: 创建过多的线程会占用大量内存资源，影响系统性能，甚至可能导致OOM问题。
2. **提高响应速度**: 过多的短期任务线程的回收会对GC带来巨大的压力，延长GC停顿时间，影响系统性能。
3. **提高线程的可管理性**: 创建大量的线程也会因线程间的切换带来巨大的开销，影响系统性能。

因此，为了最大的发挥多线程开发的优势，我们需要将线程的数量控制在一个合理的范围之中，并对已经创建的线程进行重复使用，以减少线程回收的开销，降低GC压力，而**线程池**

(**ThreadPool**) 的概念应运而生。

线程池构造方法的参数，具体介绍一下？

类型	参数	作用
int	corePoolSize	核心线程数量
int	maximumPoolSize	线程池最大线程数量
long	keepAliveTime	空闲线程的最大存活时间
TimeUnit	unit	时间单位
BlockingQueue	workQueue	任务队列，存放待执行的任务
ThreadFactory	threadFactory	创建线程的工厂
RejectedExecutionHandler	handler	拒绝策略

线程池类型？

线程池类型	
SingleThreadExecutor	适用于需要保证顺序地执行各个任务；并且在任意时
FixedThreadPool	适用于未来满足资源管理的需求，而需限制当前线程
CacheThreadPool	这是一个会根据需要创建新线程的、大小无界的线程
ScheduledThreadPoolExecutor	适用于需要多个后台线程执行周期任务，同时为里满
SingleThreadScheduledExecutor	适用于需要单个后台线程执行周期任务，同时需要保

线程池拒绝策略？

拒绝策略	说明
AbortPolicy	直接抛出异常
CallerRunsPolicy	由调用者所在线程来运行任务
DiscardOldestPolicy	丢弃任务队列里最近一个任务，并执行当前任务
DiscardPolicy	不处理当前任务，直接丢弃
	其他实现 <code>RejectedExecutionHandler</code> 接口的自定义拒绝策略

线程池实现原理?

当一个任务提交到线程池后，线程池流程如下：

1. 线程池判断核心线程池里的线程是否都在工作，如果不是，则创建一个新的工作线程执行任务；如果是，则进入下一步。
2. 线程池判断当前任务队列是否已经满了，如果没有满，则将任务加入任务队列中，等待被执行；如果满了，则进入下一步。
3. 线程池检查当前工作线程数量是否超过最大线程数量，如果没有超过，则创建一个新的工作线程来执行任务；如果满了，则进入下一步。
4. 线程池基于拒绝策略拒绝执行任务。

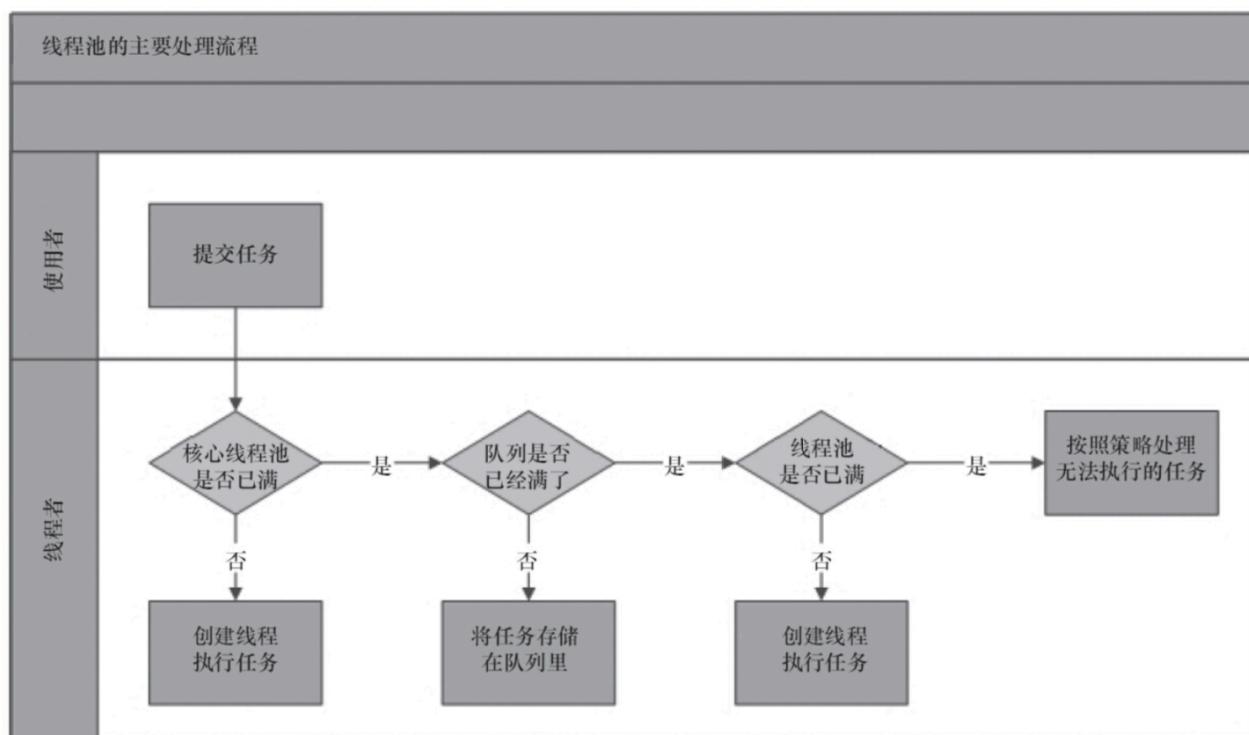


图9-1 线程池的主要处理流程

线程池里如何知道线程执行完了没有？

1. 使用线程池的原生函数 `isTerminated()`。
2. 使用 CountDownLatch。
3. 维持一个公共计数。

假设你的 CPU 是 2 核或者是 4 核的，如果采用固定大小的线程池，那你会固定多少个线程？

线程等待时间所占比例越高，需要越多线程。线程CPU时间所占比例越高，需要越少线程。

N 代表 CPU 的核数。

1. 如果是 IO 密集型应用，则线程池大小设置为 **2N+1**，IO密集型任务CPU使用率并不高，因此可以让CPU在等待IO的时候去处理别的任务，充分利用CPU时间。
2. 如果是 CPU 密集型应用，则线程池大小设置为 **N+1**，因为CPU密集型任务使得CPU使用率很高，若开过多的线程数，只能增加上下文切换的次数，因此会带来额外的开销。

线程池执行任务的方法？

Execute()

execute()直接将一个Runnable任务提交到线程池去运行，它无返回结果。它的内部执行流程遵循上述线程池实现原理。

```
public void execute(Runnable command) {
    int c = ctl.get();                                // 获取线程池控制
变量 ctl
    if (workerCountOf(c) < corePoolSize) {           // step 1: 核心
线程数判断
        if (addWorker(command, true))                // step 1.1 添加
核心线程执行任务
            return;
        c = ctl.get();                                // 重新获取最新的
控制变量 ctl (因为可能在上述过程中可能线程池控制变量发生改变)
    }
    if (isRunning(c) && workQueue.offer(command)) { // step 2 检查当
前线程池是否还在运行状态，并尝试任务加入队列
        int recheck = ctl.get();                      // 重新获取最新的
控制变量 ctl (因为可能在上述过程中可能线程池控制变量发生改变)
        if (!isRunning(recheck) && remove(command)) // step 2.1 判断
线程池是否运行，防止线程池状态的突变，如果突变，那么执行拒绝策略，reject线程，防止 workQu
ue 中增加新线程
            reject(command);
        else if (workerCountOf(recheck) == 0)          // step 2.2 判断
当前工作线程数量如果等于0，直接添加工作线程
            addWorker(null, false);
    }
    else if (!addWorker(command, false))              // step 3 如果任
务无法加入队列，尝试创建线程执行任务
        reject(command);
}
```

ctl

ctl 是线程池的最重要的控制变量，它该变量是一个 AtomicInteger 类型的原子变量，记录着线程池的线程总数量 **workerCount** 以及线程池的状态 **runState**。这个变量设计的非常巧妙，一方面减少了线程池的变量数量，更重要的一方面是，该变量是原子类型变量，线程池的实现函数中，往往需要同时获取这两个属性，如果将两个属性放入一个原子变量中，根据 Atomic 类支持线程的重入，线程池也就只需获取一把锁，便可以控制线程池的两个属性，这里实际上变相减少了一把锁的使用。

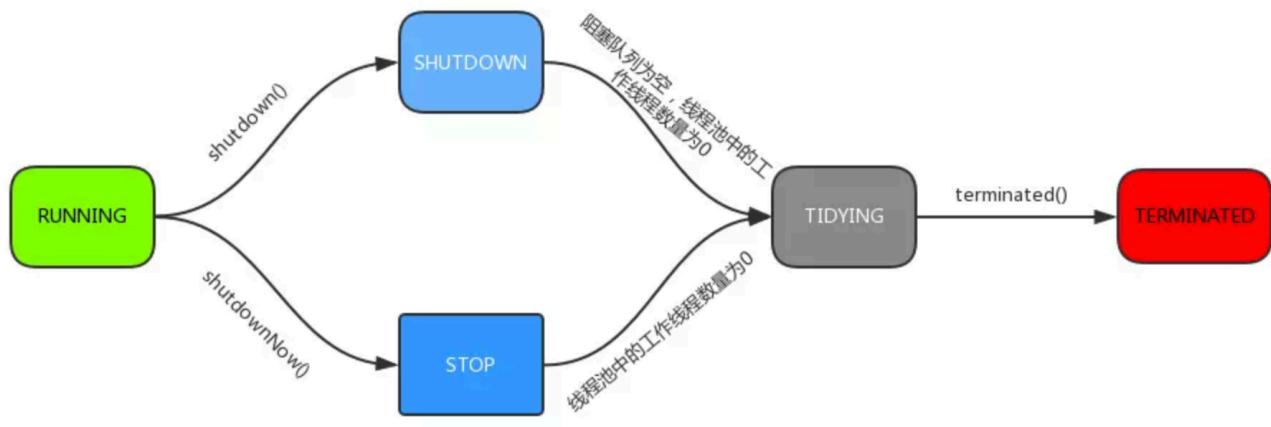
那么 ctl 是如何将两个属性合并为一个变量的呢？首先AtomicInteger类型变量为32位，workerCount 是占据着一个 atomic integer 的后29位的，而runState占据了前3位，所以，workerCount 上限是 $(2^{29}) - 1$ 。

3 bits	29 bits
runState	workerCount

runState 是整个线程池的运行生命周期，有如下取值：

取值 (COUNT-BITS == 3时)	runState	说明
111 -> -1	RUNNING	可以新加线程，同时可以处理任务队列 queue 中的任务
000 -> 0	SHUTDOWN	不增加新线程，但是处理任务队列 queue 中的任务
001 -> 1	STOP	不增加新线程，同时不处理任务队列 queue 中的任务
010 -> 2	TIDYING	所有的线程都终止了（任务队列 queue 中），同时 workerCount 为0，那么此时进入 TIDYING
011 -> 3	TERMINATED	terminated()方法结束，变为 TERMINATED

线程池状态转换图如下所示：



它的相关操作如下所示：

```

private static final int COUNT_BITS = Integer.SIZE - 3;           //workerCount
t 位数
private static final int CAPACITY   = (1 << COUNT_BITS) - 1;     //workerCount
t 上限是(2^29)-1

/*几个状态, 用 Integer 的高三位表示*/
private static final int RUNNING     = -1 << COUNT_BITS;          //111
private static final int SHUTDOWN    = 0 << COUNT_BITS;            //000
private static final int STOP        = 1 << COUNT_BITS;            //001
private static final int TIDYING     = 2 << COUNT_BITS;            //010
private static final int TERMINATED  = 3 << COUNT_BITS;            //011

/*获取后COUNT_BITS位数值, 即 workerCount */
private static int workerCountOf(int c) {
    return c & CAPACITY;
}

/*获取前COUNT_BITS位数值 runState */
private static int runStateOf(int c) {
    return c & ~CAPACITY;
}

/*获取 ctl */
private static int ctlOf(int rs, int wc) { return rs | wc; }
  
```

Submit()

用 `ThreadPoolExecutor.submit(Runnable task)` 方法来向线程池提交任务，该方法会返回一个 `Futtrue` 类型的结果，通过以下代码便可以判断任务是否执行成功了。

```

Future<Object> threadFuture = threadPoolExecutor.submit(task);

try{
    Object result = threadFuture.get();
} catch (InterruptedException e){
    // 处理线程中断异常
} catch (ExecutionException e){
    // 处理无法执行异常
} finally {
    threadPoolExecutor.shutdown();
}

```

scheduled() 与 **scheduledAtFixedRate()** 与 **scheduledWithFixedDelay** 区别？

方法	说明
scheduled()	到达指定时间后执行任务
scheduledAtFixedRate()	每隔一段时间 Rate 启动执行一个任务
scheduledWithFixedDelay	当一个任务完成后过 Delay 时间，再启动一个新任务

多线程 Future 接口，里面有什么方法

```

public interface Future<V> {
    // 尝试取消此次任务 mayInterruptIfRunning - true 如果执行该任务的线程应该被中断;
    // 否则，正在进行的任务被允许完成
    boolean cancel(boolean mayInterruptIfRunning);
    // 如果此任务在正常完成之前被取消，则返回 true。
    boolean isCancelled();
    // 返回true如果任务已完成。 完成可能是由于正常终止，异常或取消 - 在所有这些情况下，此
    // 方法将返回true。
    boolean isDone();
    // 等待计算完成然后返回结果
    V get() throws InterruptedException, ExecutionException;
    // 在指定的时间之内进行等待，超时不等待
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}

```

锁的四种状态？

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。

无锁

即没有任何锁的状态。

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！

在 Java 6 和 Java 7 里面，偏向锁时默认启动的。

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(JDK 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。

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！

重量级锁

即类似于 synchronized 关键字这种，依赖于依赖于底层的操作系统的 Mutex Lock 来实现。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。

自旋锁的底层实现？

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。

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。

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，在任何时刻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：自旋的时间不再固定了，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。

什么是锁消除？

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。

什么是锁粗化？

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反复加锁和解锁，那么会带来很多不必要的性能消耗。

乐观锁和悲观锁？

(先说两者概念，引出synchronized和CAS，谈下锁升级和详细CAS过程及Java中的应用)

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。

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。悲观锁适合于写多的情况。

CAS ?

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。

Java CAS 如何解决ABA问题？

使用版本号机制。

介绍一下 volatile ?

`volatile` 可以说是轻量级的 `synchronized`，它可以在不引发线程上下文切换的条件下保证共享变量的可见性。因此如果 `volatile` 使用恰当的话，成本比 `synchronized` 低。同时，它还可以防止 JVM 对指令的重排，保证执行顺序。但是，`volatile` 不保证操作的原子性。

那么它是如何保证可见性的呢？通过 JIT 查看汇编代码可以看出，`volatile` 变量在进行写操作的时候会多出一行 `Lock` 前缀指令，它的作用是：

1. 将当前处理器缓存行（这是 CPU cache 可以分配的最小存储单位）的数据写回到系统内存。因为 `Lock` 前缀指令会在执行的时候声言 `LOCK#` 信号，这个信号会锁住总线，导致其他 CPU 不能访问总线，从而确保声言期间当前 CPU 可以独占任何共享内存。
2. 这个写回内存的操作会使其他 CPU 里缓存了该内存地址的数据无效。这里用的是处理器的嗅探技术，在此不过多介绍。

说一说自己对于 `synchronized` 关键字的了解？

`synchronized` 关键字可以保证被修饰对象或操作的原子性和可见性，保证被修饰对象或操作任何时候只能有一个线程对其进行操作。

`synchronized` 是悲观锁。

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。JDK 1.6 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。

`synchronized` 既可修饰非静态方法，也可修饰静态方法，还可修饰代码块，有何区别？

当 `synchronized` 修饰静态方法或变量时，其实是获得被修饰方法或者变量所在的类的锁，即对类进行加锁。

当 `synchronized` 修饰非静态方法或者变量时，实际上是获得对应对象的锁。

尽量不要使用 `synchronized(String a)` 因为 JVM 中，字符串常量池具有缓存功能！

`synchronized` 底层实现？

当 synchronized 修饰同步语块时：

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

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

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

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) 的持有权。当计数器为0则可以成功获取，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。

当 synchronized 修饰方法时

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

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

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

synchronized 和锁与 volatile 区别？

1. synchronized 和锁需要通过操作系统来仲裁谁获得锁，开销比较高，而 volatile 开销小很多，因此 volatile 的效果更好。
2. volatile 本质是在告诉 JVM 当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
3. volatile 仅能使用在变量级别。synchronized 则可以使用在变量、方法、和类级别的。
4. volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。
5. volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。

6. `volatile` 标记的变量不会被编译器优化；`synchronized` 标记的变量可以被编译器优化。

介绍一下各种锁

(...)

Lock 和 synchronized 区别，哪种性能更好以及原因？

1. 首先 `synchronized` 是 Java 内置关键字，在 JVM 层面，`Lock` 是个 Java 类。
2. `synchronized` 无法判断是否获取锁的状态，`Lock` 可以判断是否获取到锁。
`(java.lang.Thread.holdsLock())`
3. `synchronized` 会自动释放锁（`a` 线程执行完同步代码会释放锁。`b` 线程执行过程中发生异常会释放锁），`Lock` 需在 `finally{...}` 中手工释放锁（`unlock()` 方法释放锁），否则容易造成线程死锁。
4. 用 `synchronized` 关键字的两个线程`1`和线程`2`，如果当前线程`1`获得锁，线程`2`线程等待。如果线程`1`阻塞，线程`2`则会一直等待下去。而 `Lock` 锁就不一定会等待下去，如果尝试获取不到锁，线程可以不用一直等待就结束了。
`(lock.lockInterruptibly())`
5. `synchronized` 的锁可重入、不可中断、非公平，而 `Lock` 锁可重入、可判断、可公平（两者皆可）
6. `Lock` 锁适合大量同步的代码的同步问题，`synchronized` 锁适合代码少量的同步问题。

ReentrantLock 与 synchronized 的异同？性能差异？

相同点：两者都是可重入锁。即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。

不同点：除了上述 `Lock` 和 `synchronized` 区别所列举的不同点外，`ReentrantLock` 比 `synchronized` 增加了一些高级功能：

1. `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。
2. `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。

在性能上：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。

锁和 synchronized 是否保证可见性？如何保证？

根据 JDK 中对 concurrent 包的说明，一个线程的写结果保证对另外线程的读操作可见，只要该写操作可以由 happen-before 原则推断出在读操作之前发生，因此锁和 synchronized 通过保证同一时间只有一个线程执行目标代码段来实现的。

锁和 synchronized 可以保证原子性，为什么又需要 AtomicInteger 来保证原子操作？

锁和 synchronized 需要通过操作系统来仲裁谁获得锁，开销比较高，而 AtomicInteger 是通过 CPU 级的 CAS 操作来保证原子性，开销比较小。

如何判断一个对象是否有锁？

`java.lang.Thread.holdsLock()`，它是一个由native修饰的非java代码实现的方法。

Unsafe 类详解一下？

封装了一些不安全的操作，例如类似指针的 CAS 操作：`compareAndSwapObject()` 等，一

一旦操作不慎，指针偏移量设置错误，就可能导致系统崩溃等灾难性后果。Unsafe 实例需要调用其工厂 getUnsafe() 来获得，但 JDK 开发人员不希望开发者使用这个类，因此 getUnsafe() 内部实现中会判断调用者的类的类加载器是否为空，不为空则说明是应用程序类的 App Loader，为空的则为系统核心类如 Bootstrap 类的类加载器。只用为空的情况即调用类是系统核心类才能获得 Unsafe 类的实例，因此一般开发者开发时不能直接调用 Unsafe 类。

CAS 中 Unsafe 调的哪个方法？

UNSAFE.compareAndSwapObject()

讲讲并发包（JUC）下面的类

atomic、CyclicBarrier、CountDownLatch、CopyOnWriteArrayList、
CopyOnWriteArraySet、ConcurrentSkipListSet、ConcurrentHashMap、
ConcurrentSkipListMap、ArrayBlockingQueue, LinkedBlockingQueue、
LinkedBlockingDeque、ConcurrentLinkedQueue、ConcurrentLinkedDeque ...

Atomic包？

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

CAS 的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。Unsafe 类的 objectFieldOffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffset。另外 value 是一个 volatile 变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

```
// setup to use Unsafe.compareAndSwapInt for updates (更新操作时提供“比较并替换”的作用)
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long valueOffset;

static {
    try {
        valueOffset = unsafe.objectFieldOffset
            (AtomicInteger.class.getDeclaredField("value"));
    } catch (Exception ex) { throw new Error(ex); }
}

private volatile int value;
```

AtomicInteger 常用方法

```
public final int get() //获取当前的值  
public final int getAndSet(int newValue)//获取当前的值，并设置新的值  
public final int getAndIncrement()//获取当前的值，并自增  
public final int getAndDecrement() //获取当前的值，并自减  
public final int getAndAdd(int delta) //获取当前的值，并加上预期的值  
boolean compareAndSet(int expect, int update) //如果输入的数值等于预期值，则  
以原子方式将该值设置为输入值 (update)  
public final void lazySet(int newValue)//最终设置为newValue，使用 lazySet 设  
置之后可能导致其他线程在之后的一小段
```

Threadlocal?

通常情况下，我们创建的变量是可以被任何一个线程访问并修改的。如果想实现每一个线程都有自己的专属本地变量该如何解决呢？JDK 中提供的 ThreadLocal 类正是为了解决这样的问题。ThreadLocal 类主要解决的就是让每个线程绑定自己的值，可以将 ThreadLocal 类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。

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

每个线程内部有一个 ThreadLocalMap（相当于 HashMap），然后在存储变量的时候，最终的变量是放在了当前线程的 ThreadLocalMap 中，并且以 ThreadLocal 作为 key 值，并不是存在 ThreadLocal 上，ThreadLocal 可以理解为只是 ThreadLocalMap 的封装，传递了变量值。

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

AQS?

RPC 调用？

多个线程对同一个资源调用，如何区分每个线程

JVM

静态变量可以序列化吗？

静态成员属于类级别的，所以不能序列化，序列化只是序列化了对象而已，这里“不能序列化”的意思是序列化信息中不包含这个静态成员域。

什么是 Java 虚拟机？

Java 虚拟机是一个可以执行Java字节码的虚拟机进程。Java 源文件被编译成能被 Java 虚拟机执行的字节码文件。

为什么 Java 被称作是“平台无关的编程语言”？

因为 JVM 不和包括 Java 在内的任何语言绑定，它只和“`.class` 文件”这种特殊的二进制文件绑定，`.class` 文件包含了 JVM 指令集和符号表以及若干其他辅助信息；JVM 屏蔽了底层硬件平台的指令长度和其他特性，Java 程序都运行在 JVM 上而无需考虑计算机底层硬件平台的指令长度及其他特性。

JVM 作用？

1. 屏蔽底层硬件平台的指令长度及其他特性，实现 Java 平台无关性，增加 Java 的可移植性。
2. 提供 GC 内存回收机制，使得开发者可以让他们摆脱繁琐的内存管理工作，让开发更有效率。
3. 提供 Java 类加载机制。

JVM 调优？

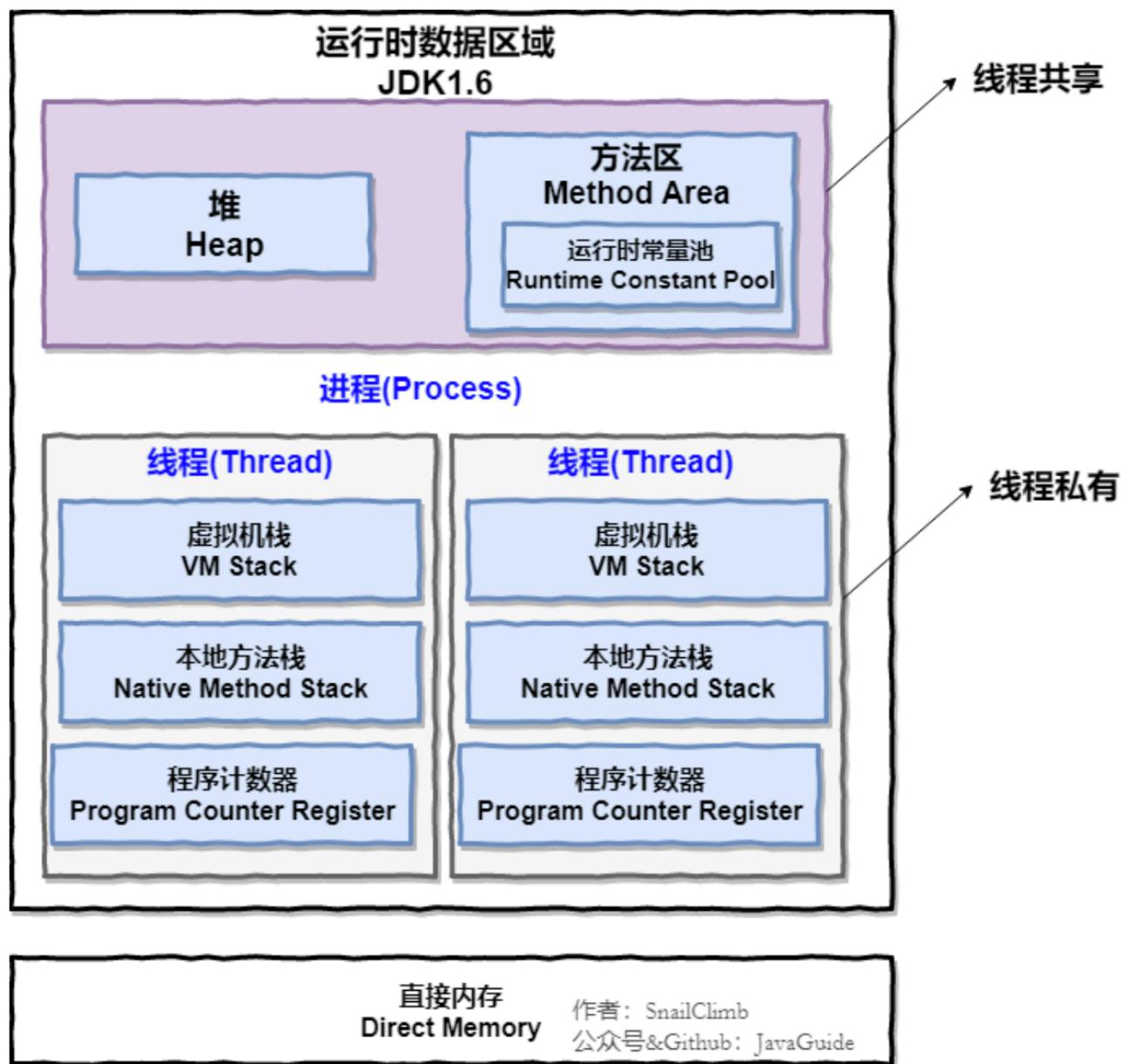
(详细到参数)

JVM 内存结构？

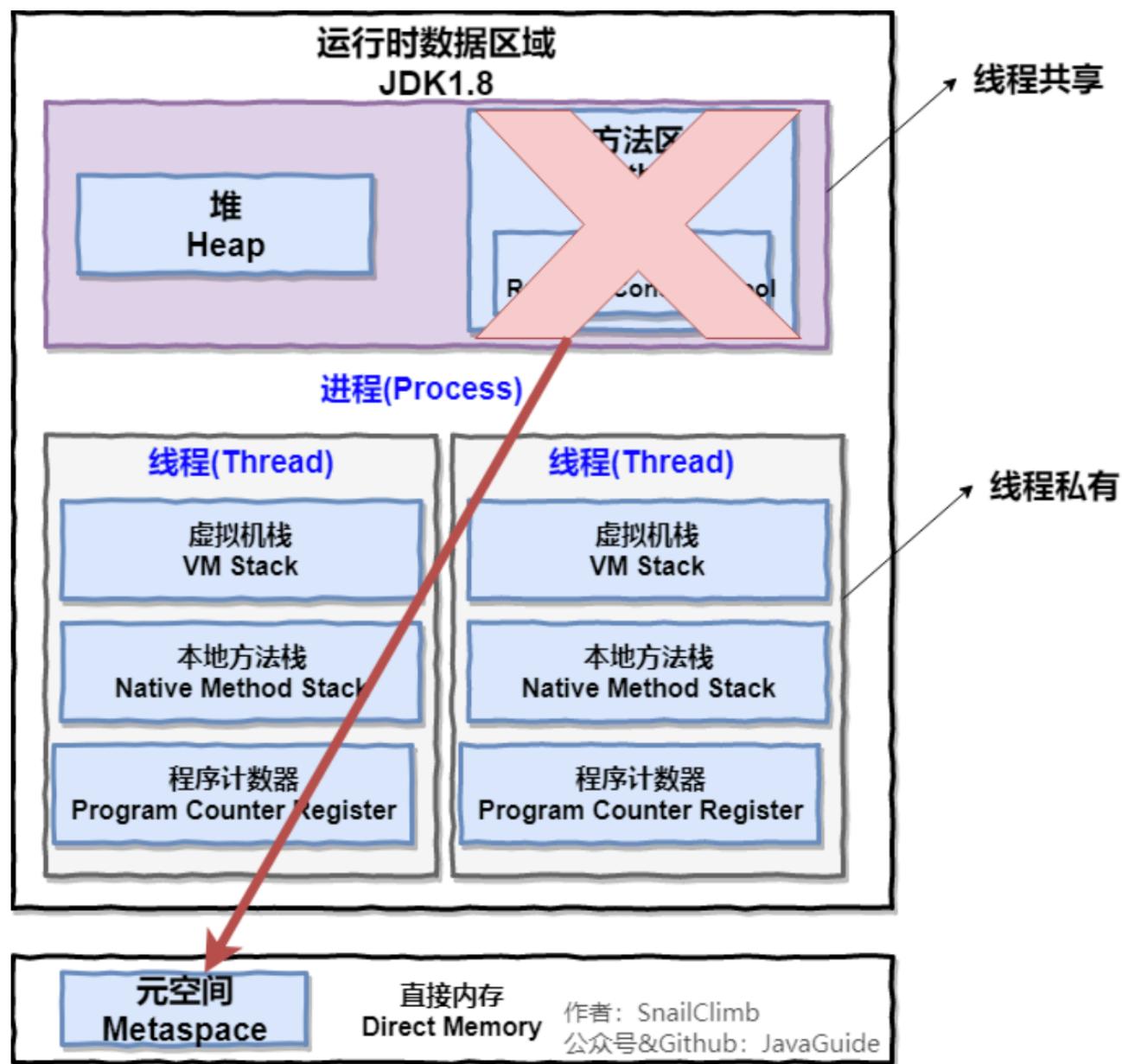
Java 虚拟机在执行 Java 程序的过程中会把它管理的内存划分成若干个不同的数据区域。

JDK 1.8 和之前的版本略有不同，下面会介绍到。

JDK 1.8之前：



JDK 1.8开始：



共享情况	区域
线程私有	程序计数器、虚拟机栈、本地方法栈
线程共享	堆、方法区、运行时常量池、直接内存、元空间

哪些是运行时数据区?

程序计数器、虚拟机栈、本地方法栈、堆、方法区、运行时常量池

程序计数器

用于记录当前线程执行的字节码的行号。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，我们称程序计数器为“线程私有”的内存。

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。

虚拟机栈

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

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。**Java** 栈可用类比数据结构中栈，**Java** 栈中保存的主要内容是栈帧，每一次函数调用都会有一个对应的栈帧被压入 **Java** 栈，每一个函数调用结束（`return` 语句或抛出异常）后，都会有一个栈帧被弹出。

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**。

1. **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。
2. **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。

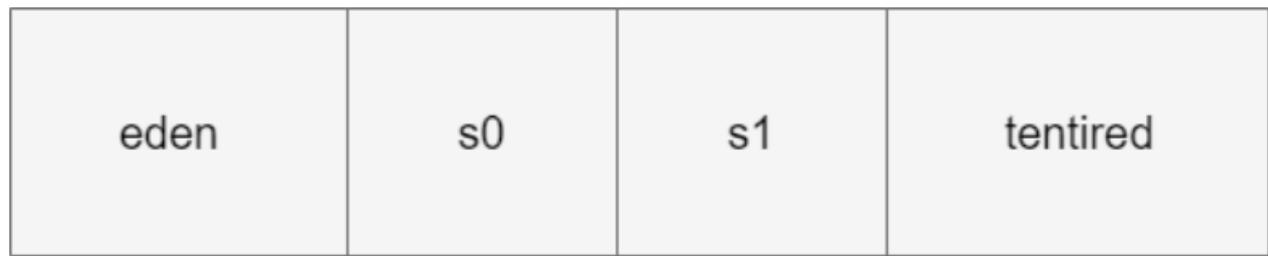
本地方法栈

本地方法栈用途与 **Java** 虚拟机栈一样，只是服务于 Native 本地方法。它会出现 **StackOverFlowError** 和 **OutOfMemoryError** 两种异常。在 HotSpot 虚拟机中和 **Java** 虚拟机栈合二为一。

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；再细致一点有：Eden 空间、From Survivor、To Survivor 空间等。进一步划分的目的是更好地回收内存，或者更快地分配内存。



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

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。但在 JDK 1.8 开始，方法区被取消，取而代之的是在直接内存中划分了一片区域称为元空间。

方法区常用参数：

|参数|功能|

|:---:|---|

|`-XX:PermSize=N` |方法区（永久代）初始大小|

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

元空间

JDK 1.8 的时候，方法区（HotSpot 的永久代）被彻底移除了（JDK1.7 就已经开始了），取而代之是元空间，元空间使用的是直接内存。与方法区（永久代）很大的不同就是，如果不指定大小的话，随着更多类的创建，虚拟机会耗尽所有可用的系统内存。整个方法区（永久代）有一个 JVM 本身设置固定大小上限，无法进行调整，而元空间使用的是直接内存，受本机可用内存的限制，并且永远不会得到 `java.lang.OutOfMemoryError`。

元空间常用参数：

参数	功能
-XX:MetaspaceSize=N	元空间初始大小
-XX:MaxMetaspaceSize=N	元空间最大大小

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

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。

JDK1.7 对 JVM 内存结构 / Java 虚拟机运行时数据区有何改变？

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。

JDK1.8 对 JVM 内存结构/ Java 虚拟机运行时数据区有何改变？

取消了方法区，取而代之的是在直接内存中设置了元空间，代替其作用

对象晋升到老年带的年龄阈值参数

-XX:MaxTenuringThreshold

方法区和永久带的关系

方法区和永久带的关系很像 Java 中接口和类的关系，类实现了接口，而永久带就是 HotSpot 虚拟机对虚拟机规范中方法区的一种实现方式。也就是说，永久带是 HotSpot 的概念，方法区是 Java 虚拟机规范中的定义，是一种规范，而永久带是一种实现，一个是标准一个是实现，其他的虚拟机实现并没有永久带这一说法。

为什么要将永久带 (PermGen) 替换为元空间 (Metaspace) 呢？

除了一些底层原因外，还有整个永久带有一个 JVM 本身设置固定大小上限，无法进行调整，而元空间使用的是直接内存，受本机可用内存的限制，并且永远不会得到 `java.lang.OutOfMemoryError`。你可以使用 `-XX: MaxMetaspaceSize` 标志设置最大元空间大小，默认值为 `unlimited`，这意味着它只受系统内存的限制。`-XX: MetaspaceSize` 调整标志定义元空间的初始大小如果未指定此标志，则 Metaspace 将根据运行时的应用程序需求动态地重新调整大小。

JVM 内存分配的方法？

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。



JVM 内存分配时的并发问题

由于创建对象是非常频繁的操作，因此 JVM 通过两种方式创建对象：

1. **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。

2. **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。

如何设置堆的大小？

JVM 初始分配的内存由 `-Xms` 指定，默认是物理内存的1/64；JVM 最大分配的内存由 `-Xmx` 指定，默认是物理内存的1/4。默认空余堆内存小于40%时，JVM 就会增大堆到 `-Xmx` 的最大限制；空余堆内存大于70%时，JVM 会减少堆直到 `-Xms` 的最小限制。因此服务器一般设置 `-Xms`、`-Xmx` 相等以避免在每次GC后调整堆的大小。

如何设置非堆（方法区）内存的大小？

JVM 使用 `-XX:PermSize` 设置非堆内存初始值，默认是物理内存的1/64；由 `-XX:MaxPermSize` 设置最大非堆内存的大小，默认是物理内存的1/4。

JVM 最大内存限制多少？

首先 JVM 内存限制于实际的最大物理内存，假设物理内存无限大的话，JVM 内存的最大值跟操作系统有很大的关系。简单的说就32位处理器虽然可控内存空间有4GB,但是具体的操作系统会给一个限制，这个限制一般是2GB-3GB（一般来说 Windows 系统下为1.5G-2G, Linux 系统下为2G-3G），而64bit以上的处理器就不会有限制了。

OOM出现

虚拟机栈或本地方法栈堆动态扩容不足，Java 堆中引用无法正确释放等。

JVM 存放变量的位置

对于局部方法变量，存放在 Java 虚拟机栈中的局部变量表中；

对于本地（Native）方法的变量，存放在本地方法栈中的响应区域中；

对于静态变量，存放在方法区（或元空间）中；

对于实例变量，存放在堆中。

Java 创建对象过程？

Java 创建对象的过程

公众号: JavaGuide



- 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。
- 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。
- 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。
- 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。
- 执行方法:** 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。

JVM 是如何实现线程的？

在JVM规范里是没有规定线程要以哪种形式创建——具体实现用1:1（内核线程）、N:1（用户态线程）、M:N（混合）模型的任何一种都可以。Java并不暴露出不同线程模型的区别，上层应用是感知不到差异的（只是性能特性会不太一样...）。其中用户线程在内核之上支持，并在用户层通过线程库来实现。不需要用户态/核心态切换，速度快。操作系统内核不知道多线程的存在，因此一个线程阻塞将使得整个进程（包括它的所有线程）阻塞。由于这里的处理器时间片分配是以进程为基本单位，所以每个线程执行的时间相对减少。内核线程由操作系统直接支持。由操作系统内核创建、调度和管理。内核维护进程及线程的上下文信

息以及线程切换。一个内核线程由于I/O操作而阻塞，不会影响其它线程的运行。

Java 对象结构 / Java 对象内存布局

Java 对象的内存分配在堆中，其中，在 Hotspot 虚拟机中，对象在内存中的布局可以分为 3 块区域：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。

Java 对象访问方式

建立对象就是为了使用对象，Java 程序通过栈上的 reference 数据来操作堆上的具体对象。对象的访问方式由虚拟机实现而定，目前主流的访问方式有两种：

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。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，reference 本身不需要修改。

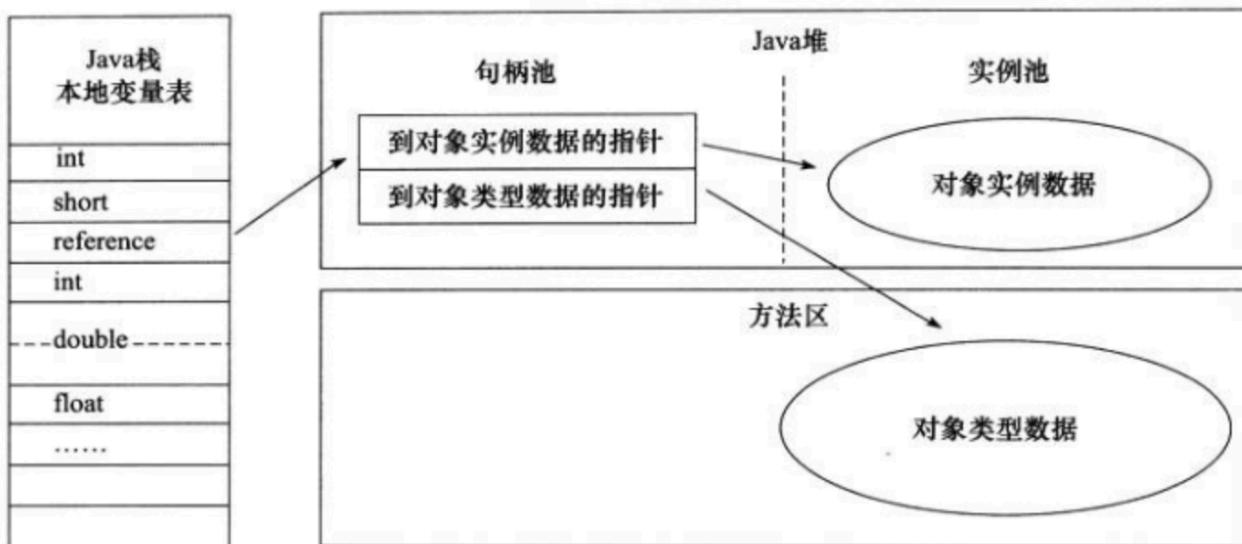


图 2-2 通过句柄访问对象

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

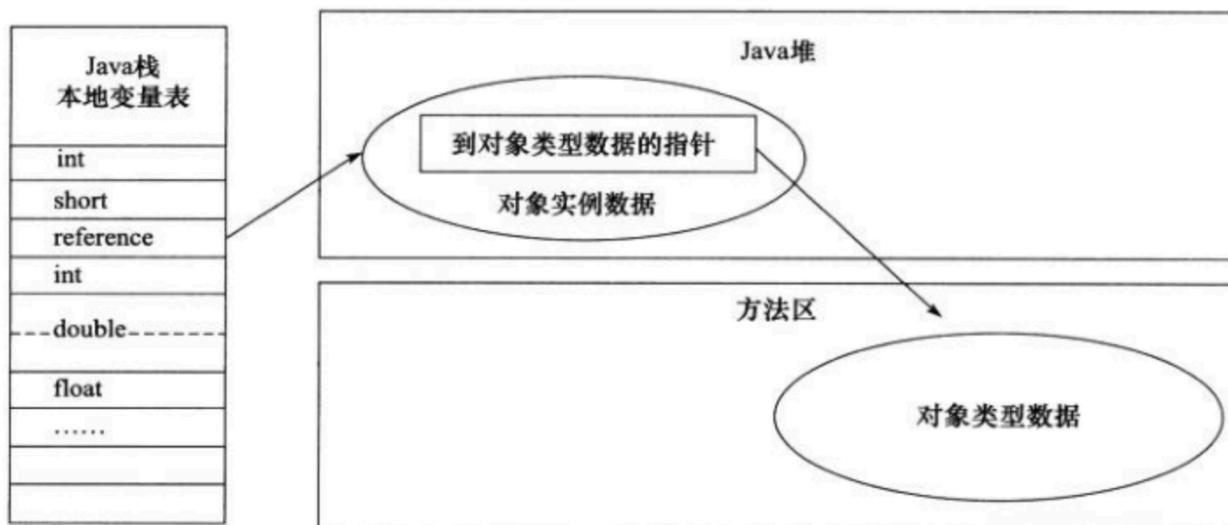


图 2-3 通过直接指针访问对象

String 对象的两种创建方式

```
String str1 = "abcd"; // 先检查字符串常量池中有没有"abcd", 如果字符串常量池中没有, 则创建一个, 然后 str1 指向字符串常量池中的对象, 如果有, 则直接将 str1 指向"abcd";
String str2 = new String("abcd"); // 堆中创建一个新的对象
String str3 = new String("abcd"); // 堆中创建一个新的对象
System.out.println(str1==str2); // false
System.out.println(str2==str3); // false
```

1. 在常量池中拿对象: 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, 在 JDK1.6 中, `intern()` 方法会把首次遇到的字符串实例复制到永久代 (运行时常量池) 中, 返回的也是永久代中这个字符串的引用; 而在 JDK1.7 中, `intern()` 则不会再复制实例, 只是在常量池中记录首次出现的实例的引用。
2. 直接在堆内存空间创建一个新的对象: 只要使用 `new` 方法, 便需要创建新的对象。

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

```
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
```

如果运行时常量池中已经存在了“abc”字符串，则只会在 new 时创建一个对象；

如果运行时常量池不存在“abc”，则会先在运行时常量池中创建一个“abc”字符串对象，然后再在 new 的时候创建一个对象，一共创建两个对象。

线程什么时候创建在 JVM？

(...)

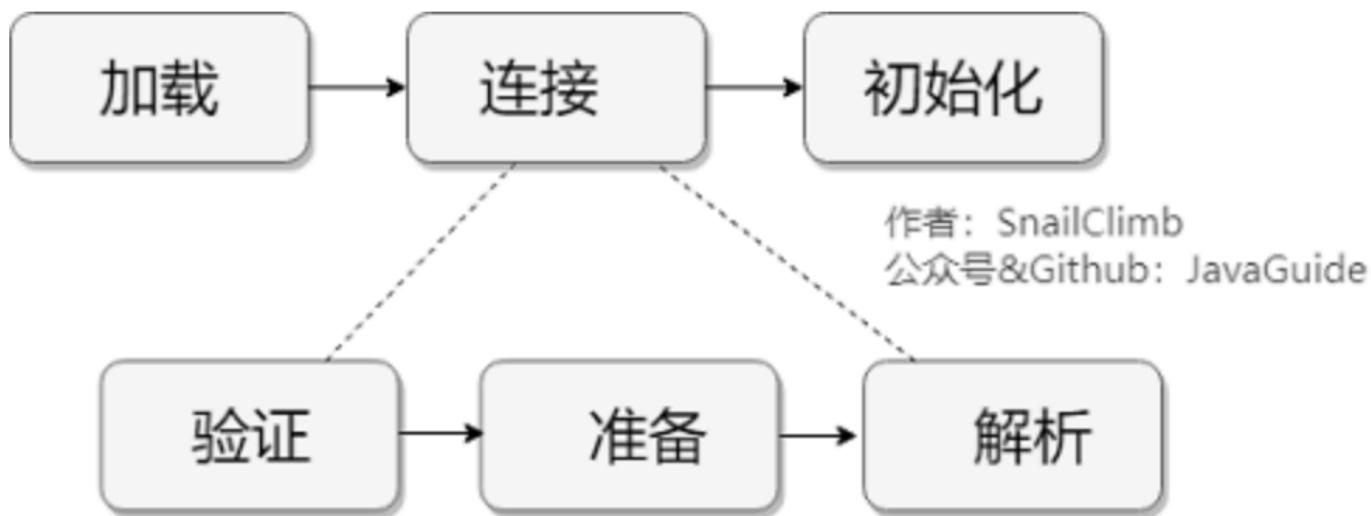
JVM 是否一定会创建线程？

(...)

类的唯一性决定条件

类本身和加载这个类的类加载器。只有两个类是同一类加载器加载的前提下，两个类才可能相等。否则，即使两个类源自同一个 .class 文件，被同一个 JVM 加载，只要加载它们的类加载器不相同，这两个类必定不相等。

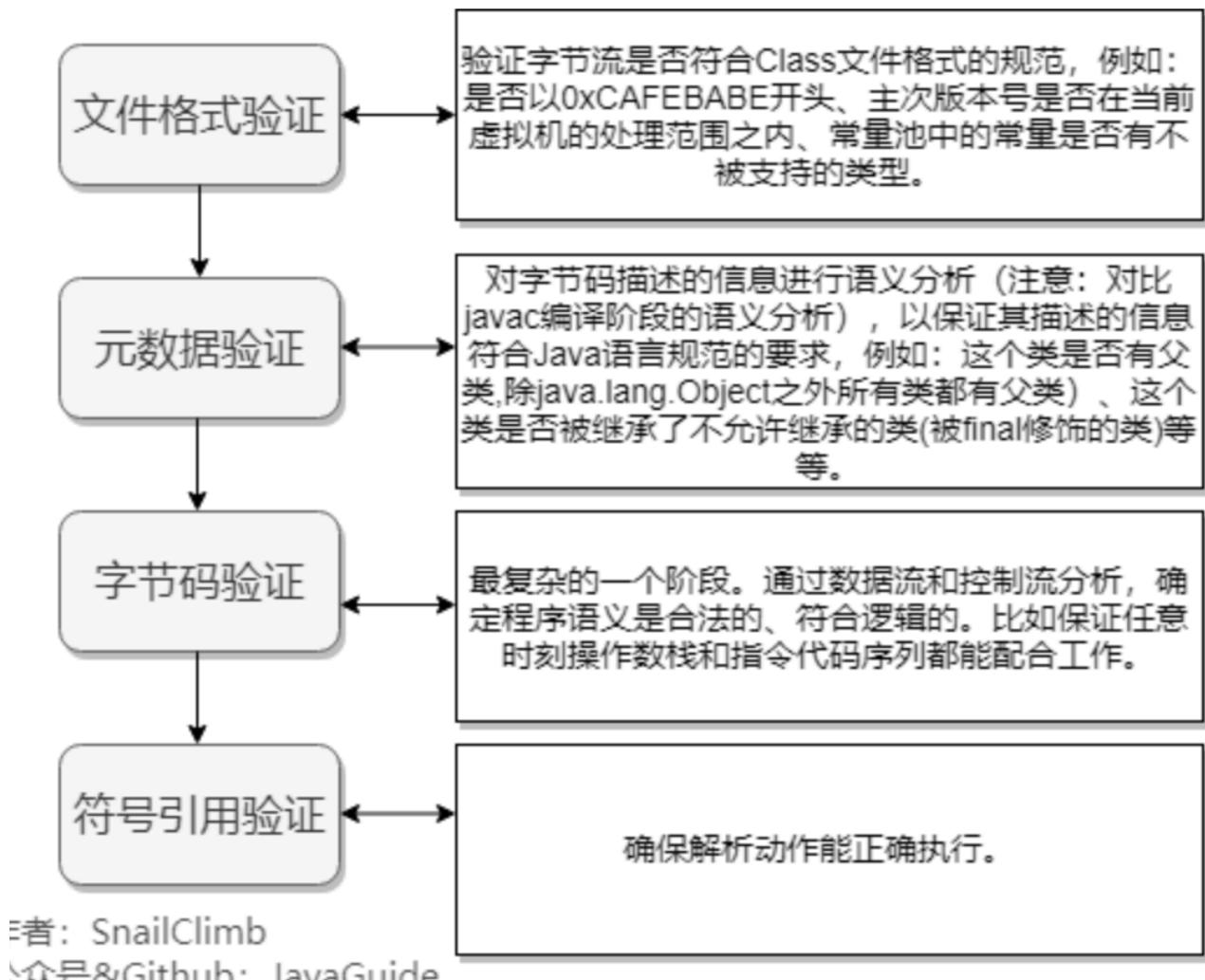
Java 类加载过程？



1. 加载: 1. 通过全类名获取定义此类的二进制字节流。 2. 将字节流所代表的静态存储

结构转换为方法区的运行时数据结构。3. 在内存中生成一个代表该类的 Class 对象,作为方法区这些数据的访问入口。一个非数组类的加载阶段（加载阶段获取类的二进制字节流的动作）是可控性最强的阶段，这一步我们可以去完成还可以自定义类加载器去控制字节流的获取方式（重写一个类加载器的 `loadClass()` 方法）。数组类型不通过类加载器创建，它由 Java 虚拟机直接创建。

2. 验证：



3. 准备：准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。对于该阶段有以下几点需要注意：
 1. 这时候进行内存分配的仅包括类变量（static），而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在 Java 堆中。
 2. 这里所设置的初始值“通常情况”下是数据类型默认的零值（如0、0L、null、false等），比如我们定义了public static int value=111，那么 value 变量在准备阶段的初始值就是0而不是111（初始化阶段才会复制）。特殊情况：比如给 value 变量加上了 final 关键字public static final int value=111，那么准备阶段 value 的值就被复制为 111。
4. 解析：解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用限定符7类符号

引用进行。Java 虚拟机为每个类都准备了一张方法表来存放类中所有的方法。当需要调用一个类的方法的时候，只要知道这个方法在方法表中的偏移量就可以直接调用该方法了。通过解析操作符号引用就可以直接转变为方法在类中方法表的位置，从而使得方法可以被调用。综上，解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，也就是得到类或者字段、方法在内存中的指针或者偏移量。

5. **初始化**：初始化是类加载的最后一步，也是真正执行类中定义的 Java 程序代码(字节码)，初始化阶段是执行类构造器 `<clinit>()` 方法的过程。对于 `<clinit>()` 方法的调用，虚拟机会自己确保其在多线程环境中的安全性。因为 `<clinit>()` 方法是带锁线程安全，所以在多线程环境下进行类初始化的话可能会引起死锁，并且这种死锁很难被发现。

必须立即对类进行初始化的情况？ / 主动引用的情况？

除了一下情况下，其他都是被动引用。

1. 当遇到 `new` 、 `getstatic` 、 `putstatic` 或 `invokestatic` 这4条直接码指令时，比如 `new` 一个类，读取一个静态字段(未被 `final` 修饰)、或调用一个类的静态方法时。
2. 使用 `java.lang.reflect` 包的方法对类进行反射调用时，如果类没初始化，需要触发其初始化。
3. 初始化一个类，如果其父类还未初始化，则先触发该父类的初始化。
4. 当虚拟机启动时，用户需要定义一个要执行的主类 (包含 `main` 方法的那个类)，虚拟机会先初始化这个类。
5. 当使用 JDK1.7 的动态语言时，如果一个 `MethodHandle` 实例的最后解析结构为 `REF_getStatic`、`REF_putStatic`、`REF_invokeStatic`、的方法句柄，并且这个句柄没有初始化，则需要先触发器初始化。

什么是符号引用？什么是直接引用？

符号引用与虚拟机实现的布局无关，引用的目标并不一定要已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须是一致的，因为符号引用的字面量形式明确定义在 Java 虚拟机规范的 `.class` 文件格式中。

直接引用可以是指向目标的指针，相对偏移量或是一个能间接定位到目标的句柄。如果有直接引用，那引用的目标必定已经在内存中存在。

类加载器的种类？

从 JVM 的角度看有两种：

1. 启动类加载器 (C++语言实现, 是 JVM 的一部分)
2. 其他类加载器 (Java 语言实现, 独立于 JVM, 全部都是继承自抽象类 `java.lang.ClassLoader`)。

从 Java 开发人员来看有三种:

1. 启动类加载器 (负责加载 `JAVA_HOME\lib` 目录中的, 或通过 `-Xbootclasspath` 参数指定路径中的, 且被虚拟机认可 (按文件名识别, 如 `rt.jar`) 的类, 它无法被 Java 程序直接引用)
2. 扩展类加载器 (负责加载 `JAVA_HOME\lib\ext` 目录中的, 或通过 `java.ext.dirs` 系统变量指定路径中的类库, 开发者可以直接使用)
3. 应用程序类加载器 (负责加载用户路径 (`classpath`) 上的类库, 开发者可以直接使用))

什么是双亲委派模型? 它有什么作用?

当一个类加载器收到类加载任务, 会先交给其父类加载器去完成, 因此最终加载任务都会传递到顶层的启动类加载器, 只有当父类加载器无法完成加载任务时, 才会尝试执行加载任务。采用双亲委派的一个好处是保证 Java 核心库的安全性。比如加载位于 `.rt.jar` 包中的类 `java.lang.Object`, 不管是哪个加载器加载这个类, 最终都是委托给顶层的启动类加载器进行加载, 这样就保证了使用不同的类加载器最终得到的都是同样一个 Object 对象。

什么时候会用到双亲委派模型? / 如何避免双亲委派模型?

为了避免双亲委托机制, 我们可以自己定义一个类加载器, 然后重载 `loadClass()` 即可。

GC基本原理

对于 GC 来说, 当程序员创建对象时, GC 就开始监控这个对象的地址、大小以及使用情况。通常, GC 采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是“可达的”, 哪些对象是“不可达的”。当 GC 确定一些对象为“不可达”时, GC 就有责任回收这些内存空间。可以。程序员可以手动执行 `System.gc()`, 通知 GC 运行, 但是 Java 语言规范并不保证 GC 一定会执行。

GC算法?

1. 标记-清除算法

- 2. 复制算法
- 3. 标记-整理算法
- 4. 分代收集算法

STW

STW 即 Stop The World 的简称，指 JVM 在进行 Full GC 时将所有工作线程暂停、挂起来进行垃圾回收。过多的 STW 会影响程序性能。

Full GC 发生的条件？

- 1. 调用 `System.gc()`
- 2. **老年代空间不足**：老年代空间不足的常见场景为前文所讲的大对象直接进入老年代、长期存活的对象进入老年代等，当执行 Full GC 后空间仍然不足，则抛出 `Java.lang.OutOfMemoryError`。为避免以上原因引起的 Full GC，调优时应尽量做到让对象在 Minor GC 阶段被回收、让对象在新生代多存活一段时间以及不要创建过大的对象及数组。
- 3. **空间内存担保失败**
- 4. **Concurrent Mode Failure**：执行 CMS GC 的过程中同时有对象要放入老年代，而此时老年代空间不足（有时候“空间不足”是 CMS GC 时当前的浮动垃圾过多导致暂时性的空间不足触发 Full GC），便会报 Concurrent Mode Failure 错误，并触发 Full GC。

怎么减少 Full GC？

- 1. 方法区空间增大
- 2. 老年代空间增大
- 3. 新生代空间减小
- 4. 禁止使用 `System.gc()` 方法（或者少使用）
- 5. 使用标记-整理算法，尽量让连续空间保持最大
- 6. 排查代码中的无用大对象(内存泄漏)

你使用的 Java JDK 版本，该版本有哪些 GC，其运行机制是怎样的？

(从 堆的分区 + 关注堆和元空间（注意JDK 1.8 已经没有方法区了！） + GC算法 + GC 回收器等方面来谈)

GC 中如何判断对象是否需要被回收？

1. 引用计数法
2. 可达性分析算法

JVM GC Root 以及为什么它们可以是 GC Root?

JVM 栈中局部变量表中的引用对象、方法区中类静态属性引用的对象、方法区中常量引用的对象、本地方法栈中 JNI 引用的对象。因为这些节点主要在全局性的引用或者执行上下文中，不会被轻易回收。

不可达的对象是否“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。当对象没有重写 `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。

如何判断一个常量是一个废弃的常量？

运行时常量池主要回收的是废弃的常量。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。

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。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”：

1. 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。
2. 加载该类的 `ClassLoader` 已经被回收。
3. 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。

四种引用？软引用和弱引用的区别？软引用和强引用的区别？

1. 强引用：大部分对象是强引用的。强引用是指必要的引用，是 GC 不会回收的引用。
2. 软引用：软引用是指可有可无的引用，只有内存不够时 GC 才会回收。
3. 弱引用：弱引用是当 GC 一旦发现就会立即回收的引用。
4. 虚引用：虚引用即没有引用，必须和引用队列联合使用。

虚引用作用？

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。

垃圾回收器可以马上回收内存吗？

垃圾回收器通常是一个单独的低级别的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清楚和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。程序员可以手动执行 `System.gc()`，通知 GC 运行，但是 Java 语言规范并不保证 GC 一定会执行。

有什么办法可以主动通知虚拟机进行垃圾回收呢？

`System.gc()`

请说明一下 Eden 区和 Survivor 区的含义以及工作原理？

Eden 区和 Survivor 区是 Java 堆的年轻代的两种分区，一般有两个 Survivor 区（From Survivor 和 To Survivor），且每一次 Minor GC 后，两个 Survivor 区的角色会互换。对象基本上都是在 Eden 区内创建的（除了如数组等大对象直接在老年代中创建），然后每次 Minor GC 的时候通过复制算法将 Eden 区和 From Survivor 区中存活的对象复制到 To Survivor 区中，然后两个 Survivor 区角色互换。大约经历 15 次 Minor GC 后，Survivor 区中存活的对象会进入老年代中。

为什么要分代？

对传统的、基本的 GC 实现来说，由于它们在 GC 的整个工作过程中都要“stop-the-world”，导致整个 GC 堆耗时太长，因此需要想办法只收集其中的一部分以减少停顿时间。因此而诞生了几种不同的划分 (partition) GC 堆的方式来实现部分收集，而分代式 GC 就是这其中的一个思路。

分配担保机制？

把新生代的对象提前转移到老年代中去，只要老年代上的空间足够存放，就不会出现 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。

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）。原因：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。

动态对象年龄判定

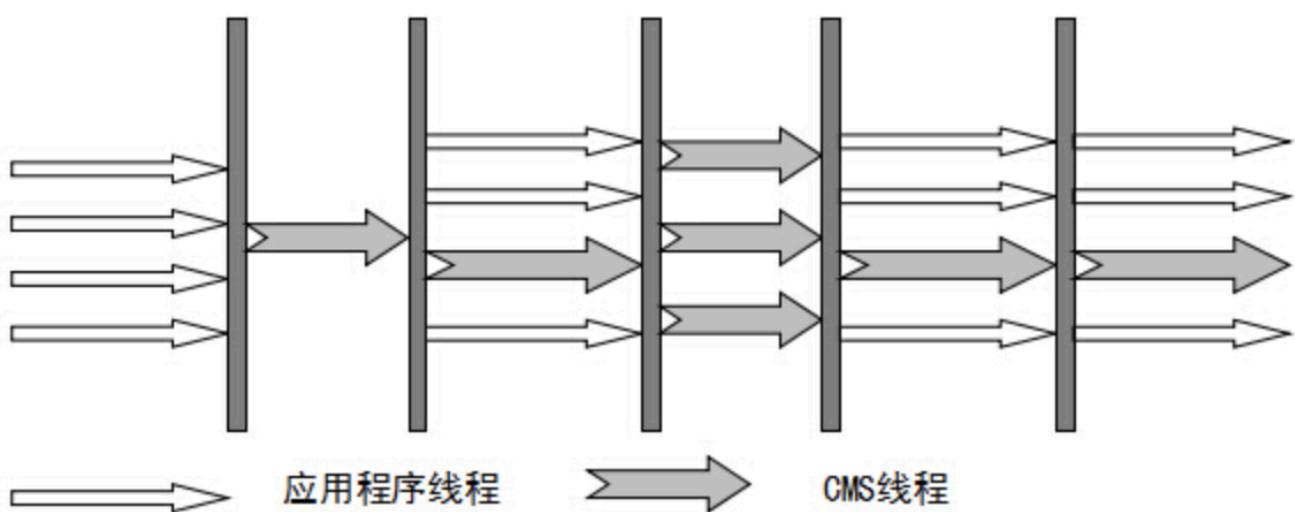
如果 Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无需达到要求的年龄。

什么原因会导致Minor GC运行频繁？同样的，什么原因又会导致Minor GC运行很慢？请简要说明一下

可能是堆内存太小或者是年轻代分配内存过小。

简要介绍一下CMS的回收流程？

应用程序线程 初始标记 并发标记 重新标记 并发清理 并发重置



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：

1. **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。
2. **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。
3. **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。
4. **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。

CMS采用哪种回收算法？

标记-清除算法 —— 会导致碎片问题

CMS 的浮动垃圾问题以及解决方法

由于 CMS 在并发清除的时候还有应用线程在工作，导致这些线程产生的垃圾在 GC 中无法被清除，而到下一次 CMS 并发清除又有一段时间，可能导致“Concurrent Mode Failure”而引发 Full GC，因此 CMS 不能像别的收集器一样等到老年代快满的时候才开始工作，而是

应该到达一定的阈值后进行一次回收。可以通过设置参数 `-XX:`

`CMSInitiatingOccupancyFraction` 来设置该阈值

CMS 遇到“Concurrent Mode Failure”会怎样？

会启动后备预案：临时启动Serial Old收集器重新进行老年代的垃圾收集，但这样会导致停顿时间延长。

使用 CMS 怎样解决内存碎片的问题呢？

开启 `-XX: +UseCMSCompactAtFullCollection` 参数，用于在 CMS 顶不住要进行 Full GC 时开启碎片合并过程，但这样解决了碎片问题，停顿时间又变长了。因此，可以设置 `-XX: CMSFullGCsBeforeCompaction` 参数用于设置执行多少次不压缩的 Full GC 后进行一次带压缩的 Full GC

介绍一下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：

1. **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。
2. **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。
3. **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。**
4. **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：

1. 初始标记
2. 并发标记
3. 最终标记
4. 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）。

Servlet

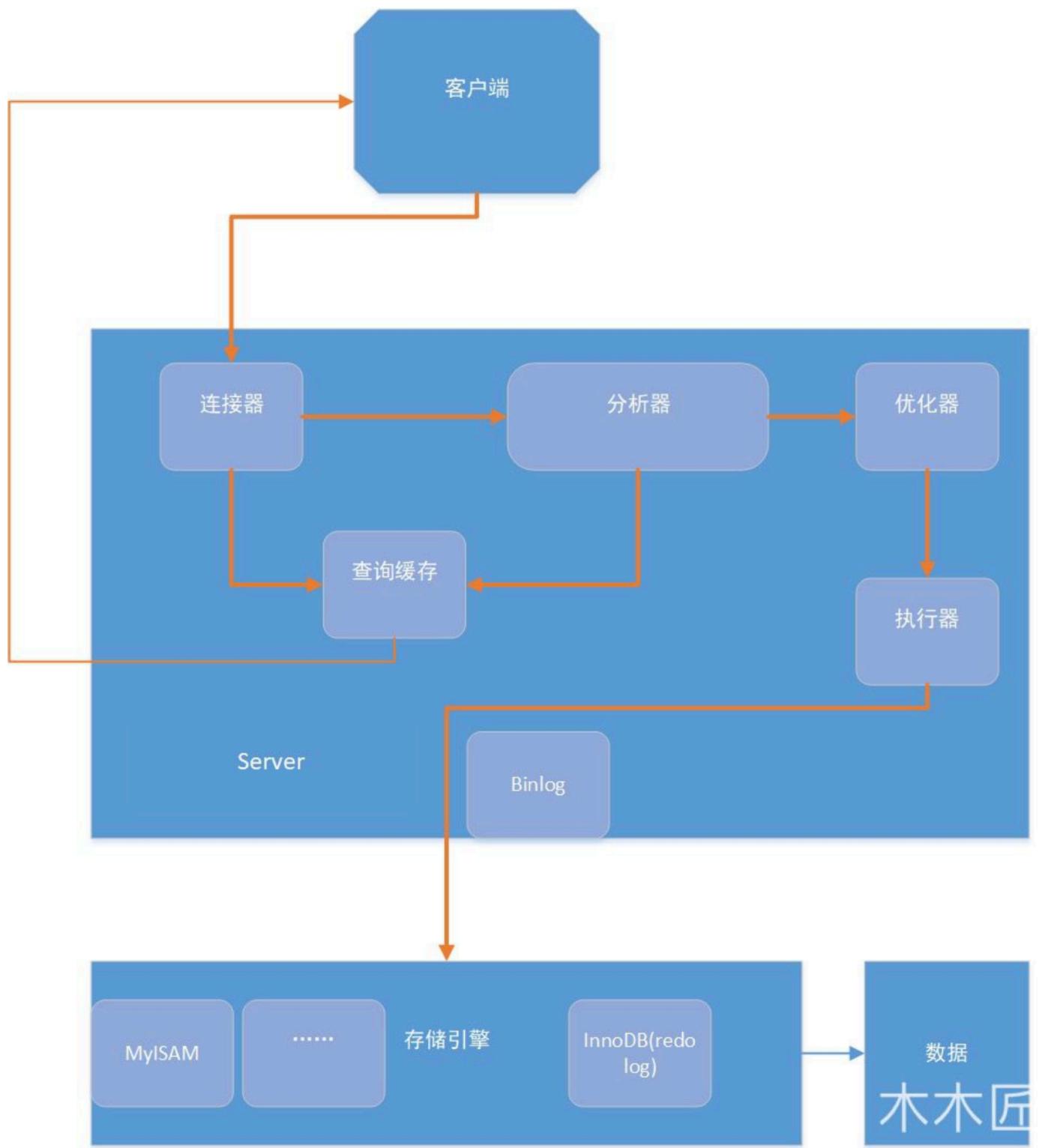
Spring

MySQL

MySQL基础语法?

(...)

MySQL 基础架构?



简单来说 MySQL 主要分为 **Server 层** 和 **存储引擎层**：

Server 层：主要包括连接器、查询缓存、分析器、优化器、执行器等，所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图，函数等，还有一个通用的日志模块 binlog 日志模块。

存储引擎：主要负责数据的存储和读取，采用可以替换的插件式架构，支持 InnoDB、MyISAM、Memory 等多个存储引擎，其中 InnoDB 引擎有自有的日志模块 redolog 模块。现在最常用的存储引擎是 InnoDB，它从 MySQL 5.5.5 版本开始就被当做默认存储引擎了。

Server 层组成部分介绍？

连接器：连接器主要和身份认证和权限相关的功能相关，就好比一个级别很高的门卫一样。主要负责用户登录数据库，进行用户的身份认证，包括校验账户密码，权限等操作，如果用户账户密码已通过，连接器会到权限表中查询该用户的所有权限，之后在这个连接里的权限逻辑判断都是会依赖此时读取到的权限数据，也就是说，**后续只要这个连接不断开，即时管理员修改了该用户的权限，该用户也是不受影响的。**

查询缓存（MySQL 8.0 后移除）：查询缓存主要用来缓存我们所执行的 **SELECT** 语句以及该语句的结果集。连接建立后，执行查询语句的时候，会先查询缓存，MySQL 会先校验这个 SQL 是否执行过，以 Key-Value 的形式缓存在内存中，Key 是查询预计，Value 是结果集。如果缓存 key 被命中，就会直接返回给客户端，如果没有命中，就会执行后续的操作，完成后也会把结果缓存起来，方便下一次调用。当然在真正执行缓存查询的时候还是会校验用户的权限，是否有该表的查询条件。MySQL 查询不建议使用缓存，因为查询缓存失效在实际业务场景中可能会非常频繁，假如你对一个表更新的话，这个表上的所有的查询缓存都会被清空。对于不经常更新的数据来说，使用缓存还是可以的。所以，一般在大多数情况下我们都是不推荐去使用查询缓存的。MySQL 8.0 版本后删除了缓存的功能，官方也是认为该功能在实际的应用场景比较少，所以干脆直接删掉了。

分析器：MySQL 没有命中缓存，那么就会进入分析器，分析器主要是用来分析 SQL 语句是来干嘛的，分析器也会分为几步：

1. 词法分析，一条 SQL 语句有多个字符串组成，首先要提取关键字，比如 **select**，提出查询的表，提出字段名，提出查询条件等等。做完这些操作后，就会进入第二步。
2. 语法分析，主要就是判断你输入的 SQL 是否正确，是否符合 MySQL 的语法。

优化器：它的作用就是按照它认为的最优的执行方案去执行（有时候可能也不是最优），比如多个索引的时候该如何选择索引，多表查询的时候如何选择关联顺序等。可以说，经过了优化器之后可以说这个语句具体该如何执行就已经定下来。

执行器：当选择了执行方案后，MySQL 就准备开始执行了，首先执行前会校验该用户有没有权限，如果没有权限，就会返回错误信息，如果有权限，就会去调用引擎的接口，返回接口执行的结果。

binlog：通用日志模块。

WHERE 和 HAVING区别？

WHERE 是直接从表中进行筛选，而 **HAVING** 是从前面已经筛选过的字段中进一步筛选。即 **HAVING** 后面的字段必须在前面出现过，否则只能用 **WHERE**；而当 **WHERE** 后面的字段是

经过处理如avg()、max()……等方法处理后，则意味着处理后的字段不存在原表中，此时则必须使用 HAVING 而不能使用 WHERE。

AUTO_INCREMENT 有什么需要注意的？

自增字段必须是索引，而且是索引的第一列，不一定要是主键。

MySQL 获取 AUTO_INCREMENT 下一值的方法？

```
mysql>CREATE TABLE `get_max_id` (
    `id` int(11) unsigned NOT NULL AUTO_INCREMENT COMMENT '业务主键',
    `content` char(25) DEFAULT NULL COMMENT '业务内容',
    PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

1. **max(id)**: 直接简单获取，同时，它无视其它客户端连接（db_connection）的影响。

```
mysql>select max(id) from get_max_id;
+-----+
| max(id) |
+-----+
|      NULL |
+-----+
row in set (0.00 sec)
```

2. **LAST_INSERT_ID()**: 返回最后一个INSERT 或 UPDATE 查询中，AUTO_INCREMENT 列设置的第一个表的值。但它有些限制的：

1. 同一个 Connection 连接对象(同一客户端)中，SELECT 的结果为最后一次 INSERT 的 AUTO_INCREMENT 属性列的 ID。这句话的重点在于“同一个”，即其他连接的客户端不对其查询的结果造成影响。假设客户端 A 和 B，表 ta 原自增 ID 为3，在A中插入记录后产生自增 ID 为4，在客户端 A 中通过该函数查询的结果为4，但在客户端 B 中查询的结果值仍为3。
2. 与表无关，即假设 ta 表和 tb 表，向 ta 插入记录后，再向 tb 插入记录，结果值为 tb 的max(id) 值。
3. 使用非魔术方法 ('magic') 来 INSERT 或 UPDATE 一条记录时，即使用非0/非 NULL 值作为插入的字段，则 LAST_INSERT_ID() 返回值不会发生变化。
4. 同一条 INSERT 语句中，传入多个 VALUES 值，则 LAST_INSERT_ID() 返回值为该查询第一条记录的ID。
5. 在进阶方面，可运用作分表ID的唯一性。

```
mysql>select LAST_INSERT_ID();
+-----+
| LAST_INSERT_ID() |
+-----+
|          0 |
+-----+
row in set (0.00 sec)
```

3.查看表状态 **show table status**: 该方式提供了当前 DB (use db_name;) 下每个表的基本信息；可以通过 WHERE 条件获取到 Auto_increment 属性的值。

```
mysql> show table status where Name='get_max_id';
+-----+-----+-----+-----+-----+-----+-----+-----+
| Name      | Engine | Version | Row_format | Rows | Avg_row_length | Data_length | Max_data_length | Index_length | Data_free | Auto_increment | Create_time   | Update_time | Check_time | Collation       | Checksum | Create_options | Comment |
+-----+-----+-----+-----+-----+-----+-----+-----+
| get_max_id | InnoDB |      10 | Compact    |     0 |          0 | 16384 |           0 |          0 | 10485760 |          1 | 2015-04-20 11:49:07 | NULL | utf8_general_ci |      NULL |           |
|           |       |      |       |       |       |       |       |       |       |       |       |       |       |       |       |
+-----+-----+-----+-----+-----+-----+-----+-----+
row in set (0.00 sec)
```

4.**information_schema.tables**: 提供关于数据库中的表（包括视图）的信息。详细描述了某个表属于哪个schema，表类型，表引擎等等信息。

```
mysql> select table_name, AUTO_INCREMENT from information_schema.tables where table_name="get_max_id";
+-----+-----+
| table_name | AUTO_INCREMENT |
+-----+-----+
| get_max_id |          1 |
+-----+-----+
```

```
1 row in set (0.01 sec)
```

5.@@IDENTITY全局变量：（基础：以@@开头的变量为全局变量，而以@开头的变量为用户自定义的变量。）此处 @@IDENTITY 表示最近一次向具有 identity 属性 (auto_increment) 的表 INSERT 数据时对应的自增列的值。此处得到的值是0。

1. 类似于 LAST_INSERT_ID() 函数，该方式必须在同一个客户端内进行的 INSERT 与 SELECT，且不受其他客户端影响。
2. 与表无关。
3. 非魔术方法插入不影响结果值。
4. 同一 INSERT 插入多条记录，取第一条记录的 ID 值为结果。

```
mysql> select @@IDENTITY;
+-----+
| @@IDENTITY |
+-----+
|      0 |
+-----+
1 row in set (0.00 sec)
```

一条MySQL语句的执行流程？

查询语句执行流程

```
select * from tb_student A where A.age='18' and A.name=' 张三 ';
```

1. 连接器检查用户权限，如果没有权限，则返回错误信息，否则进行下一步。
2. 在 MySQL 8.0 之前，会将这条语句作为 key 查询缓存，如果有缓存结果，则直接返回结果集，结束查询，否则进入下一步。在 MySQL 8.0 后，则直接忽略此步。
3. 分析器提取并分析关键字，提取名为 tb_student 的表，提取需要查询所有的列，查询条件是这个表的 id='1'。然后判断这个 SQL 语句是否有语法错误，比如关键词是否正确等等，如果检查没问题就执行下一步。
4. 优化器根据自己的优化算法进行选择执行效率最好的一个方案（优化器认为，有时候不一定最好）。
5. 执行器根据上述方案调用对应的引擎接口执行查询操作并返回结果。

更新语句执行流程

```
update tb_student A set A.age='19' where A.name=' 张三 ';
```

执行更新的时候肯定需要记录日志，这就会引入 MySQL 自带的日志模块式 binlog（归档日

志) , 所有的存储引擎都可以使用, 常用的 InnoDB 引擎还自带了一个日志模块 redo log (重做日志) , 现在就以 InnoDB 模式下来探讨这个语句的执行流程。

1. 连接器检查用户权限, 如果没有权限, 则返回错误信息, 否则进行下一步。
2. 在 MySQL 8.0 之前, 会将这条语句作为 key 查询缓存, 如果有缓存结果, 则直接返回结果集, 结束查询, 否则进入下一步。在 MySQL 8.0 后, 则直接忽略此步。
3. 分析器提取并分析关键字, 提取名为 tb_student 的表, 提取需要查询所有的列。然后判断这个 SQL 语句是否有语法错误, 比如关键词是否正确等等, 如果检查没问题就执行下一步。
4. 优化器根据自己的优化算法进行选择执行效率最好的一个方案 (优化器认为, 有时候不一定最好) 。
5. 执行器然后拿到查询的语句, 把 age 改为 19, 然后调用引擎 API 接口, 写入这一行数据
6. InnoDB 引擎把数据保存在内存中, 同时记录 redo log, 此时 redo log 进入 prepare 状态, 然后告诉执行器, 执行完成了, 随时可以提交。
7. 执行器收到通知后记录 binlog。
8. 执行器调用引擎接口, 提交 redo log 为 commit 提交状态。更新完成。

为什么 MySQL 要 binlog 和 redo log 两个日志文件? / MySQL 如何解决数据一致性问题?

这是因为最开始 MySQL 并没与 InnoDB 引擎(InnoDB 引擎是其他公司以插件形式插入 MySQL 的) , MySQL 自带的引擎是 MyISAM, 但是 redo log 是 InnoDB 引擎特有的, 其他存储引擎都没有, 这就导致会没有 crash-safe 的能力(crash-safe 的能力即使数据库发生异常重启, 之前提交的记录都不会丢失), binlog 日志只能用来归档。

并不是说只用一个日志模块不可以, 只是 InnoDB 引擎就是通过 redo log 来支持事务的。那么, 用两个日志模块, 为什么 redo log 要引入 prepare 预提交状态? 这里用反证法来说明下为什么要这么做?

1. 先写 redo log 直接提交, 然后写 binlog, 假设写完 redo log 后, 机器挂了, binlog 日志没有被写入, 那么机器重启后, 这台机器会通过 redo log 恢复数据, 但是这个时候 binlog 并没有记录该数据, 后续进行机器备份的时候, 就会丢失这一条数据, 同时主从同步也会丢失这一条数据。
2. 先写 binlog, 然后写 redo log, 假设写完了 binlog, 机器异常重启了, 由于没有 redo log, 本机是无法恢复这一条记录的, 但是 binlog 又有记录, 那么和上面同样的道理, 就会产生数据不一致的情况。

如果采用 redo log 两阶段提交的方式就不一样了, 写完 binlog 后, 然后再提交 redo log

就会防止出现上述的问题，从而保证了数据的一致性。

MySQL更新数据库时异常重启的处理过程？ / MySQL 如何实现回滚？

那么问题来了，有没有一个极端的情况呢？假设 redo log 处于预提交状态，binlog 也已经写完了，这个时候发生了异常重启会怎么样呢？这个就要依赖于 MySQL 的处理机制了，MySQL 的处理过程如下：

1. 判断 redo log 是否完整，如果判断是完整的，就立即提交。
2. 如果 redo log 只是预提交但不是 commit 状态，这个时候就会去判断 binlog 是否完整，如果完整就提交 redo log，不完整就回滚事务。

这样就解决了数据一致性的
问题。

MySQL如果多次执行同一条 update 语句，会执行成功吗？

在 `binlog_format=row` 和 `binlog_row_image=FULL` 时，由于 MySQL 需要在 binlog 里面记录所有的字段，所以在读数据的时候就会把所有数据都读出来，那么重复数据的 update 不会执行。即 MySQL 调用了 InnoDB 引擎提供的“修改为 (1,55)”这个接口，但是引擎发现值与原来相同，不更新，直接返回。

在 `binlog_format=statement` 和 `binlog_row_image=FULL` 时，InnoDB 内部认真执行了 update 语句，即“把这个值修改成 (1,999)“这个操作，该加锁的加锁，该更新的更新。

介绍一下 SQL 语句预编译？

通常一条 SQL 在 db 接收到最终执行完毕返回可以分为下面三个过程：

1. 词法和语义解析
2. 优化sql语句，制定执行计划
3. 执行并返回结果

我们把这种普通语句称作 Immediate Statements。

但是很多情况，我们的一条 SQL 语句可能会反复执行，或者每次执行的时候只有个别的值不同（比如 query 的 where 子句值不同，update 的 set 子句值不同，insert 的 values 值不同）。如果每次都需要经过上面的词法语义解析、语句优化、制定执行计划等，则效率就明显不行了。

所谓预编译语句就是将这类语句中的值用占位符替代，可以视为将 SQL 语句模板化或者说参数化，一般称这类语句叫 Prepared Statements 或者 Parameterized Statements 预编译语句的优势在于归纳为：一次编译、多次运行，省去了解析优化等过程；此外预编译语句能防止 SQL 注入。当然就优化来说，很多时候最优的执行计划不是光靠知道 SQL 语句的模板

就能决定了，往往就是需要通过具体值来预估出成本代价。

MySQL 如何获得最新插入的一条记录？

通过 `LAST_INSERT_ID()` 获取刚插入的记录的 ID，再用 `SELECT` 获取最新插入记录。

MySQL 集合操作？

(...)

char 与 varchar 的区别？

CHAR 属于固定长度的字符类型，而 VARCHAR 属于可变长度的字符类型。

表 8-1 CHAR 和 VARCHAR 的对比

值	CHAR(4)	存储需求	VARCHAR(4)	存储需求
"	' '	4 个字节	"	1 个字节
'ab'	'ab '	4 个字节	'ab '	3 个字节
'abcd'	'abcd'	4 个字节	'abcd'	5 个字节
'abcdefg'	'abcd'	4 个字节	'abcd'	5 个字节

由于 CHAR 是固定长度的，所以它的处理速度比 VARCHAR 快得多，但是其缺点是浪费存储空间，程序需要对行尾空格进行处理，所以对于那些长度变化不大并且对查询速度有较高要求的数据可以考虑使用 CHAR 类型来存储。

另外，随着 MySQL 版本的不断升级，VARCHAR 数据类型的性能也在不断改进并提高，所以在许多的应用中，VARCHAR 类型被更多地使用。

MyISAM 存储引擎：建议使用固定长度的数据列代替可变长度的数据列。

MEMORY 存储引擎：目前都使用固定长度的数据行存储，因此无论使用 CHAR 或 VARCHAR 列都没有关系。两者都是作为 CHAR 类型处理。

InnoDB 存储引擎：建议使用 VARCHAR 类型，对于 InnoDB 数据表，内部的行存储格式没有区别固定长度和可变长度列（所有数据行都使用指向数据列值的头指针），因此在本质上，使用固定长度的 CHAR 列不一定比使用可变长度 VARCHAR 列性能要好，因而，主要的性能因素是数据行使用的存储总量。由于 CHAR 平均占用的空间多于 VARCHAR，因此使用 VARCHAR 来最小化需要处理的数据行的存储总量和磁盘 I/O 是比较好的。

删除表的几种方式？

DROP

`DROP` 方法适用于完全放弃当前表，删除表全部数据和表结构，立刻释放磁盘空间，不管是

Innodb 和 MyISAM

```
DROP TABLE student;
```

TRUNCATE

TRUNCATE 适用于仅删除当前表中所有的数据，但保留表结构，删除后立刻释放磁盘空间，不管是 Innodb 和 MyISAM

```
TRUNCATE TABLE student;
```

DELETE / DELETE FROM

DELETE 删除表中数据，但保留表结构，删除操作会被记录到 binlog 日志中，可以回滚；而且它可以与 WHERE 连用，进行条件查询删除特定行

如果只针对一张表进行删除，则 DELETE 和 DELETE FROM 效果一样；如果需要联合其他表，则需要使用 DELETE FROM

对于 DELETE 操作，它会删除表全部数据，表结构不变，对于 MyISAM 会立刻释放磁盘空间，InnoDB 不会释放磁盘空间

```
DELETE FROM student;
```

DELETE 操作以后，使用 OPTIMIZE TABLE table_name 会立刻释放磁盘空间，不管是 InnoDB 还是 MyISAM

对于与 WHERE 连用的 DELETE FROM 的带条件的删除，表结构不变，不管是 innodb 还是 MyISAM 都不会释放磁盘空间

```
DELETE FROM student WHERE T_name = "张三";
```

DELETE FROM 表以后虽然未释放磁盘空间，但是下次插入数据的时候，仍然可以使用这部分空间

TRUNCATE VS DELETE

1. 事务：TRUNCATE 删除后不记录 binlog 日志，因此不可以回滚，更不可以恢复数据；而 DELETE 则记录 binlog，所以是可以回滚。因此 TRUNCATE 相当于保留原 MySQL

表的结果，重新创建了这个表，所有的状态都相当于新的，而 DELETE 的效果相当于一行行删除，即可以回滚

2. 效果：效率上 TRUNCATE 比 DELETE 快，而且 TRUNCATE 删除后将重建索引（新插入数据后id从0开始记起），而 DELETE 不会删除索引（新插入的数据将在删除数据的索引后继续增加）
3. TRUNCATE 不会触发任何 DELETE 触发器
4. 返回值：DELETE 操作后返回删除的记录数，而 TRUNCATE 返回的是0或者-1（成功则返回0，失败返回-1）

表连接由哪几种？笛卡儿积用哪种？

ACID 特性详解一下？

原子性 (Atomicity)：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。

一致性 (Consistency)：执行事务前后，数据保持一致，多个事务对同一个数据读取的结果是相同的。

隔离性 (Isolation)：并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的。

持久性 (Durability)：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。

MySQL 并发带来的问题？

脏读 (Dirty read)：当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。

丢失修改 (Lost to modify)：指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。

不可重复读 (Unrepeatable read)：指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。

幻读 (Phantom read) : 幻读与不可重复读类似。它发生在一个事务 (T1) 读取了几行数据, 接着另一个并发事务 (T2) 插入了一些数据时。在随后的查询中, 第一个事务 (T1) 就会发现多了一些原本不存在的记录, 就好像发生了幻觉一样, 所以称为幻读。

MySQL 事务 / 事务隔离级别?

READ-UNCOMMITTED(读取未提交): 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(读取已提交): 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(可串行化): 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。

如何查看事务隔离级别?

```
SELECT @@tx_isolation;
```

```
mysql> SELECT @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| REPEATABLE-READ |
```

如何设置事务隔离级别?

```
SET [SESSION | GLOBAL] TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE}
```

查看全局/会话事务隔离级别的语句?

```
SELECT @@global.tx_isolation; SELECT @@session.tx_isolation; SELECT @@tx_isolation;
```

并发控制语句?

`START TRANSACTION | BEGIN` : 显式地开启一个事务; `COMMIT`: 提交事务, 使得对数据库

做的所有修改成为永久性；ROLLBACK 回滚会结束用户的事务，并撤销正在进行的所有未提交的修改。

MySQL InnoDB 存储引擎的默认支持的隔离级别是？为什么默认是这个隔离级别？

REPEATABLE-READ（可重读）。与 SQL 标准不同的地方在于InnoDB 存储引擎在 REPEATABLE-READ（可重读）事务隔离级别下使用的是 Next-Key Lock 锁(间隙锁)算法，因此可以避免幻读的产生。所以说 InnoDB 存储引擎的默认支持的隔离级别是 REPEATABLE-READ（可重读）已经可以完全保证事务的隔离性要求，即达到了 SQL 标准的 SERIALIZABLE（可串行化）隔离级别。

MySQL InnoDB 存储引擎何时用得到 SERIALIZABLE（可串行化）隔离级别？

分布式事务的情况下

MySQL 更新数据库时异常重启的处理过程？

判断 redo log 是否完整，如果判断是完整的，就立即提交。如果 redo log 只是预提交但不是 commit 状态，这个时候就会去判断 binlog 是否完整，如果完整就提交 redo log，不完整就回滚事务。

MySQL 范式？

MySQL 优化？

1. 避免在 where 子句上进行 null 值判断
2. 慎用 in 与 not in 能用 between 就不要用 in
3. 避免在 where 字句对 = 左边进行表达式或函数操作
4. 组合索引情况下，必须使用到索引的第一个字段才能使索引生效，并且尽可能使字段顺序与索引顺序一致
5. 要注意索引并不是越多越好，索引可以提高查询效率但是会降低更改表的效率，并且也会占用存储空间，所以要视情况而定
6. 可以使用 explain 语句来解析sql方便制定优化方案

MySQL 引擎？

(...)

MySQL 分库分表?

(...)

MySQL 索引?

normal: 表示普通索引。

unique: 表示唯一的, 不允许重复的索引, 可以有 null 值, 如果该字段信息保证不会重复例如身份证号用作索引时, 可设置为 unique, 一个表可以有多个唯一索引。

full text: 表示全文搜索的索引。 FULLTEXT 用于搜索很长一篇文章的时候, 效果最好。用在比较短的文本, 如果就一两行字的, 普通的 INDEX 也可以。

union: 多列值组成一个索引, 专门用于组合搜索, 其效率大于索引合并。

总结, 索引的类别由建立索引的字段内容特性来决定, 通常 normal 最常见。

唯一索引和主键的区别

唯一索引: 一个表中可以有多个唯一索引, 索引中可以存在 null 值。

主键索引: 一种特殊的唯一索引, 一个表中只能有一个主键索引, 索引值不可以为 null。

MySQL 如何查看是否创建索引?

1. `EXPLAIN` 关键字查看 type 字段, 如果是 ALL 则没有索引。
2. `SHOW INDEX FROM table_name`。

如何检测 MySQL 中建立的索引是否生效?

`EXPLAIN` 关键字查看 type 字段: 这是重要的列, 显示连接使用了何种类型。从最好到最差的连接类型为 const、eq_reg、ref、range、index 和 ALL。

const: 表中的一个记录的最大值能够匹配这个查询 (索引可以是主键或唯一索引)。因为只有一行, 这个值实际就是常数, 因为 MySQL 先读这个值然后把它当做常数来对待。

eq_ref: 在连接中, MySQL 在查询时, 从前面的表中, 对每一个记录的联合都从表中读取一个记录, 它在查询使用了索引为主键或唯一键的全部时使用。

ref: 这个连接类型只有在查询使用了不是唯一或主键的键或者是这些类型的部分 (比如, 利用最左边前缀) 时发生。对于之前的表的每一个行联合, 全部记录都将从表中读出。这个

类型严重依赖于根据索引匹配的记录多少—越少越好。

range: 这个连接类型使用索引返回一个范围中的行，比如使用`>`或`<`查找东西时发生的情况。

index: 这个连接类型对前面的表中的每一个记录联合进行完全扫描（比 ALL 更好，因为索引一般小于表数据）。

ALL: 这个连接类型对于前面的每一个记录联合进行完全扫描，这一般比较糟糕，应该尽量避免。

索引是不是越多越好？

不是。索引本身也会占用一定的内存空间，过多的索引会占用过多的内存空间，影响 MySQL 的性能。数据量小的表不需要建立索引，因为建立索引会增加额外的开销。数据变更需要维护索引，因此更多的索引意味着更多的维护成本。

触发器和存储过程区别？

(...)

处理大数据？

分库分表、缓存……

MVCC 实现原理？

MVCC 如何“加锁”？

什么是一致性 hash？

介绍一下负载均衡？

如何保证缓存与数据库的数据一致性？

知道 MySQL 插入和查询分别用的是什么锁吗？

数据库中键的数据结构是？

MySQL 取唯一值（类似随机数）？

MySQL 查询关键字 in 、 exist 什么时候使用，怎么使用？

数据库中一条记录中一个字段，多个线程对它修改，如何不加锁保证只有一个线程修改成功？

MySQL 的容错性、主从分离的详细种类

数据库表中有三列 A、B、C，假设 A 的取值有1000种，B的取值有500种，C 的取值有100种，现在考虑 ABC 上建联合索引，问 SQL 中 ABC 的顺序应该怎么排？

给你一个联合索引 ABC，查询条件为 A=1,B>2,C=3，问你这个索引有没有被用到？

有一张成绩表，写个 SQL 分别统计分数段在 0-10,10-20,20-30...90-100 的人数？

MyBatis

Redis

分布式锁有了解嘛？

Nginx

ZooKeeper

Java工具

计算机网络

OSI分层作用?

| (...)

世界上有那么多主机，但IP有限，那么是如何解决的？

| (...)

说一下IP分类？

| (...)

说一下IP子网划分？

| (...)

说一下IP子网聚合？

| (...)

路由器作用

| (...)

网络层的路由算法有哪些，简述RIP，OSPF过程

| (...)

简述ARP协议过程，是如何通过IP地址获取MAC地址的

(...)

ping命令所使用的协议是什么（ICMP），简述其过程

(...)

ICMP处于哪一层

(...)

介绍一下NAT？

(...)

介绍一下DNS？

DNS是一个分布式的客户机/服务器网络数据库，用于实现IP地址与主机名称之间的映射（反之亦然）。之所以称之为分布式的，是因为互联网中不存在单独一个站点知道所用映射信息。

从应用程序的角度来看，DNS是通过一个域名解析器的应用程序库来实现IP地址与主机名之间的映射操作的。

DNS名称是分层的（划分的层次通过句点“.”分割，大小写不敏感），因此可以实现拓展性。从左往右代表从根标签到子域名标签，每个标签最多可达63个字符长。DNS的分层结构说明没有一个单一的实体需要管理整个DNS名称空间的变化。

名称服务器（DNS服务器）包含的映射信息至少存在两台服务器上，它们拥有完全一样的信息。一台成为主服务器，它的磁盘文件中包含区域数据库；其余的一台或多台服务器称为辅助服务器，它们使用区域传输进程，从主服务器中完整地获取其数据库中的映射信息副本。

名称服务器中包含的IP映射信息，可以从三个渠道获取：

1. 直接来自主服务器的区域数据库（该服务器包含该区域的授权信息，因此该服务器亦可称为授权服务器）
2. 来自区域传输的结果（如一个从属/辅助服务器）

3. 来自处理解析过程的另一台服务器

介绍一下DNS缓存？

大部分名称服务器（DNS服务器）缓存它们学习的区域信息，直到称为生存时间（TTL）的时间限制为止。其中，TTL不宜过大，以减少网络中存在不正确缓存数据的窗口，提高缓存准确性。

缓存同时使用于成功的解析和不成功的解析（否定解析）。缓存否定解析可以在出错的应用程序一再请求不存在的域名时降低网络流量。

介绍一下DNS解析过程？

1. 本地应用程序首先查询本地客户端域名解析器的缓存，如果不存在目标IP，则向本地客户端域名解析器申请域名解析。
2. 如果本地客户端解析器不知道解析地址的IP，则递归地向本地ISP提供的DNS服务器申请解析地址。同样，ISP提供的DNS服务器也会先查询DNS缓存，如果找到，则返回结果；若无法解析，则会再次递归地向上一级DNS域名服务器请求解析，直到根服务器。
3. 如果到达了根域名服务器，根服务器不回进一步处理请求，而是迭代地向下一级域名服务器返回需要联系的下一台根域名服务器的一个或多个IP地址。
4. 根据上一步提供的信息，该级服务器会向对应的根服务器迭代地请求解析，直到拿到解析成功或失败。并将解析结果进行缓存，然后递归地将解析结果按照上面的路径返回给客户端的对应的应用程序。返回过程中，各级服务器会对解析结果进行缓存。

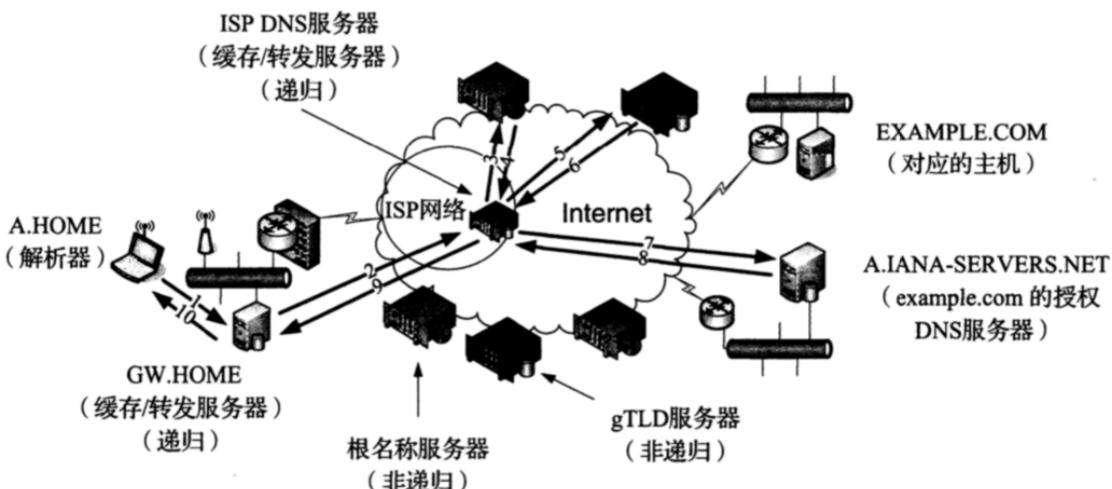


图 11-2 A.HOME 查询 EXAMPLE.COM 的典型递归 DNS 查询过程涉及多达 10 条消息。本地递归服务器（此处为 GW.HOME）使用由 ISP 提供的 DNS 服务器。该服务器依次使用互联网根域名服务器和 gTLD 服务器（用于 COM 和 NET TLD）来查找 EXAMPLE.COM 域名的名称服务器。该名称服务器（此处为 A.IANA-SERVERS.NET）提供主机 EXAMPLE.COM 对应的 IP 地址。所有的递归服务器缓存学习到的任何信息供以后使用

介绍一下TCP三次握手、四次挥手？

正常开启、关闭

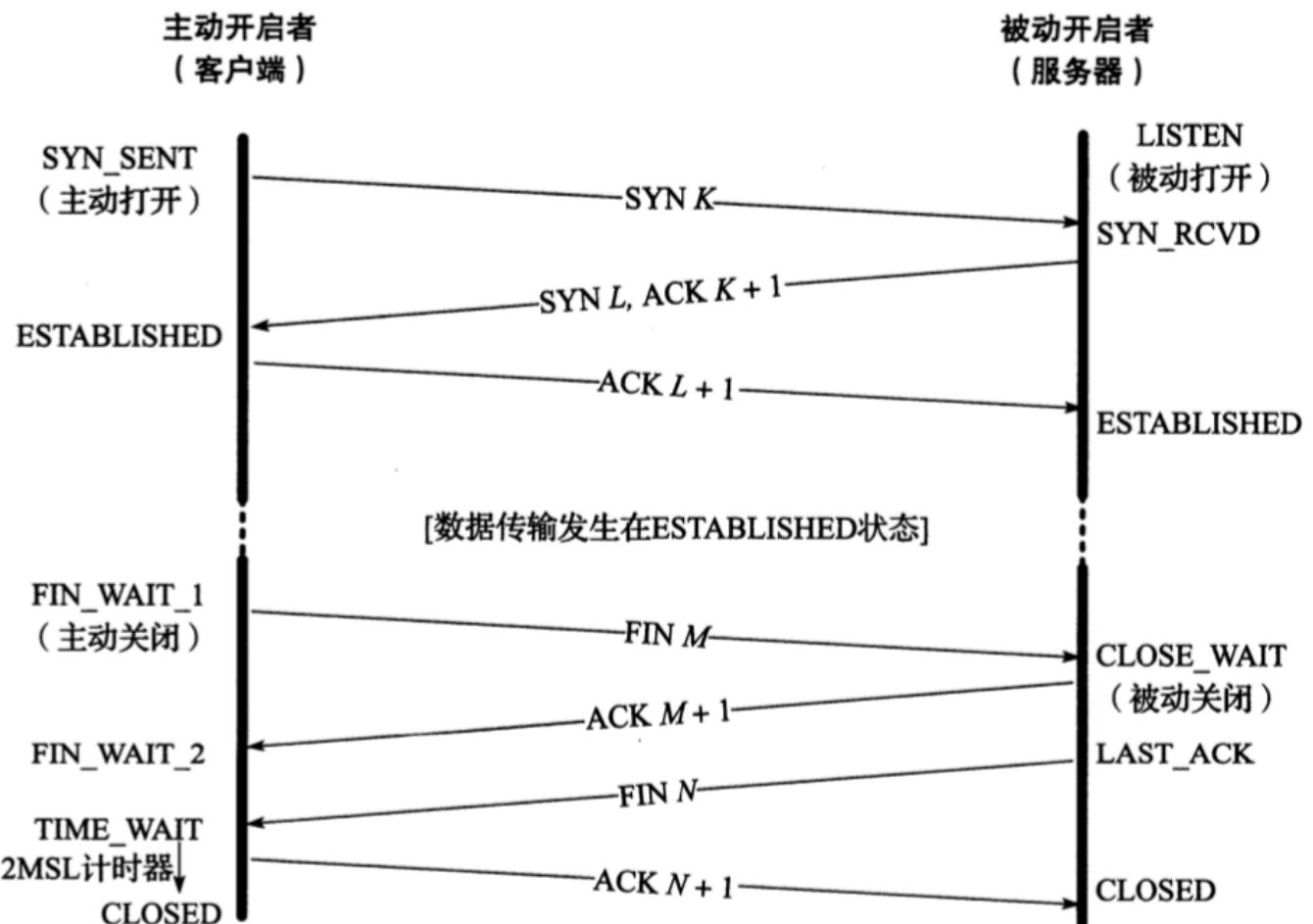


图 13-9 与正常连接的建立与终止相关的 TCP 状态

同时开启、关闭

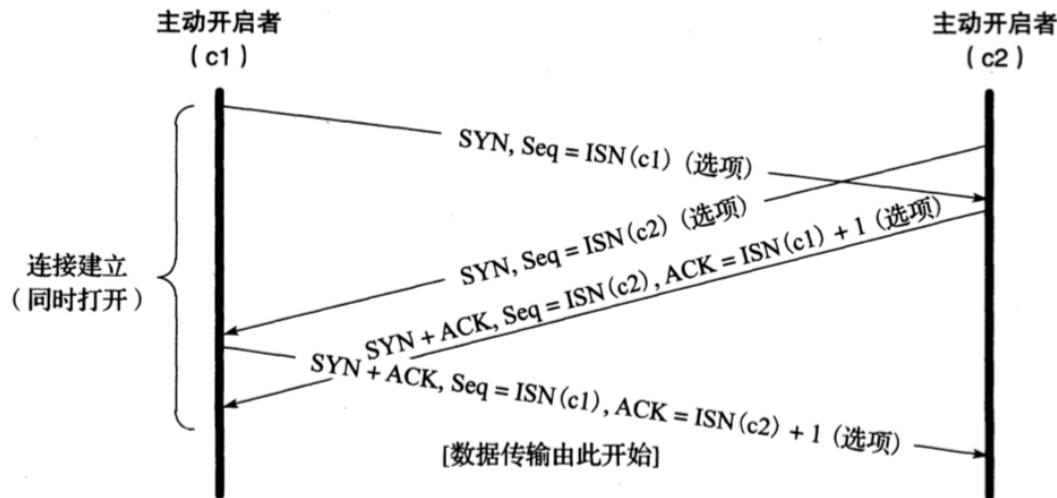


图 13-3 在同时打开中交换的报文段。与正常的连接建立过程相比，需要增加一个报文段。数据包的 SYN 位将置位直到接收到一个 ACK 数据包为止

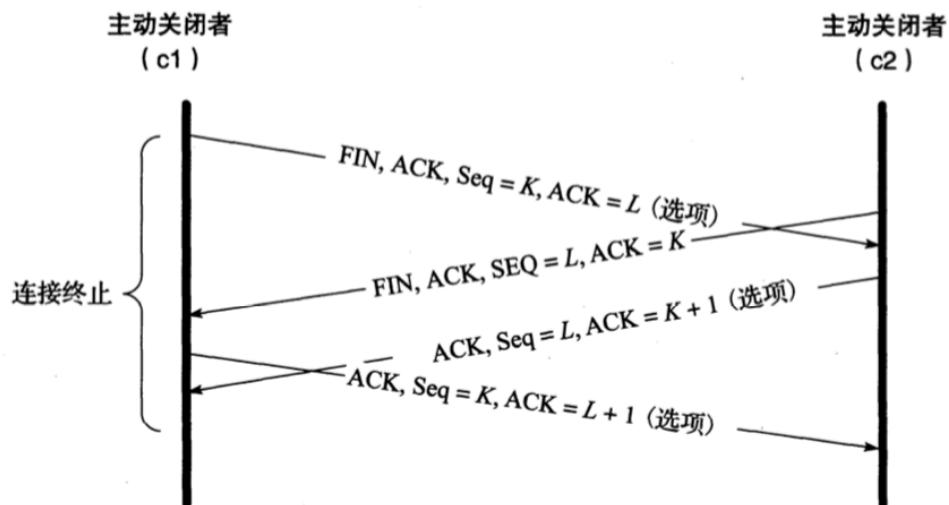


图 13-4 在同时关闭中交换的报文段。与正常关闭相似，只是报文段的顺序是交叉的

同时关闭需要交换与正常关闭相同数量的报文段。两者真正的区别在于报文段序列是交叉的还是顺序的。下文将会介绍 TCP 实现中同时打开与同时关闭操作使用特殊状态这一不常见的方法。

TCP状态转换图

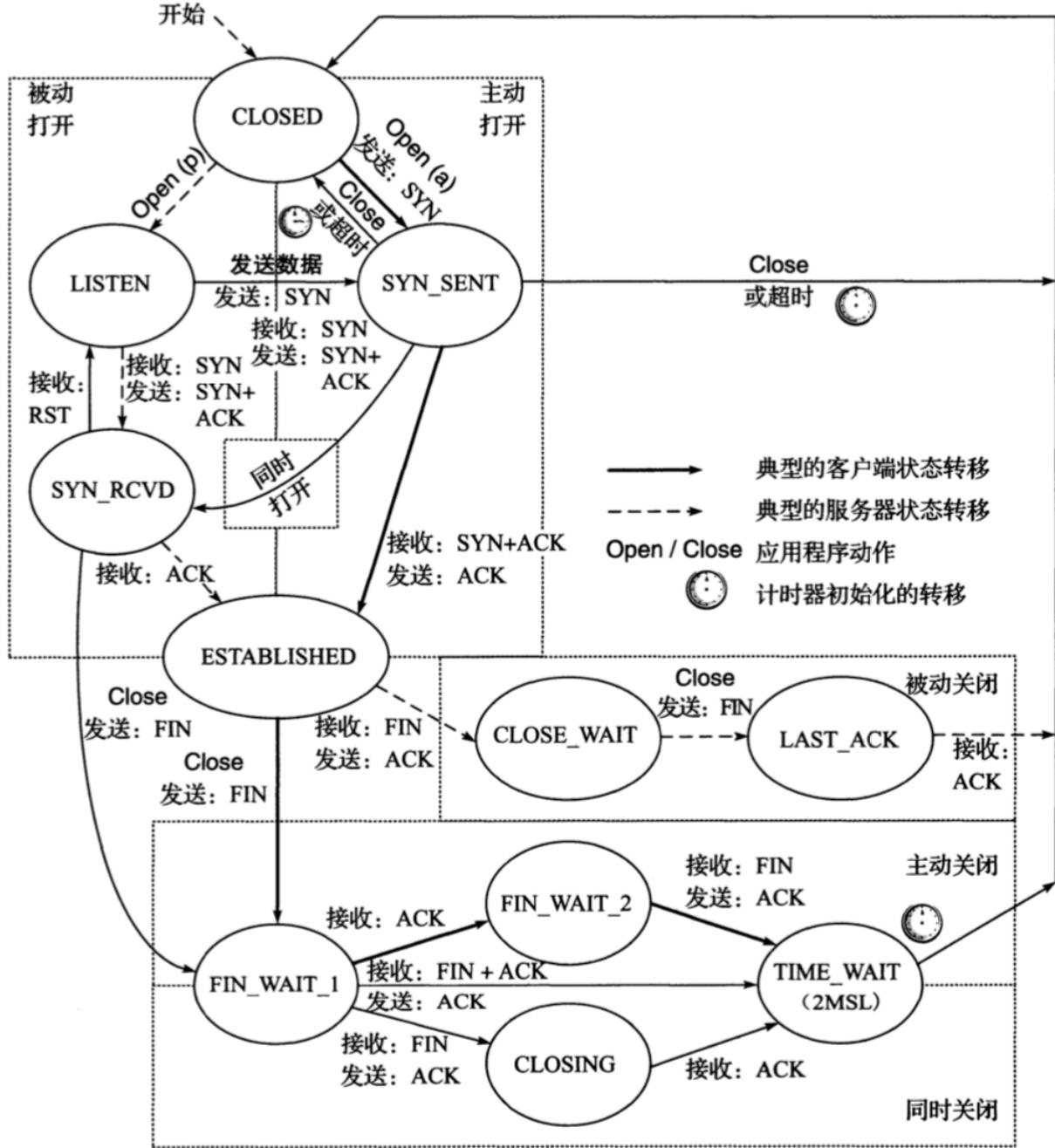


图 13-8 TCP 状态转换图（也称作有限状态机）。箭头表示因报文段传输、接收以及计时器超时而引发的状态转换。粗箭头表示典型的客户端的行为，虚线箭头表示典型的服务器行为。粗体指令（例如 open、close）是应用程序执行的操作

TIME_WAIT (2MSL)

TIME_WAIT 状态时，TCP 将会等待 2 倍于最大生存期 (MSL) 的时间，它代表任何报文在丢弃前在网络中允许的最大存在时间。这个 2MSL 时间能够让 TCP 重新发送最终的 ACK 以避免丢失的情况。重新发送最终的 ACK 并不是重新发送了最后的那个 ACK，而是通信的另一方重新发送了它的 FIN。事实上，TCP 总是重传 FIN 直到它收到一个 ACK。

TCP 拥塞控制

(...)

TCP滑动窗口

(...)

TCP为什么传输安全

(...)

HTTP 头结构?

第一行：请求/响应行

第二行开始：头部字段

HTTP报文头与主体之间通过空一行进行分隔

①请求方法 ②请求URL ③HTTP协议及版本
④报文头
⑤报文体

```
POST /chapter17/user.html HTTP/1.1
Accept: image/jpeg, application/x-ms-application, ... , /**
Referer: http://localhost:8088/chapter17/user/register.html?
code=100&time=123123
Accept-Language: zh-CN
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1;
Content-Type: application/x-www-form-urlencoded
Host: localhost:8088
Content-Length: 112
Connection: Keep-Alive
Cache-Control: no-cache
Cookie: JSESSIONID=24DF2688E37EE4F66D9669D2542AC17B
name=tom&password=1234&realName=tomson
```

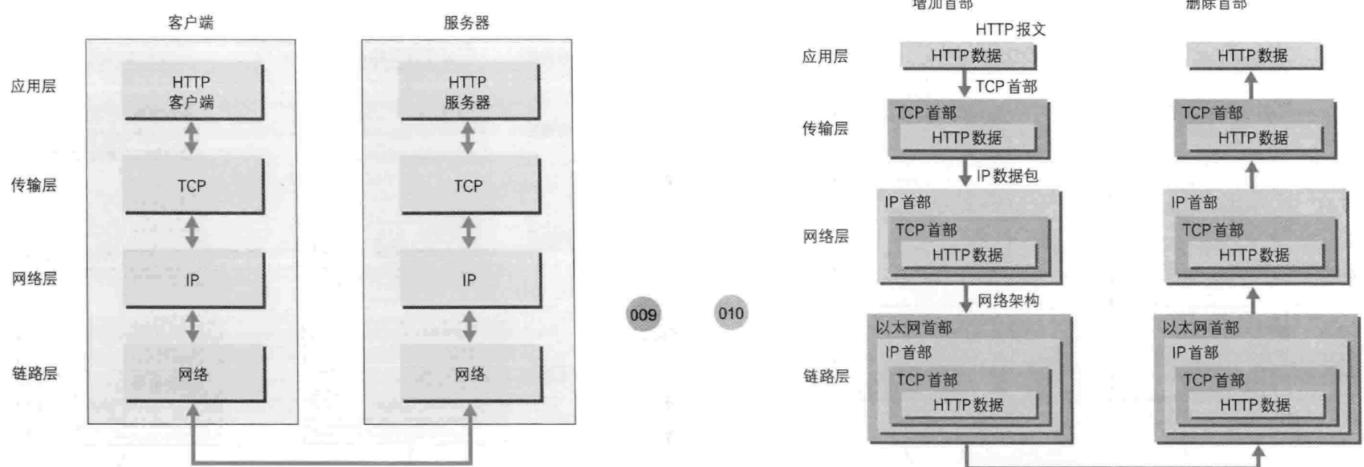
图 15-4 HTTP 请求报文

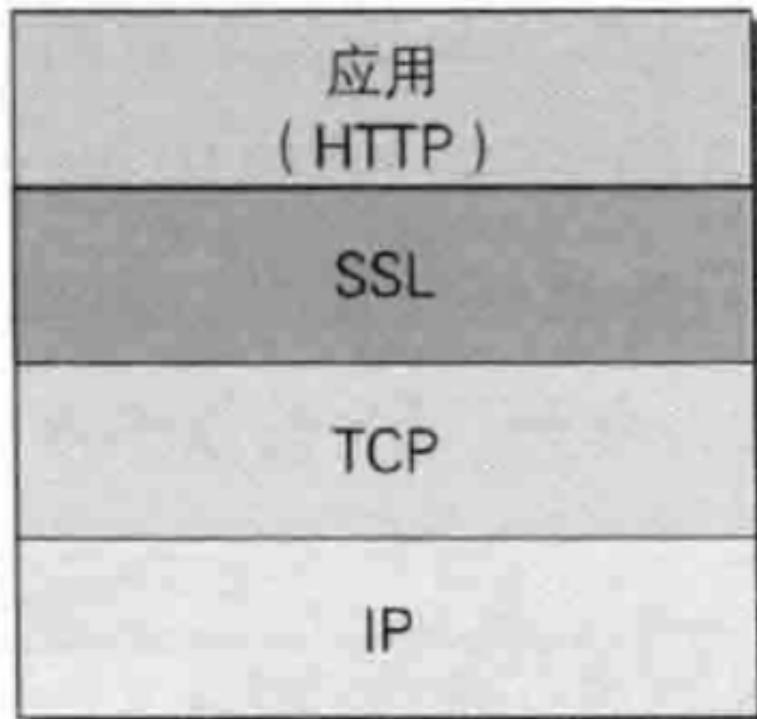


讲一下HTTP协议栈

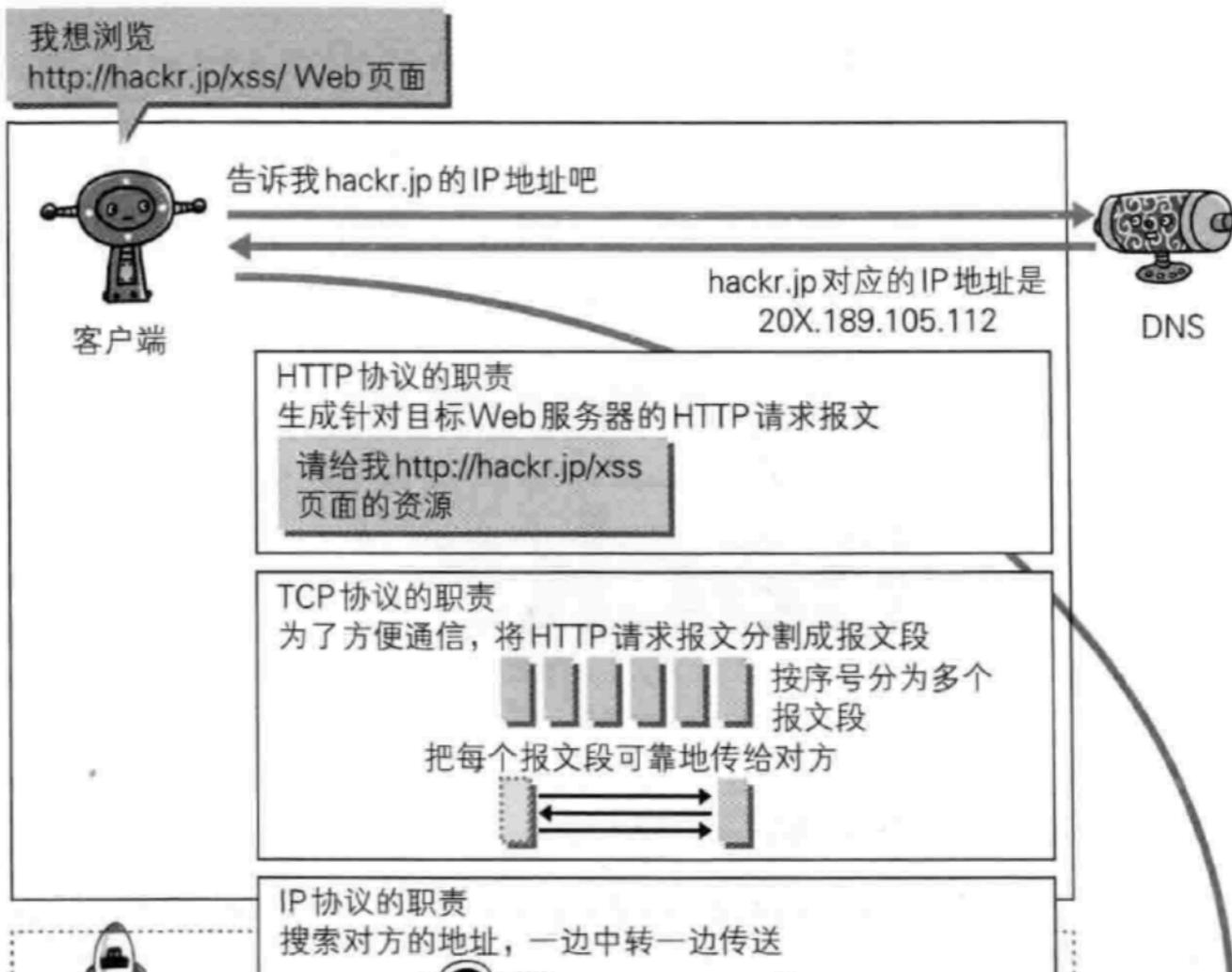
网络层: IP
 传输层: TCP
 会话层: TSL/SSL、SPDY、WebSocket
 应用层: HTTP、DNS

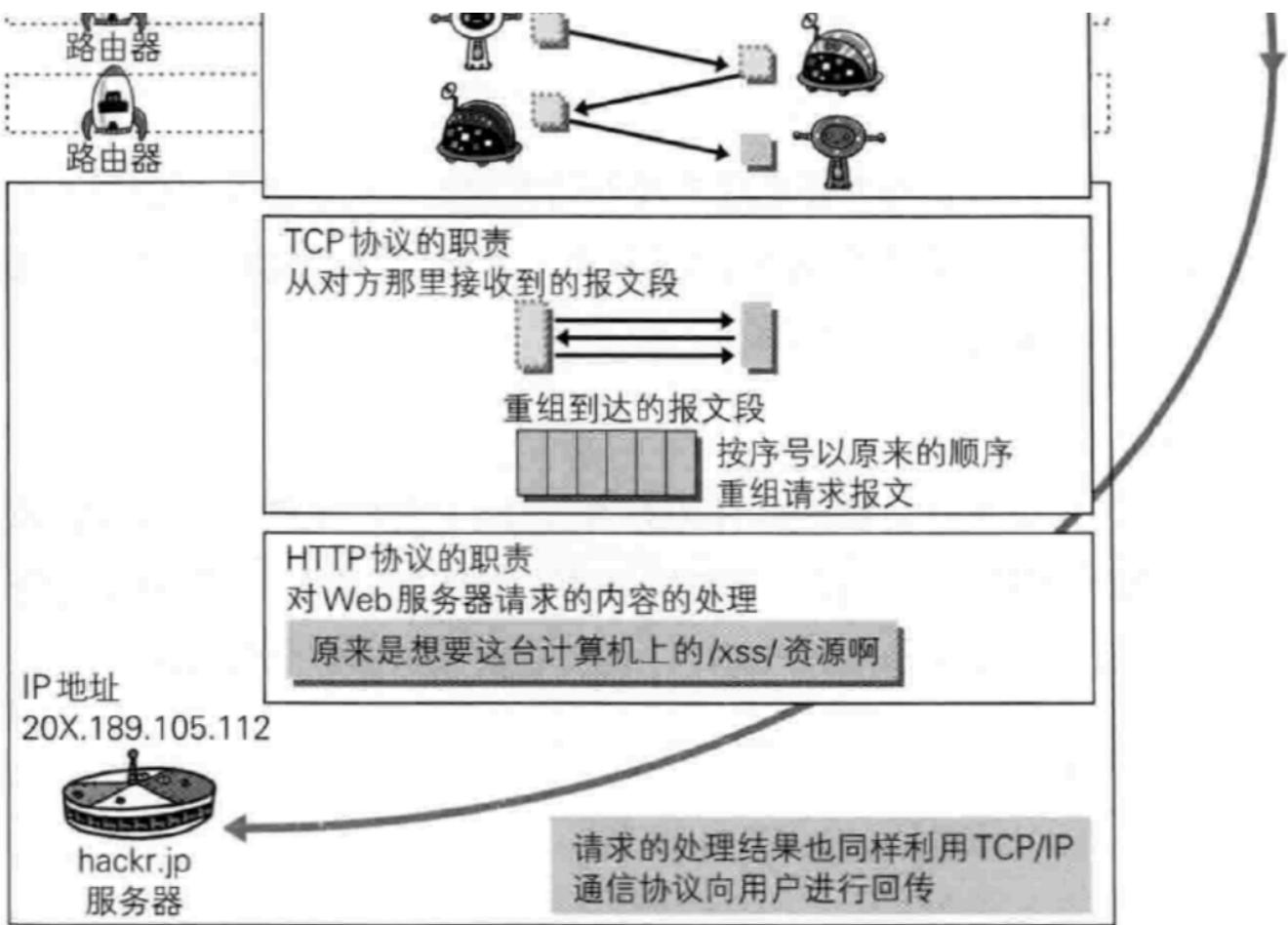
1.3.3 TCP/IP 通信传输流





HTTPS





介绍一下 HTTP: keep-alive?

在 HTTP/1.1 版本的默认连接是持久连接的（默认开启 `Connection: keep-alive`），因此客户端会在持久连接上连续发送请求。当服务器想断开链接时，需要指定 `Connection : close`。

在 HTTP/1.1 之前的版本默认连接时非持久的，因此若想在旧版本的 HTTP 协议上维持持久连接，需要指定 `Connection : keep-alive`。

介绍一下长连接和短连接？

长连接：指在一个 TCP 连接上可以连续发送多个数据包，在 TCP 连接保持期间，如果没有数据包发送，需要双方发检测包以维持此连接，一般需要自己做在线维持。在 HTTP 层上，长连接需要 `Connection : keep-alive`。

短连接：是指通信双方有数据交互时，就建立一个 TCP 连接，数据发送完成后，则断开此 TCP 连接，一般银行都使用短连接。

HTTP/1.0、HTTP/1.1、HTTP/1.x、HTTP/2.0，有什么区别及其改进？

影响HTTP 网络请求的因素

影响一个 HTTP 网络请求的因素主要有两个：带宽和延迟。

带宽：影响网速，在当前网络基础设施的建设上，带宽的影响明显降低。

延迟：

1. **浏览器阻塞 (HOL blocking)**：浏览器会因为一些原因阻塞请求。浏览器对于同一个域名，同时只能有 4 个连接（这个根据浏览器内核不同可能会有所差异），超过浏览器最大连接数限制，后续请求就会被阻塞。
2. **DNS 查询 (DNS Lookup)**：浏览器需要知道目标服务器的 IP 才能建立连接。将域名解析为 IP 的这个系统就是 DNS。这个通常可以利用 DNS 缓存结果来达到减少这个时间的目的。
3. **建立连接 (Initial connection)**：HTTP 是基于 TCP 协议的，浏览器最快也要在第三次握手时才能捎带 HTTP 请求报文，达到真正的建立连接，但是这些连接无法复用会导致每次请求都经历三次握手和慢启动。三次握手在高延迟的场景下影响较明显，慢启动则对文件类大请求影响较大。

HTTP/1.0和HTTP/1.1的一些区别

1. **缓存处理：**在 HTTP/1.0 中主要使用 header 里的 If-Modified-Since、Expires 来做为缓存判断的标准，HTTP/1.1 则引入了更多的缓存控制策略例如 Entity tag, If-Unmodified-Since, If-Match, If-None-Match 等更多可供选择的缓存头来控制缓存策略。
2. **带宽优化及网络连接的使用：**HTTP/1.0 中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP/1.1 则在请求头引入了 range 头域，它允许只请求资源的某个部分，即返回码是 206 (Partial Content)，这样就方便了开发者自由的选择以便于充分利用带宽和连接。
3. **错误通知的管理：**在 HTTP/1.1 中新增了 24 个错误状态响应码，如 409 (Conflict) 表示请求的资源与资源的当前状态发生冲突；410 (Gone) 表示服务器上的某个资源被永久性的删除。
4. **Host 头处理：**在 HTTP/1.0 中认为每台服务器都绑定一个唯一的 IP 地址，因此，请求消息中的 URL 并没有传递主机名 (hostname)。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机 (Multi-homed Web Servers)，并且它们共享一个 IP 地址。**HTTP/1.1 的请求消息和响应消息都应支持 Host 头域，且请求消息中如果没有**

Host头域会报告一个错误（400 Bad Request）。

5. **长连接：**HTTP 1.1支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟，在HTTP1.1中默认开启Connection: keep-alive，一定程度上弥补了HTTP1.0每次请求都要创建连接的缺点。

HTTP/1.x + SPDY

SPDY优化了HTTP1.X的请求延迟，解决了HTTP1.X的安全性，具体如下：

1. **降低延迟：**针对HTTP高延迟的问题，SPDY采取了**多路复用（multiplexing）**。多路复用通过多个请求stream共享一个（单个域名/IP地址）TCP连接的方式，解决了HOL blocking的问题，降低了延迟同时提高了带宽的利用率。
2. **请求优先级（request prioritization）：**多路复用带来一个新的问题是，在连接共享的基础之上有可能会导致关键请求被阻塞。SPDY允许给每个request设置优先级，这样重要的请求就会优先得到响应。比如浏览器加载首页，首页的html内容应该优先展示，之后才是各种静态资源文件，脚本文件等加载，这样可以保证用户能第一时间看到网页内容。
3. **header压缩：**HTTP/1.x的header很多时候都是重复多余的。选择合适的压缩算法可以减小包的大小和数量。
4. **基于HTTPS的加密协议传输：**大大提高了传输数据的可靠性。
5. **服务端推送（server push）：**采用了SPDY的网页，例如我的网页有一个sytle.css的请求，在客户端收到sytle.css数据的同时，服务端会将sytle.js的文件推送给客户端，当客户端再次尝试获取sytle.js时就可以直接从缓存中获取到，不用再发请求了。

SPDY（会话层）位于HTTP之下，TCP和SSL之上，这样可以轻松兼容老版本的HTTP协议（将HTTP1.x的内容封装成一种新的frame格式），同时可以使用已有的SSL功能。



HTTP/2.0 VS SPDY

HTTP/2.0可以说是SPDY的升级版（其实原本也是基于SPDY设计的），但是，HTTP/2.0 跟 SPDY 仍有不同的地方，如下：

1. HTTP/2.0 支持明文 HTTP 传输，而 SPDY 强制使用 HTTPS
2. HTTP/2.0 消息头的压缩算法采用 HPACK，而非 SPDY 采用的 DEFLATE

HTTP/2.0 VS HTTP1.X相比的新特性

1. **新的二进制格式（Binary Format）**，**HTTP/1.x的解析是基于文本**：基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认0和1的组合。基于这种考虑HTTP/2.0的协议解析决定采用二进制格式，实现方便且健壮。
2. **多路复用（MultiPlexing）**：即连接共享，即每一个request都是用作连接共享机制的。一个request对应一个id，这样一个连接上可以有多个request，每个连接的request

可以随机的混杂在一起，接收方可以根据request的 id将request再归属到各自不同的服务端请求里面。

3. **header压缩**：如上文中所言，对前面提到过HTTP/1.x的header带有大量信息，而且每次都要重复发送，HTTP/2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小。
4. **服务端推送（server push）**：同SPDY一样，HTTP/2.0也具有server push功能。

HTTPS与HTTP区别？

HTTP缺点

1. 使用明文通信（不加密），内容可能被窃听
2. 不验证对方身份，有可能遭遇伪装
3. 无法验证报文的完整性，所以有可能遭到篡改

HTTPS（HTTP Secure）

HTTPS = HTTP + 加密 + 认证 + 完整性保护

加密：HTTP通信接口部分用SSL和TSL协议代替

认证：SSL协议提供对方证书认证手段

完整性保护：MD5、SHA-1等散列值方法、PGP验证SSL提供数字签名

对称加密和非对称加密的区别？

对称加密（共享密钥加密）：加密和解密都使用同一个密钥。但如何将密钥安全地送至对方存在问题。由于简单效率较高，但不安全。

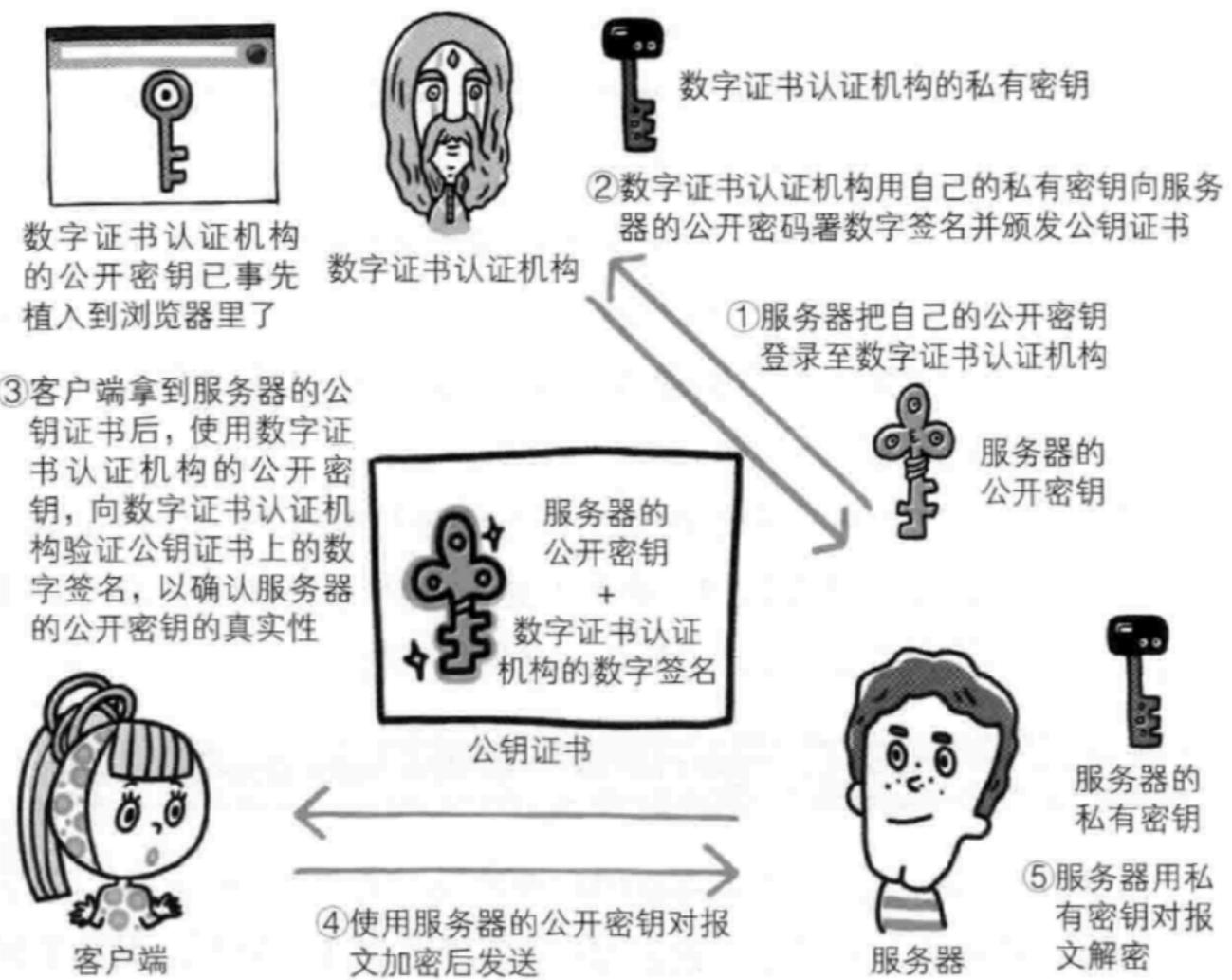
非对称加密（公开密钥加密）：使用一对密钥：一把公开的公钥，一把私有的私钥。报文发送方使用对方公开的密钥对报文进行加密后发送，报文接收方使用私钥对报文进行解密。要想通过密文和公钥进行解密是几乎不可能的，因为解密过程就是在离散对数中进行求值。由于过程复杂，效率不高

HTTPS如何加密通信、认证？

由于非对称加密的效率比对称加密的效率低得多，因此，HTTPS采用混合加密方式：用非对称加密方式对密钥进行加密进行传输，传输完成后，密文部分通过对称加密的方式进行传输。具体步骤为：

1. 开始时，服务器会把自己的公钥登录到数字证书认证机构上。而客户端在安装浏览器的时候，一般也会集成有数字认证机构的公钥。

2. 数字认证机构用私钥对服务器的公钥进行数字签名，并向服务器颁发公钥证书（服务器公钥+数字签名）。
3. 服务器向客户端发送自己的公钥证书后，客户端用数字认证机构集成的公钥验证数字签名，以确定服务器的真实性。
4. 客户端使用服务器的公钥对密文进行加密后向客户端进行传输。
5. 服务器收到客户端发送的密文后用自己的私钥解密密文。

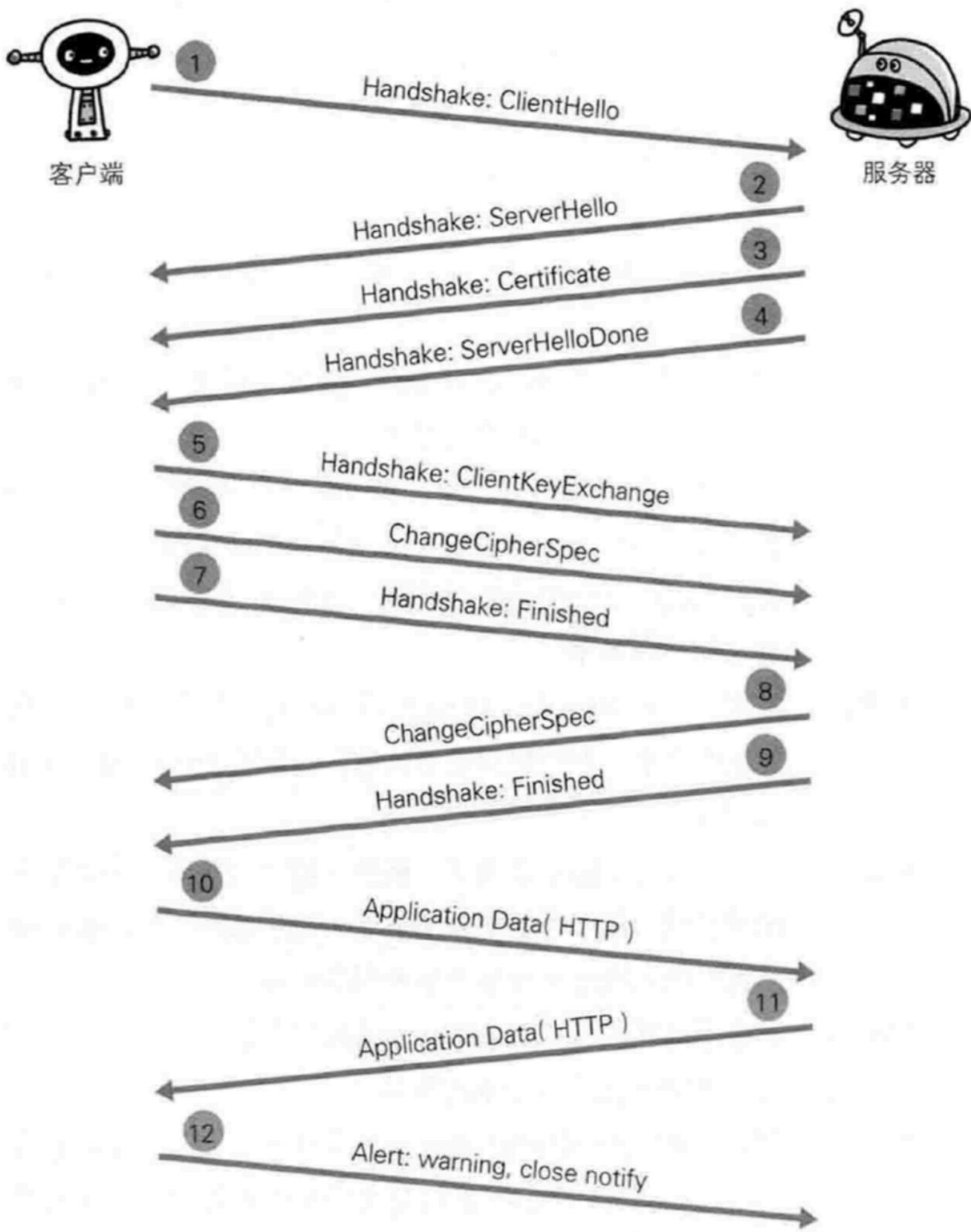


简述HTTPS传输过程？

1. 客户端发送ClientHello报文开始SSL通信。报文中包含客户端支持的指定SSL版本、加密组件列表等。
2. 服务器接收到ClientHello后以ServerHello报文开始SSL通信。报文中包含服务端支持的指定SSL版本和从客户端提供的加密列表中筛选出来的加密组件（子）列表。
3. 服务器再发送Certificate报文，里面包含公钥证书。
4. 服务器最后发送ServerHelloDone报文，表明最初阶段SSL协商完成。
5. 客户端收到ServerHelloDone报文后，发送利用步骤3的公钥证书中的公钥加密后的

ClientKeyExchange报文进行回应。里面包含用Pre-master Secret的随机码。

6. 客户端发送ChangeCipherSpec报文以提示服务器后面用Pre-master Secret密钥进行加密通信。
7. 客户端发送Finish报文，该报文包含至今所有报文的整体校验值。本次握手连接成功与否，要看服务器能否正确解密该报文作为评判标准。
8. 服务器收到Finish报文后也发送ChangeCipherSpec报文。
9. 服务器也发送Finish报文。
10. 服务器和客户端的Finish报文交换完毕后，SSL连接建立完成。此时通信会受到SSL保护，开始进行应用层HTTP协议通信。
11. 进行应用层HTTP协议通信。
12. 最后由客户端断开连接。断开连接时，发送close_notify报文。
13. 之后会发送TCP的FIN报文来关闭TCP通信。



图：HTTPS 通信

介绍一下HTTP方法？

注意HTTP方法大小写敏感，要用大写！

表 2-1: HTTP/1.0 和 HTTP/1.1 支持的方法

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

在这里列举的众多方法中，LINK 和 UNLINK 已被 HTTP/1.1 废弃，不再支持。

解释一下HTTP状态码？

仅列举部分常问状态码：

302: Found 临时性重定向：表示请求资源已经有新的URI了（但不是永久变化，可能后面还会变），希望用户（本次）能使用新的URI访问资源。E.g. 用户把URI保存成书签。

304: Not Modified 没有更新：表示客户端发送附带条件请求（指客户端采用GET方法发送的请求报文中包含If-Match、If-Modified-Since、If-None-Match、If-Range、If-Unmodified-Since中任一首部）时，服务器允许访问资源，但未满足条件的情况。该状态码返回时不包含响应主体部分。

403: Forbidden 服务器拒绝访问：该状态表示服务器理解了本次请求但是拒绝执行该任务，该请求不该重发给服务器。可能是客户端未获得授权等情况。

404: Not Found 未找到资源：表示服务器上无法找到请求的资源。也可以在服务器拒绝访问时不想给出任何理由时使用。

当访问一个web应用，出现403错误，怎么排查可能出现的原因

403状态码分类

403.1: 错误是由于"执行"访问被禁止而造成的，若试图从目录中执行CGI、ISAPI或其他可执行程序，但该目录不允许执行程序时便会出现此种错误。

403.2: 错误是由于"读取"访问被禁止而造成的。导致此错误是由于没有可用的默认网页并且没有对目录启用目录浏览，或者要显示的HTML网页所驻留的目录仅标记为"可执行"或"脚本"权限。

403.3: 错误是由于"写入"访问被禁止而造成的，当试图将文件上载到目录或在目录中修改文件，但该目录不允许"写"访问时就会出现此种错误。

403.4: 错误是由于要求SSL而造成的，您必须在要查看的网页的地址中使用"https"。

403.5: 错误是由于要求使用128位加密算法的Web浏览器而造成的，如果您的浏览器不支持128位加密算法就会出现这个错误，您可以连接微软网站进行浏览器升级。

403.6: 错误是由于IP地址被拒绝而造成的。如果服务器中有不能访问该站点的IP地址列表，并且您使用的IP地址在该列表中时您就会返回这条错误信息。

403.7: 错误是因为要求客户证书，当需要访问的资源要求浏览器拥有服务器能够识别的安全套接字层(SSL)客户证书时会返回此种错误。

403.8: 错误是由于禁止站点访问而造成的，若服务器中有不能访问该站点的DNS名称列表，而您使用的DNS名称在列表中时就会返回此种信息。请注意区别403.6与403.8错误。

403.9: 错误是由于连接的用户过多而造成的，由于Web服务器很忙，因通讯量过多而无法处理请求时便会返回这条错误。

403.10: 错误是由于无效配置而导致的错误，当您试图从目录中执行CGI、ISAPI或其他可执行程序，但该目录不允许执行程序时便会返回这条错误。

403.11: 错误是由于密码更改而导致无权查看页面。

403.12: 错误是由于映射器拒绝访问而造成的。若要查看的网页要求使用有效的客户证书，而您的客户证书映射没有权限访问该Web站点时就会返回映射器拒绝访问的错误。

403.13: 错误是由于需要查看的网页要求使用有效的客户证书而使用的客户证书已经被吊

销，或者无法确定证书是否已吊销造成的。

403.14：错误Web服务器被配置为不列出此目录的内容，拒绝目录列表。

403.15：错误是由于客户访问许可过多而造成的，当服务器超出其客户访问许可限制时会返回此条错误。

403.16：错误是由于客户证书不可信或者无效而造成的。

403.17：错误是由于客户证书已经到期或者尚未生效而造成的。

导致403错误的主要原因

1. 你的IP被列入黑名单。
2. 你在一定时间内过多地访问此网站（一般是用采集程序），被防火墙拒绝访问了。
3. 网站域名解析到了空间，但空间未绑定此域名。
4. 你的网页脚本文件在当前目录下没有执行权限。
5. 在不允许写/创建文件的目录中执行了创建/写文件操作。
6. 以http方式访问需要ssl连接的网址。
7. 浏览器不支持SSL 128时访问SSL 128的连接。
8. 在身份验证的过程中输入了错误的密码。
9. DNS解析错误，手动更改DNS服务器地址。
10. 连接的用户过多，可以过后再试。
11. 服务器繁忙，同一IP地址发送请求过多，遭到服务器智能屏蔽

403解决方法

1、重建dns缓存：

对于一些常规的403 forbidden错误，首先尝试重建DNS缓存，在运行中输入cmd，然后输入ipconfig /flushdns即可。如果不行的话，就需要在hosts文件里把主页解析一下了。

同时，查看是否在网站虚拟目录中添加默认文档，一般默认文档为：

index.html；index.asp；index.php；index.jsp；default.htm；default.asp等。

2、修改文件夹安全属性：

用以下命令修改文件夹安全属性：

```
chcon -R -t httpd_user_content_t
```

所用命令解析：

```
ls -Z -d public_html/
# 显示文件 / 目录的安全语境-Z, --context
Display security context so it fits on most displays. Displays only mode, user, group, security context and file name.-d, --directory
list directory entries instead of contents, and do not dereference symbolic links
chcon -R -t httpd_user_content_t public_html/
# 修改文件 / 目录的安全语境-R, --recursive
change files and directories recursively-t, --type
set type TYPE in the target security context
```

3、关于apache导致的403错误的解决办法:

打开apache的配置文件httpd.conf, 找到这段代码:

```
Options FollowSymLinks
AllowOverride None
Order deny,allow
Deny from all
```

有时候由于配置了php后, 这里的“Deny from all”已经拒绝了一切连接。把该行改成“allow from all”, 修改后的代码如下, 问题解决。

```
Options FollowSymLinks
AllowOverride None
Order deny,allow
Allow from all
```

之所以会出现错误, 是因为大多数的国外主机在配置Apache的时候启用了mod_security, 也就是开启了安全检查, 如果提交的信息中包含select , % , bin等关键字, Apache就会禁止, 并给出403, 404, 500等错误。

4、关于HawkHost空间出现403错误的解决方法:

有的时候在共享服务器上安装了Mod security, 当网址包含有“%”号等其它敏感字符时, 就会被Mod security阻止。

解决HawkHost 403 Forbidden 错误的方法:

在.htaccess文件里添加如下代码:

```
SecFilterEngine Off
SecFilterScanPOST Off
```

直接放在网站的根目录或者程序运行的目录下。

5、关于WordPress导致的403错误解决方法：

对于一些使用WordPress管理程序搭建的博客来说，就需要修改.htaccess文件，在后面添加上如下内容即可，其实就是disable mod_security

```
SecFilterEngine Off  
SecFilterScanPOST Off
```

另外dedecms的可能还需要再加一条，以让默认访问的是index.html文件的DirectoryIndex index.html。

修改.htaccess文件，将文件上传之后，再重新打开之前出现403 Forbidden的URL就没有再出现错误，直接可以打开了。

输入网址后发生的全过程

(...)

hosts文件

其作用就是将一些常用的网址域名与其对应的IP地址建立一个关联“数据库”，当用户在浏览器中输入一个需要登录的网址时，系统会首先自动从Hosts文件中寻找对应的IP地址，一旦找到，系统会立即打开对应网页，如果没有找到，则系统会再将网址提交DNS域名解析服务器进行IP地址的解析。也就是说Hosts的IP解析优先级比DNS要高。

hosts文件作用：

1、加快域名解析

对于要经常访问的网站，我们可以通过在Hosts中配置域名和IP的映射关系，提高域名解析速度。由于有了映射关系，当我们输入域名计算机就能很快解析出IP，而不用请求网络上的DNS服务器。

2、方便局域网用户

在很多单位的局域网中，会有服务器提供给用户使用。但由于局域网中一般很少架设DNS服务器，访问这些服务器时，要输入难记的IP地址。这对不少人来说相当麻烦。可以分别给这些服务器取个容易记住的名字，然后在Hosts中建立IP映射，这样以后访问的时候，只要输入这个服务器的名字就行了。

3、屏蔽网站（域名重定向）

有很多网站不经过用户同意就将各种各样的插件安装到你的计算机中，其中有些说不定就是木马或病毒。对于这些网站我们可以利用Hosts把该网站的域名映射到错误的IP或本地计算机的IP，这样就不用访问了。在WINDOWS系统中，约定 127.0.0.1 为本地计算机的IP地址，0.0.0.0是错误的IP地址。

如果，我们在Hosts中，写入以下内容：

127.0.0.1要屏蔽的网站A的域名

0.0.0.0要屏蔽的网站B的域名

这样，计算机解析域名A和 B时，就解析到本机IP或错误的IP，达到了屏蔽网站A 和B的目的。

4、顺利连接系统

对于Lotus的服务器和一些数据库服务器，在访问时如果直接输入IP地址那是不能访问的，只能输入服务器名才能访问。那么我们配置好Hosts文件，这样输入服务器名就能顺利连接了。

5、虚拟域名

很多时候，网站建设者需要把“软环境”搭建好，再进行上传调试。但类似于邮件服务，则需要使用域名来辅助调试，这时就可以将本地 IP 地址与一个“虚拟域名”做地址指向，就可以达到要求的效果，且无需花费。如：

127.0.0.1 网站域名

之后在浏览器地址栏中输入对应的网站域名即可。

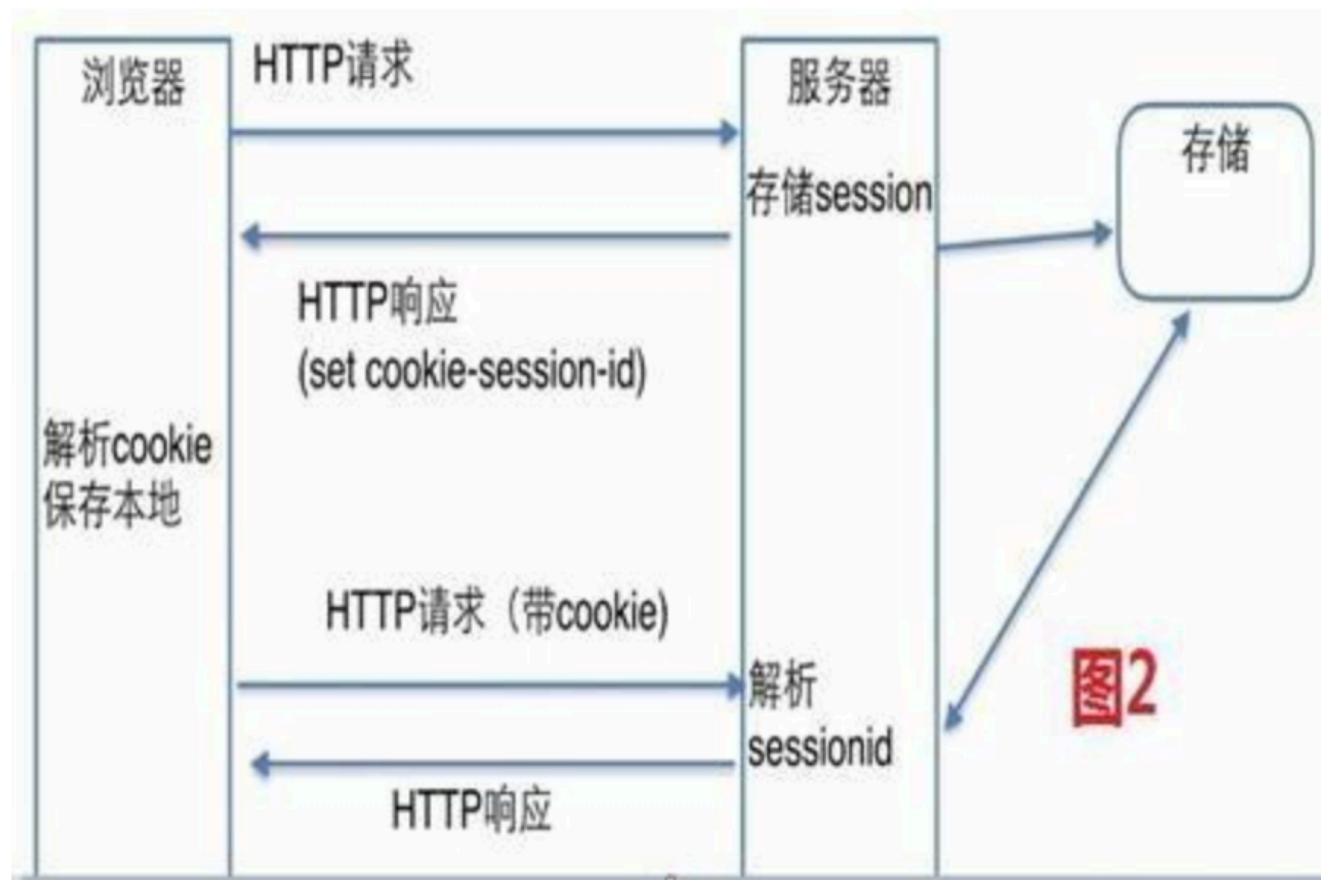
Cookie & Session

cookie：位于客户端上，用来维护用户计算机中的信息，直到用户删除。比如我们在网页上登录某个软件时输入用户名及密码时如果保存为cookie，则每次我们访问的时候就不需要登录网站了。我们可以在浏览器上保存任何文本，而且我们还可以随时随地的去阻止它或者删除。我们同样也可以禁用或者编辑cookie，但是有一点需要注意不要使用cookie来存储一些隐私数据，以防隐私泄露。

session：称为会话信息，位于web服务器上，主要负责访问者与网站之间的交互，当访问浏览器请求http地址时，将传递到web服务器上并与访问信息进行匹配，当关闭网站时就表示会话已经结束，网站无法访问该信息了，所以它无法保存永久数据，我们无法访问以及禁用网站。

区别：

1. cookie以文本文件格式存储在客户端浏览器中，而session存储在服务端。
2. cookie的存储限制了数据量，只允许4KB，而session是无限量的。
3. 我们可以轻松访问cookie值但是我们无法轻松访问session值，因此它更安全。
4. 设置cookie时间可以使cookie过期。但是使用session-destory ()，我们将会销毁会话。



WebSocket

WebSocket即Web浏览器与Web服务器之间的全双工通信标准。

WebSocket特点

1. 支持推送功能：服务端可以主动向客户端推送消息而不用等客户端请求。
2. 减少通信量：WebSocket首部信息很小，通信量小。

WebSoket过程

成功握手确立 WebSocket 连接之后，通信时不再使用 HTTP 的数据帧，而采用 WebSocket 独立的数据帧。

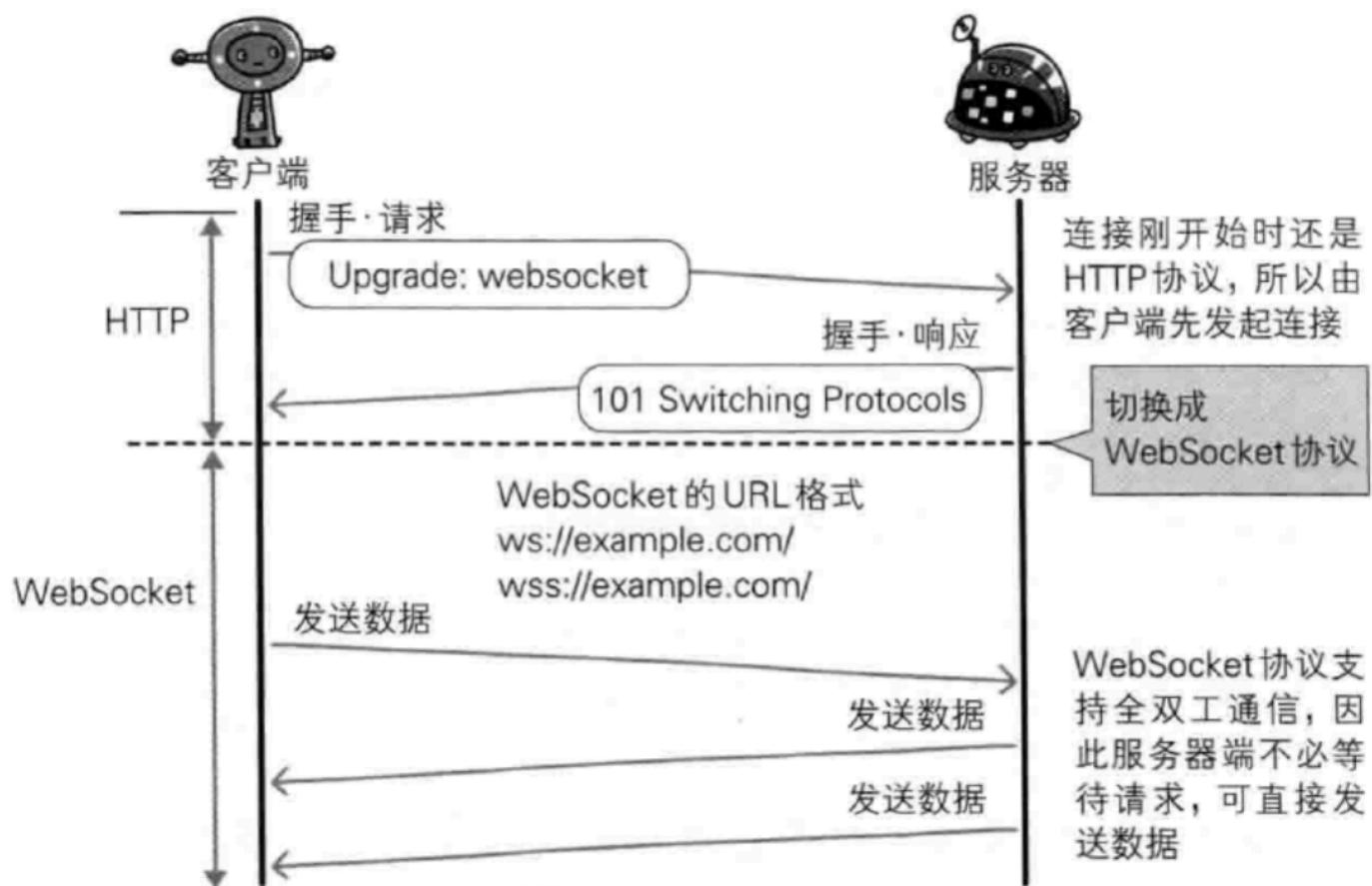


图: WebSocket 通信

RPC

(...)

Linux操作

如何查看网络节点状态？

netstat -a -n -t

- a: 输出所有网络节点，包括处于或者未处于侦听状态的节点
- n: 按照点分十进制（或者十六进制）的形式打印IP地址
- t: 只选择TCP节点

操作系统

设计模式

软件工程

数据结构

算法

大规模数据处理

智力题
