

Java

并发

假设有一家商店卖苹果，现在有两个人（线程）要来买苹果，他们同时进入了商店，但是只有一个人可以在柜台前买苹果（即访问共享资源），另一个人需要等待。如果第一个人（线程A）正在购买苹果（即在执行 `synchronized` 方法/块），而此时第二个人（线程B）也想买苹果，这时候线程B就会进入 `BLOCKED`（阻塞）状态，等待线程A释放锁。

当线程A买完苹果离开柜台（即执行完 `synchronized` 方法/块），此时锁就被释放了。这时候线程B就会被唤醒并重新尝试获取锁进入柜台买苹果（即重新进入 `synchronized` 方法/块）。如果此时锁还是被其它线程占用，线程B就又会进入 `BLOCKED`（阻塞）状态。

在Java中，`wait` 和 `blocked` 都是线程的阻塞状态，但是它们之间还是有一定的区别的。

`wait` 状态是线程主动进入的，即线程执行了 `Object.wait()` 方法，该方法会释放该线程持有的锁，并将该线程置于等待状态，直到其它线程调用该对象的 `notify()` 或 `notifyAll()` 方法，或者指定等待时间到达后，线程才会被唤醒并重新进入锁池（锁等待队列），等待获取锁进入运行状态。

而 `blocked` 状态是线程因为请求获取某个锁而被动阻塞的状态，当一个线程试图获取一个对象锁，但是该锁已经被其它线程所持有时，该线程就会进入 `blocked` 状态，等待锁的释放。当锁被释放后，线程就会重新进入锁池，等待获取锁进入运行状态。

简单来说，`wait` 状态是线程主动释放锁并等待唤醒，而 `blocked` 状态是线程因为等待获取锁而被动阻塞。

具体的代码示例如下：

```
class AppleStore {
    private int appleCount = 10;

    public synchronized void buyApple(String name) {
        while (appleCount <= 0) {
            try {
                System.out.println(name + " 没有买到苹果，等待中...");
                wait(); // 等待并释放锁
            } catch (InterruptedException e) {
```

```

        e.printStackTrace();
    }
}
appleCount--;
System.out.println(name + " 买到了一个苹果，还剩 " + appleCount + "
个苹果。");
notifyAll(); // 唤醒其它等待的线程
}
}

public class Main {
    public static void main(String[] args) {
        AppleStore appleStore = new AppleStore();
        Thread threadA = new Thread(new Runnable() {
            @Override
            public void run() {
                appleStore.buyApple("小明");
            }
        });
        Thread threadB = new Thread(new Runnable() {
            @Override
            public void run() {
                appleStore.buyApple("小红");
            }
        });
        threadA.start(); // 启动线程 A
        threadB.start(); // 启动线程 B
    }
}

```

在上面的代码中，`AppleStore` 类表示商店，有一个共享资源 `appleCount` 表示苹果的数量，`buyApple()` 方法表示买苹果的行为。在 `buyApple()` 方法中，通过 `synchronized` 方法/块来保证线程安全，并且通过 `wait()` 方法在苹果卖完的时候等待并释放锁，防止忙等待。在 `Main` 类中创建了两个线程，分别表示两个人来买苹果，同时启动这两个线程。在这个过程中，线程B会先进入 `BLOCKED`（阻塞）状态，等待线程A释放锁；然后线程A买完苹果离开柜台，锁被释放。

区别：

- `sleep()` 方法没有释放锁，而 `wait()` 方法释放了锁。
- `wait()` 通常被用于线程间交互/通信，`sleep()` 通常被用于暂停执行。

- `wait()` 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 `notify()` 或者 `notifyAll()` 方法。`sleep()` 方法执行完成后，线程会自动苏醒，或者也可以使用 `wait(long timeout)` 超时后线程会自动苏醒。
- `sleep()` 是 `Thread` 类的静态本地方法，`wait()` 则是 `Object` 类的本地方法。

为什么 `wait()` 方法不定义在 `Thread` 中？

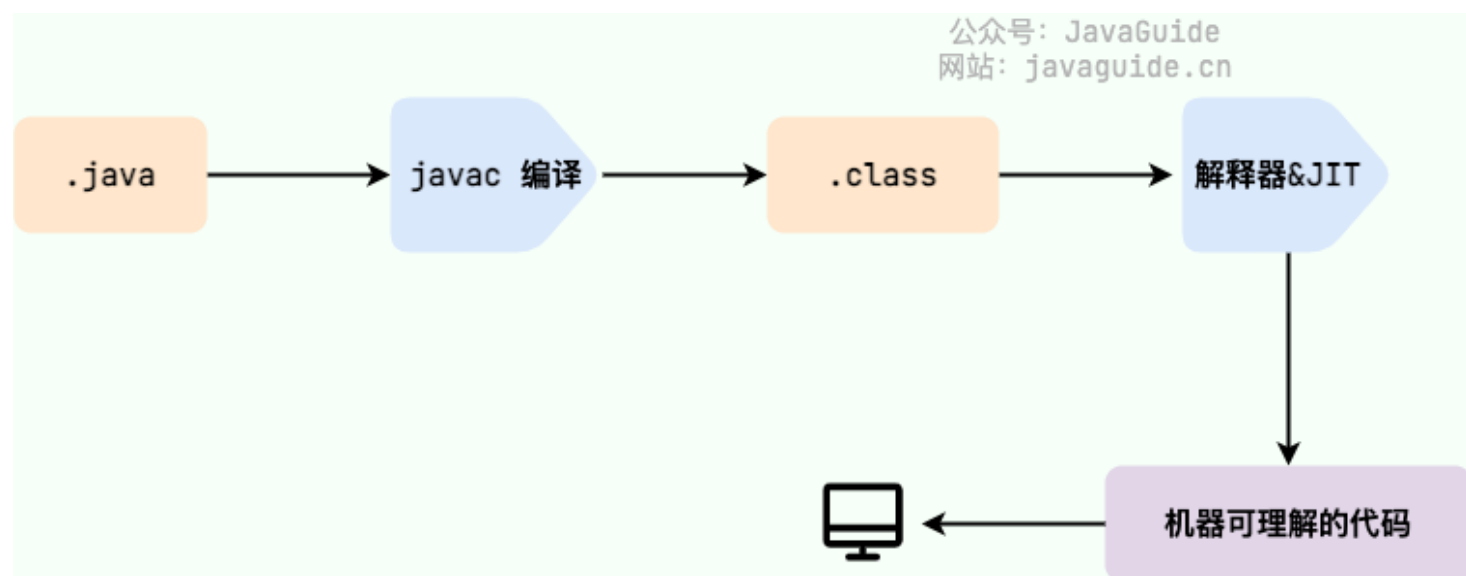
`wait()` 是让获得对象锁的线程实现等待，会自动释放当前线程占有的对象锁。每个对象 (`Object`) 都拥有对象锁，既然要释放当前线程占有的对象锁并让其进入 `WAITING` 状态，自然是要操作对应的对象 (`Object`) 而非当前的线程 (`Thread`)。

类似的问题：为什么 `sleep()` 方法定义在 `Thread` 中？

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

编译与解释并行怎么理解

Java 程序从源代码到运行的过程如下图所示：



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

JVM

程序计数器

程序计数器 (Program Counter Register) 是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。在Java虚拟机的概念模型里，字节码解释器工作时就是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，它是程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

由于Java虚拟机的多线程是通过线程轮流切换、分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器(对于多核处理器来说是一个内核)都只会执行一条线程中的指令。因此，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，各条线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

如果线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址:如果正在执行的是本地 (Native) 方法，这个计数器值则应为空(Undefined)。此内存区域是唯一个在《Java虚拟机规范》中没有规定任何OutOfMemoryError情况的区域。

如果正在执行的是本地 (Native) 方法，这个计数器值则应为空(Undefined):

在恢复线程的过程中，如果线程之前执行了本地方法，那么在恢复时，程序计数器的值将被置为Undefined。这是因为本地方法的执行位置不是由程序计数器来记录的，而是由JVM内部的本地方法栈来管理的。

当线程需要被恢复时，JVM会将该线程的上下文信息恢复到对应的寄存器和内存中。然后，JVM会根据程序计数器的值来确定线程需要从哪里开始执行。如果程序计数器的值为Undefined，则线程将从本地方法返回到Java代码，或者从其他地方继续执行。如果程序计数器的值为非Undefined，则线程将从程序计数器所指示的位置开始执行指令。

如果在本地方法执行期间发生了线程切换，那么当该线程被恢复时，JVM会检查程序计数器的值是否为Undefined。如果程序计数器的值为Undefined，JVM会让线程从本地方法返回到Java代码。具体来说，JVM会执行以下步骤：

1. 恢复线程的上下文信息，包括本地方法栈、堆栈等；
2. 将程序计数器的值设置为Java方法的返回地址；
3. 将线程的状态设置为“返回”状态；
4. 让线程继续执行Java代码，从程序计数器所指示的位置开始执行指令。

String intern()

两种创建字符串的形式

字面量的形式

```
String s = "lmc";
```

执行过程：

首先jvm会在常量池中查找是否已经有了值为“lmc”的字符串对象，如果没有，则在常量池中创建一个值为“lmc”的字符串对象，然后将该字符串对象的引用赋值给s，如果常量池中已经存在值为“lmc”的字符串对象，则直接将该字符串的引用直接赋值给变量s；

使用new的形式

```
String s = new String("lmc");
```

执行过程：

会创建两个对象

- 1、在堆中创建一个包含字符序列“lmc”的字符串对象
- 2、如果字符串常量池中不存在相同的字符串对象，就在字符串常量池中创建一个字符串“lmc”的对象；如果在字符串常量池中存在相同的字符串，则不会创建新的对象，而是直接返回已有的对象引用，最终返回堆中String对象的引用，而不是字符串常量池中对对象的引用。

intern()方法

对于上述new关键字创建的字符串对象，返回的只有堆中的字符串的常量引用，没有字符串常量池中的字符串对象引用，鉴于此，该方法会将这个对象的引用加入到字符串常量池。

该方法首先会检查字符串常量池中是否有该对象的引用，如果存在，则将这个引用返回给变量，否则将引用加入并返回给变量。

https://blog.csdn.net/Lucky_Boy_Luck/article/details/106587260

JDK 1.8版本的字符串常量池中存的是字符串对象，以及字符串常量值

<https://www.cnblogs.com/flyingrun/p/12781257.html>

```

public class StringInter {
    public static void main(String[] args) {
        String s = new String("1");
        s.intern();
        String s2 = "1";
        (1) System.out.println(s == s2); //
        String s3 = new String("a") + new String("bc");
        s3.intern();
        String s4 = "abc";

        (2) System.out.println(s3 == s4); //
    }
}

```

在jdk7/8中，结果为false true

原因：

(1) 执行完第一行代码，会创建两个对象：堆空间的new String ()和字符串常量池（jdk6中字符串常量池在永久代）中的“1”；这个时候字符串常量池中是存在“1”的；所以执行s.intern(); 直接返回“1”的引用。执行 String s2 = "1";由于常量池中已经存在“1”，所以s2直接指向常量池中“1”的引用！

所以，s2的地址是字符串常量池中的“1”，s的地址在堆空间，二者并不相等！结果为false

(2) 这是一个字符串拼接操作，这个“11”是**存在于堆空间中的，字符串常量池中并没有“11”**；在JDK 7/8中，创建一个指向堆空间中new String(“11”)的地址。所以执行完s3.intern(), 此时的字符串常量池中的“11”是引用堆空间的new String(“11”)的地址。当执行String s4 = "11"时，s4会从常量池中找到“引用地址”，所以这样看来，s4 指向“引用地址”，“引用地址”指向堆空间中的new String(“11”)，而s3本来就指向堆空间中的new String(“11”)，二者相等，结果为true!

当执行第一行代码时，字符串常量池中会有a、bc的引用，而s3是指向堆中对象abc的引用

当执行第二行代码时，jvm会检测字符串常量池中是否有对象“abc”的引用，如果没有，则添加此对象的引用到常量池中去；否则，直接返回该引用。显然，这里是没的，所以，常量池中会存一个指向堆中对象“abc”的引用。

执行第三行代码时，jvm还是会检测常量池中是否有指向对象“abc”的引用。显然，这里是有的。所以，会把指向堆中对象“abc”的引用赋值给s2。也就是说，s2也会指向堆。

HashTable 和 HashMap 的区别

HashTable和HashMap都是用来存储键值对的数据结构，但它们在实现细节和用法上有所不同。

主要区别如下：

1. 线程安全性：HashTable是线程安全的，HashMap是非线程安全的。
2. 效率：HashTable因为保证线程安全，需要进行加锁操作，因此效率相对较低。而HashMap在单线程情况下效率更高，但在多线程环境下需要使用并发控制机制，否则会出现数据不一致的问题。
3. 键和值：HashTable和HashMap都使用键值对存储数据，但是HashTable的键和值都不能为null，而HashMap的键和值都可以为null。
4. 迭代器：HashTable的迭代器是通过Enumeration实现的，而HashMap的迭代器是通过Iterator实现的。

线程是否安全： HashMap 是非线程安全的， Hashtable 是线程安全的,因为 Hashtable 内部的方法基本都经过 synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；

效率： 因为线程安全的问题， HashMap 要比 Hashtable 效率高一点。另外， Hashtable 基本被淘汰，不要在代码中使用它；

对 Null key 和 Null value 的支持： HashMap 可以存储 null 的 key 和 value，但 null 作为键只能有一个，null 作为值可以有多个；Hashtable 不允许有 null 键和 null 值，否则会抛出 NullPointerException。

初始容量大小和每次扩充容量大小的不同： ① 创建时如果不指定容量初始值， Hashtable 默认的初始大小为 11，之后每次扩充，容量变为原来的 $2n+1$ 。 HashMap 默认的初始化大小为 16。之后每次扩充，容量变为原来的 2 倍。② 创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为 2 的幂次方大小（HashMap 中的 tableSizeFor() 方法保证，下面给出了源代码）。也就是说 HashMap 总是使用 2 的幂作为哈希表的大小，后面会介绍到为什么是 2 的幂次方。

底层数据结构： JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树），以减少搜索时间（后文中我会结合源码对这一过程进行分析）。 Hashtable 没有这样的机制。

综上所述，如果需要在多线程环境下使用键值对存储数据，可以考虑使用HashTable；如果在单线程环境下需要高效地存储键值对数据，则可以使用HashMap。

ArrayDeque 与 LinkedList 的区别

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

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

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

HashMap HashSet区别

`HashMap`和`HashSet`都是Java中的集合类，但是它们的实现和用途有所不同。

1. `HashMap`： `HashMap`是一个键值对应的集合，它允许使用`null`作为`key`和`value`。它通过一个哈希函数将`key`映射为一个索引，然后将`value`存储在该索引对应的位置。
`HashMap`的访问速度很快，因为它使用哈希表的数据结构，但是它不保证元素的顺序。
2. `HashSet`： `HashSet`是一个不允许重复元素的集合，它允许使用`null`元素。它是通过一个哈希函数将元素映射为一个索引，并将元素存储在该索引对应的位置。与`HashMap`不同的是，`HashSet`中只存储元素而没有键值对应关系。

因此，`HashMap`适合需要键值对应关系的场景，而`HashSet`适合需要存储不重复元素的场景。

serialVersionUID作用

`serialVersionUID` 是 Java 中用于标识序列化类版本的唯一标识符。当一个可序列化的对象被序列化和反序列化时，Java 运行时环境会使用 `serialVersionUID` 来确定序列化的类与反序列化的类是否兼容，如果不兼容则会抛出 `InvalidClassException` 异常。

因此，`serialVersionUID` 的作用是确保在不同的 Java 虚拟机之间，不同版本的类之间进行序列化和反序列化操作时，能够正确地匹配类的版本。它是一个编译器自动生成的 `static final long` 类型的成员变量，可以通过显式声明来指定。

注意，当一个类被修改时，如果没有显式地指定 `serialVersionUID`，则编译器会自动生成一个基于类的成员变量、方法和构造函数等元素生成的默认 `serialVersionUID`。因此，当类的成员变量发生变化时，生成的 `serialVersionUID` 也会随之改变，这可能导致序列化版本的不兼容性。为了避免这种情况，可以显式指定 `serialVersionUID`，或使用其他序列化框架（如JSON）来代替Java的默认序列化机制。

ArrayList扩容机制

先来看 `add` 方法

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

注意：JDK11 移除了 `ensureCapacityInternal()` 和 `ensureExplicitCapacity()` 方法

再看看 `ensureCapacityInternal()` 方法

(JDK7) 可以看到 `add` 方法 首先调用了 `ensureCapacityInternal(size + 1)`

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

    ensureExplicitCapacity(minCapacity);
}
```

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

此处和后续JDK8 代码格式化略有不同，核心代码基本一样。

ensureExplicitCapacity() 方法

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

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

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

我们来仔细分析一下：

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

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

grow() 方法

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

/**
 * ArrayList扩容的核心方法。
 */
private void grow(int minCapacity) {
    // oldCapacity为旧容量，newCapacity为新容量
    int oldCapacity = elementData.length;
    //将oldCapacity 右移一位，其效果相当于oldCapacity / 2，
```

//我们知道位运算的速度远远快于整除运算，整句运算式的结果就是将新容量更新为旧容量的1.5倍，

```
int newCapacity = oldCapacity + (oldCapacity >> 1);
```

//然后检查新容量是否大于最小需要容量，若还是小于最小需要容量，那么就把最小需要容量当作数组的新容量，

```
if (newCapacity - minCapacity < 0)
    newCapacity = minCapacity;
```

// 如果新容量大于 MAX_ARRAY_SIZE,进入(执行) `hugeCapacity()` 方法来比较 minCapacity 和 MAX_ARRAY_SIZE,

//如果minCapacity大于最大容量，则新容量则为`Integer.MAX_VALUE`，否则，新容量大小则为 MAX_ARRAY_SIZE 即为 `Integer.MAX_VALUE - 8`。

```
if (newCapacity - MAX_ARRAY_SIZE > 0)
    newCapacity = hugeCapacity(minCapacity);
// minCapacity is usually close to size, so this is a win:
elementData = Arrays.copyOf(elementData, newCapacity);
}
```

`int newCapacity = oldCapacity + (oldCapacity >> 1)`,所以 ArrayList 每次扩容之后容量都会变为原来的 1.5 倍左右 (oldCapacity 为偶数就是 1.5 倍，否则是 1.5 倍左右)！奇偶不同，比如：10+10/2 = 15, 33+33/2=49。如果是奇数的话会丢掉小数。

ThreadLocal

`ThreadLocal` 是一个 Java 类，它允许在每个线程中存储和获取变量的副本，以避免多个线程之间出现竞争条件。

下面是一个简单的示例，说明如何使用 `ThreadLocal` 在多线程环境下存储和获取变量的值：

```
public class ThreadLocalExample {
    public static void main(String[] args) {
        ThreadLocal<Integer> threadLocal = new ThreadLocal<>();

        Runnable runnable = () -> {
            // 获取当前线程的 ID
            long threadId = Thread.currentThread().getId();

            // 将 threadId 作为值存储到 threadLocal 变量中
            threadLocal.set((int) threadId);

            // 获取存储在 threadLocal 中的值并输出
            System.out.println("Thread " + threadId + " : " +
                threadLocal.get());
        };
    }
}
```

```
// 创建 3 个线程并运行
new Thread(runnable).start();
new Thread(runnable).start();
new Thread(runnable).start();
}
}
```

在这个示例中，我们创建了一个 `ThreadLocal` 对象，它的泛型类型是 `Integer`。然后我们定义了一个 `Runnable`，它获取当前线程的 ID，将该 ID 作为值存储到 `threadLocal` 变量中，并获取存储在 `threadLocal` 中的值并输出。

我们创建了 3 个线程并运行，每个线程都会执行 `runnable`。由于每个线程都有自己的 `threadLocal` 实例，因此它们各自存储和获取自己的 ID 值，而不会相互干扰。

输出结果可能如下：

```
Thread 12 : 12
Thread 11 : 11
Thread 10 : 10
```

可以看到，每个线程都正确地存储了自己的 ID，并且获取了正确的值。这个示例说明了 `ThreadLocal` 的基本用法。

从上面 `Thread` 类源代码可以看出 `Thread` 类中有一个 `threadLocals` 和一个 `inheritableThreadLocals` 变量，它们都是 `ThreadLocalMap` 类型的变量，我们可以把 `ThreadLocalMap` 理解为 `ThreadLocal` 类实现的定制化的 `HashMap`。默认情况下这两个变量都是 `null`，只有当前线程调用 `ThreadLocal` 类的 `set` 或 `get` 方法时才创建它们，实际上调用这两个方法的时候，我们调用的是 `ThreadLocalMap` 类对应的 `get()`、`set()` 方法。最终的变量是放在了当前线程的 `ThreadLocalMap` 中，并不是存在 `ThreadLocal` 上，`ThreadLocal` 可以理解为只是 `ThreadLocalMap` 的封装，传递了变量值。`ThreadLocal` 类中可以通过 `Thread.currentThread()` 获取到当前线程对象后，直接通过 `getMap(Thread t)` 可以访问到该线程的 `ThreadLocalMap` 对象。

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

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

Future

`Future` 是Java中用于表示异步计算结果的类，它允许我们在一个线程中提交一个耗时的计算任务，然后在另一个线程中等待这个任务的计算结果。

在Java中，`Future` 类只是一个泛型接口，位于 `java.util.concurrent` 包下，其中定义了5个方法，主要包括下面这4个功能：

- 取消任务；
- 判断任务是否被取消；
- 判断任务是否已经执行完成；
- 获取任务执行结果。

`Future` 接口的主要方法包括：

- `get()` 方法：获取计算结果，如果计算还未完成，会阻塞当前线程直到计算完成。
- `get(long timeout, TimeUnit unit)` 方法：获取计算结果，但是如果计算还未完成，最多等待指定的时间，超时则抛出异常。
- `isDone()` 方法：判断计算是否已经完成。
- `cancel(boolean mayInterruptIfRunning)` 方法：取消计算任务，如果任务已经启动则根据参数决定是否中断执行。

通常情况下，我们会使用 `ExecutorService` 来提交一个 `Callable` 对象到线程池中，并获得一个 `Future` 对象来跟踪这个异步计算的结果。当计算完成后，我们可以调用 `Future` 对象的 `get` 方法来获取计算结果，如果计算还未完成，则会阻塞当前线程直到计算完成。

通过使用 `Future` 类，我们可以方便地实现异步计算和并发编程，从而提高程序的性能和效率。

JVM 垃圾回收

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

Java 堆是垃圾收集器管理的主要区域，因此也被称作 **GC 堆 (Garbage Collected Heap)** 。

从垃圾回收的角度来说，由于现在收集器基本都采用分代垃圾收集算法，所以Java堆被划分为了几个不同的区域，这样我们就可以根据各个区域的特点选择合适的垃圾收集算法。在JDK7版本及JDK7版本之前，堆内存被通常分为下面三部分：新生代（Eden区、两个Survivor区S0和S1）、老年代、永久代。JDK8版本之后PermGen(永久)已被Metaspace(元空间)取代，元空间使用的是直接内存。

内存分配和回收原则

1. 对象优先在 Eden 区分配

大多数情况下，对象在新生代中 Eden 区分配。当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC。若可以在 Survivor 空间分配则分配。GC 期间虚拟机又发现无法存入 Survivor 空间，所以只好通过 **分配担保机制**（就是当在新生代无法分配内存的时候，把新生代的对象转移到老年代，然后把新对象放入腾空的新生代）把新生代的对象提前转移到老年代中去，老年代上的空间足够存放 `allocation1`，所以不会出现 Full GC。执行 Minor GC 后，后面分配的对象如果能够存在 Eden 区的话，还是会在 Eden 区分配内存。

2. 大对象直接进入老年代

大对象就是需要大量连续内存空间的对象（比如：字符串、数组）。大对象直接进入老年代主要是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。

3. 长期存活的对象将进入老年代

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

大部分情况，对象都会首先在 Eden 区域分配。如果对象在 Eden 出生并经过第一次 Minor GC 后仍然能够存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间（s0 或者 s1）中，并将对象年龄设为 1（Eden 区->Survivor 区后对象的初始年龄变为 1）。

对象在 Survivor 中每熬过一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

4. 主要进行 gc 的区域

GC 其实准确分类只有两大种：

部分收集 (Partial GC)：

- 新生代收集 (Minor GC / Young GC)：只对新生代进行垃圾收集；
- 老年代收集 (Major GC / Old GC)：只对老年代进行垃圾收集。需要注意的是 Major GC 在有的语境中也用于指代整堆收集；
- 混合收集 (Mixed GC)：对整个新生代和部分老年代进行垃圾收集。

整堆收集 (Full GC)：收集整个 Java 堆和方法区。

5. 空间分配担保

空间分配担保是为了确保在 Minor GC 之前老年代本身还有容纳新生代所有对象的剩余空间。

JDK 6 Update 24 之前，在发生 Minor GC 之前，虚拟机必须先检查老年代**最大可用的连续空间**是否大于新生代所有对象总空间，如果这个条件成立，那这一次 Minor GC 可以确保是安全的。如果不成立，则虚拟机会先查看

`-XX:HandlePromotionFailure` 参数的设置值是否允许担保失败(Handle Promotion Failure);如果允许，那会继续检查老年代**最大可用的连续空间**是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试进行一次 Minor GC，尽管这次 Minor GC 是有风险的;如果小于，或者 `-XX: HandlePromotionFailure` 设置不允许冒险，那这时就要改为进行一次 Full GC。

JDK 6 Update 24 之后的规则变为只要老年代的连续空间大于新生代对象总大小或者历次晋升的平均大小，就会进行 Minor GC，否则将进行 Full GC。

垃圾收集器

1. Serial 收集器

只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集工作的时候必须暂停其他所有的工作线程（"**Stop The World**"），直到它收集结束。**新生代采用标记-复制算法，老年代采用标记-整理算法。**

优点：简单而高效（与其他收集器的单线程相比）。Serial 收集器由于没有线程交互的开销，自然可以获得很高的单线程收集效率。Serial 收集器对于运行在 Client 模式下的虚拟机来说是个不错的选择。

2. ParNew 收集器

ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和 Serial 收集器完全一样。**新生代采用标记-复制算法，老年代采用标记-整理算法。**它是许多运行在 Server 模式下的虚拟机的首要选择，除了 Serial 收集器外，只有它能与 CMS 收集器（真正意义上的并发收集器，后面会介绍到）配合工作。

3. Parallel Scavenge 收集器

Parallel Scavenge 收集器关注点是吞吐量（高效率的利用 CPU）。CMS 等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）。所谓吞吐量就是 CPU 中用于运行用户代码的时间与 CPU 总消耗时间的比值。Parallel Scavenge 收集器提供了很多参数供用户找到最合适的停顿时间或最大吞吐量，如果对于收集器运作不太了解，手工优化存在困难的时候，使用 Parallel Scavenge 收集器配合自适应调节策略，把内存管理优化交给虚拟机去完成也是一个不错的选择。新生代采用**标记-复制算法，老年代采用标记-整理算法。**

4. Serial Old 收集器

Serial 收集器的老年代版本，它同样是一个单线程收集器。作为 CMS 收集器的后备方案。

5. Parallel Old 收集器

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

6. CMS收集器

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

7. G1收集器

int和Integer区别

两者的区别主要体现在以下几个方面：

- 1、数据类型不同：int 是基础数据类型，而 Integer 是包装数据类型；
- 2、默认值不同：int 的默认值是 0，而 Integer 的默认值是 null；
- 3、内存中存储的方式不同：int 在内存中直接存储的是数据值，而 Integer 实际存储的是对象引用，当 new 一个 Integer 时实际上是生成一个指针指向此对象；
- 4、实例化方式不同：Integer 必须实例化才可以使用，而 int 不需要；
- 5、变量的比较方式不同：int 可以使用 == 来对比两个变量是否相等，而 Integer 一定要使用 equals 来比较两个变量是否相等。

同步锁synchronized

synchronized的作用是能够保证同一时间只有一个线程来运行这块代码，达到并发效果

一个线程访问一个对象中的synchronized(this)同步代码块时，其他试图访问该对象的线程将被阻塞

当两个并发线程(thread1和thread2)访问同一个对象(syncThread)中的synchronized代码块时，在同一时刻只能有一个线程得到执行，另一个线程受阻塞，必须等待当前线程执行完这个代码块以后才能执行该代码块。Thread1和thread2是互斥的，因为在执行synchronized代码块时会锁定当前的对象，只有执行完该代码块才能释放该对象锁，下一个线程才能执行并锁定该对象。

指令重排原因

计算机内存操作速度远慢于CPU运行速度，所以就造成CPU空置，为了提高CPU利用率，虚拟机会按照自己的一些规则会跳过执行慢的代码，去执行快的代码（即对代码重新排序），从而提升jvm的整体性能。

基本类型和包装类型的区别

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

finalize()方法

finalize()是Object中的方法，当垃圾回收器将要回收对象所占内存时，该方法被调用，即当一个对象被虚拟机宣告死亡时会先调用它的finalize()方法，当然可以重写finalize()方法，将对象重新引用到"GC Roots"链上。

finalize()只会在对象内存回收前被调用一次，但是不保证里面的方法会被执行完。如果执行比较慢，还没来得及复活，就被杀死回收了。

泛型和通配符

包含类型参数的类型，作用就是在编译的时候能够检查类型安全，并且所有的强制转换都是自动和隐式的。

通配符

常用的K、V、T、E、？

- `?` 表示不确定的 java 类型
- `T (type)` 表示具体的一个java类型
- `K V (key value)` 分别代表java键值中的Key Value
- `E (element)` 代表Element

无界通配符 “?”

可以指定任意的类型，没有任何限制作用。

上届通配符 <? extend E>

特征：用 extend 关键字声明，表示参数化的类型可能是所指定的类型，或者是此类型的子类。

好处：

- 如果传入的类型不是 E 或者 E 的子类，编译不成功
- 泛型中可以使用 E 的方法，要不然还得强转成 E 才能使用

下届通配符 <? supper E>

特征：用 supper 关键字声明，表示参数化的类型可能是所指定的类型，或者是此类型的父类型，直至 Object。

注意：在类型参数中使用 super 表示这个泛型中的参数必须是 E 或者 E 的父类。

? 和 T 的区别

? 和 T 都表示不确定的类型，区别在于我们可以对 T 进行操作，但是对 ? 不行，比如如下这种：

```
// 可以
T t = operate();
// 不可以
? car = operate();
1234
```

区别1：可以通过 T 保证参数的一致性

泛型方法的定义：

```
interface MyGeneric {
    // 通过 T 来 确保 泛型参数的一致性
    <T> void testT(List<T> dest, List<T> src);
    //通配符是 不确定的，所以这个方法不能保证两个 List 具有相同的元素类型
    void testNon(List<?> dest, List<?> src);
}

class GMapperGeneric<E> implements MyGeneric {
    @Override
    public <T> void testT(List<T> dest, List<T> src) {}
    @Override
    public void testNon(List<?> dest, List<?> src) {}
}

@Test
public void test() {
    GMapperGeneric<String> gMapperGeneric = new GMapperGeneric<>();
```

```
List<String> dest = new ArrayList<>();
List<Number> src = new ArrayList<>();
// 不报错，"? " 忽略参数是否一致，只要传入即可。
gmlMapperGeneric.testNon(dest, src);
// 报错，"T" 会校验参数是否一致。
gmlMapperGeneric.testT(dest, src);
}
```

区别2：类型参数可以多重限定而通配符不行

```
interface MultiLimitInterfaceA {}
interface MultiLimitInterfaceB {}

class MultiLimit implements MultiLimitInterfaceA,
MultiLimitInterfaceB {
    /**
     * 使用 & 符，设置多重边界
     */
    public <T extends MultiLimitInterfaceA & MultiLimitInterfaceB>
void method(T t) {
}
}
```

使用 **&** 符号设定多重边界 (**Multi Bounds**)，指定泛型类型 **T** 必须是 **MultiLimitInterfaceA** 和 **MultiLimitInterfaceB** 的共有子类型，此时变量 **t** 就具有了所有限定的方法和属性。

对于通配符 **"?"** 来说，因为它不是一个确定的类型，所以不能进行多重限定。

区别3：通配符可以使用超类限定而类型参数不行

例如：通配符 **?** 可以

```
List<? extends T> getList();
```

```
List<? super T> getList();
```

但参数 **T** 只能：

```
List getList();
```

Synchronized关键字

synchronized 是 Java 中的一个关键字，翻译成中文是同步的意思，主要解决的是多个线程之间访问资源的同步性，可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

锁的升级

1. 轻量级锁：适用于线程交替执行同步代码块的情况。轻量级锁通过CAS（Compare and Swap）操作尝试获取锁，如果成功就直接执行同步代码块；如果失败，就采用自旋的方式等待其他线程释放锁。轻量级锁的优点是性能高效，缺点是当同步块竞争激烈时会降低系统的吞吐量。
2. 偏向锁：适用于只有一个线程执行同步代码块的情况。偏向锁通过记录线程ID来判断是否需要获取锁，如果线程ID与记录的ID相同，则不需要获取锁。偏向锁的优点是可以减少同步操作的消耗，缺点是当多个线程竞争同一个锁时会失去优势。**如果在无锁状态下，某个线程持有了锁，并且在接下来的一段时间内，其他线程并没有竞争该锁，那么这个锁就会进化成偏向锁。此时，对象头中的标记字段会被设置为偏向锁，指向持有锁的线程的Thread ID。在偏向锁状态下，当其他线程想要访问同一个共享资源时，会首先检查对象头中的标记，如果标记是偏向锁，并且指向的是当前线程，那么该线程就可以直接获取锁，无需进行CAS操作，提高了程序的执行效率。**
3. 重量级锁：适用于多个线程交替执行同步代码块的情况。重量级锁通过操作系统的Mutex等原语实现，当多个线程同时访问同步代码块时，其他线程会进入等待状态，直到持有锁的线程释放锁。重量级锁的优点是可以保证并发的正确性，缺点是由于涉及到内核态的切换，性能较低。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。

轻量级锁

轻量级锁是为了优化对于锁的争用情况而提出的一种锁实现方式，它的设计思路是当锁竞争出现时，不再使用重量级锁（synchronized），而是将对象头标记为“偏向锁”或“轻量级锁”状态，以避免线程进入操作系统内核态，从而提升程序执行效率。

当一个线程进入同步代码块时，如果该锁对象处于无锁状态（锁标志位为“01”），线程通过CAS操作将对象头中的锁标志位设置为“00”，并将锁对象的Mark Word拷贝到线程的栈帧中，同时在栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象的Mark Word的拷贝。

当线程尝试释放锁时，它会将拷贝到栈帧中的Mark Word原封不动地写回到对象头中，并尝试使用CAS操作将对象头中的锁标志位设置为“01”，如果CAS操作成功，说明当前线程释放了锁，其他线程可以继续争抢该锁。

当有第二个线程尝试获得锁时，它会先判断对象头中的锁标志位是否为“00”，如果是，则说明该锁正在被当前线程持有，可以直接进入同步代码块。如果不是，则说明该锁已经被其他线程持有，当前线程将进行自旋等待，直到获得锁或者线程挂起。

当线程自旋等待的时间达到一定阈值或者其他线程已经升级为重量级锁，则该锁会被升级为重量级锁，后续线程访问该锁时直接阻塞等待。

总之，轻量级锁适用于同步代码块执行时间较短、锁竞争不激烈的场景，可以避免线程进入内核态，提高程序的执行效率。

在对象头中，存储对象的锁状态的是一个叫做 **Mark Word** 的字段。Mark Word 的具体格式取决于虚拟机的版本和配置，但是一般来说，它包含了对对象的哈希码、锁标记、分代年龄等信息。

修饰实例方法

需要注意的是，synchronized关键字用于实例方法时，锁定的是该实例对象，而不是该类的所有实例对象。因此，如果有多个BankAccount实例对象，它们之间不会互相影响，各自拥有自己的锁。如果希望锁定整个类的所有实例对象，可以使用synchronized关键字修饰静态方法。

修饰静态方法

方法被synchronized关键字修饰，因此它是一个同步方法，可以确保每次只有一个线程能够访问该方法。

SPI

SPI (Service Provider Interface) 是 Java 中一种基于接口的服务发现机制。它的实现原理是：通过在 Classpath 下的 META-INF/services 目录中创建一个以服务接口命名的文件，并在文件中写入实现服务接口的类的全限定名，从而使得程序能够在运行时动态地发现并加载这些实现类。

具体来说，假设我们有一个服务接口 `Service`：

```
public interface Service {  
    void doSomething();  
}
```

现在，我们需要为这个服务接口提供多个实现类，并让用户在程序运行时根据不同的需求来选择使用哪个实现类。这时，我们可以通过 SPI 机制来实现。

首先，我们在 Classpath 下的 META-INF/services 目录中创建一个以服务接口命名的文件 `com.example.Service`，然后在文件中写入实现服务接口的类的全限定名，每行一个类名，例如：

```
com.example.ServiceImpl1  
com.example.ServiceImpl2
```

然后，我们就可以通过 `ServiceLoader` 类来动态地加载这些实现类并使用它们：

```
ServiceLoader<Service> loader = ServiceLoader.load(Service.class);  
for (Service service : loader) {  
    service.doSomething();  
}
```

在上面的例子中，我们使用 `ServiceLoader.load` 方法来加载 `Service` 接口的所有实现类，然后通过遍历这些实现类的实例来调用它们的 `doSomething` 方法。

通过 SPI 机制，我们可以让程序更加灵活地**管理和使用不同的实现类**，从而增加了程序的扩展性和可维护性。

Spring

AspectJ

spectJ 是一个基于 Java 语言的切面编程（Aspect-Oriented Programming, AOP）扩展，它允许开发人员使用新的模块化方式来组织 Java 代码。使用 AspectJ，开发人员可以将通用的横切关注点（cross-cutting concerns）从应用程序的核心业务逻辑中分离出来，例如安全性、日志记录、事务管理、异常处理等，从而提高应用程序的可重用性、可维护性和可扩展性。

AspectJ 是 Java 语言的一个扩展，它通过引入新的语法元素，包括切入点（pointcut）、通知（advice）、切面（aspect）、连接点（join point）等，使得开发人员可以将关注点从代码中剥离出来，然后以独立的方式实现这些关注点。

AspectJ 支持五种通知类型，包括前置通知（before）、后置通知（after）、返回通知（after-returning）、抛出通知（after-throwing）和环绕通知（around），开发人员可以根据需要选择不同的通知类型来实现不同的关注点。

Spring Context

"org.springframework.context.ApplicationContext 接口"表示Spring IoC容器，负责实例化、配置和组装bean。容器通过读取配置元数据获取关于要实例化、配置和组装哪些对象的指令。配置元数据用XML、Java注解或Java代码表示。允许表达组成程序的对象以及这些对象之间丰富的相互依赖关系。

Spring提供了ApplicationContext接口的几个实现。在独立应用程序中，通常创建ClassPathXmlApplicationContext或FileSystemXmlApplicationContext的实例。

```
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyClass {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        MyBean myBean = (MyBean) context.getBean("myBean");
        // 进行相关的操作，例如调用 myBean 的方法
        myBean.doSomething();
        context.close();
    }
}
```

Spring IOC（控制反转）

可以顺便谈Bean的生命周期

指创建对象的控制权的转移。以前创建对象的主动权和时机是由自己把控，需要自己new一个对象来使用，而现在把这种权力转移到Spring容器中，由容器根据配置文件去创建实例和管理各个实例之间的依赖关系，需要用的时候从Spring容器中去getBean()来获取。从而实现具有依赖关系的对象之间的解耦，从而降低代码之间的耦合度。

```
public static void main(String[] args) {
    ApplicationContext applicationContext = new
WJZApplicationContext("xxx.xml");
    Object user = applicationContext.getBean("user");
    System.out.println(user);
}
```

DI(依赖注入)

DI其实是Ioc的实现方式，即应用程序在运行时依赖Ioc容器来动态注入对象需要的外部资源。主要有三种方式：setter注入、构造器注入、注解注入。

1、setter方法注入

使用类的setter方法注入，要求类必须有setter方法

property 标签的属性:

1. name: setXxx方法名称去掉set、首字母小写后得到的名称
2. value: 要赋予的属性值（基本数据类型与String）
3. ref: 引入外部的bean，值为对应的id属性名，只要IOC容器中有就行了，同一个xml文件可以没有

```
<bean id="student_set" class="com.zyd.pojo.Student">
    <property name="id" ref="id"/>
    <property name="uname" value="李四"/>
    <property name="age" value="18"/>
</bean>

<bean id="id" class="java.lang.Integer">
    <constructor-arg type="int" value="10"/>
</bean>
```

2、构造器注入

构造器注入用constructor-arg标签指定参数

1. type: 传入的参数类型
2. name: 参数列表的参数名，不是属性名(虽然大部分情况两者名称一致)
3. index: 参数列表的下标，从0开始

前三种方式任选一都可以

4. ref: 引入外部的bean，值为对应的id属性名，只要IOC容器中有就行了，同一个xml文件可以没有
5. value: 要传入的参数值

```

<bean id="int" class="java.lang.Integer">
    <constructor-arg type="int" value="10"/>
</bean>

<bean id="student" class="com.zyd.pojo.Student">
    <constructor-arg type="java.lang.Integer" ref="int"/>
    #ref的意思是把Integer的对象引用注入到student中来
    <constructor-arg name="username" value="张三"/>
    <constructor-arg index="2" value="20"/>
</bean>

```

3、注解注入

首先要在要添加注解的类上加一个@Component注解

@Autowired注解

在成员变量上直接标记@Autowired注解即可完成自动装配。

默认是byType方式，如果匹配不上就会byName。

```

public class People {
    @Autowired
    private Cat cat;
    @Autowired
    private Dog dog;
    private String name;
}

```

//由于在属性cat上添加了@Autowired,在自动装配时会首先在容器里寻找（Class）Type为Cat的Bean,若找不到会寻找(id)name为cat的Bean。

为了解决这个问题，可以使用byName的方式来进行依赖注入。使用byName的方式，需要在注入的字段上使用@Autowired和@Qualifier注解。

```

public interface OneService {
    void doSomething();
}

@Component("serviceImpl1")
public class ServiceImpl1 implements OneService {
    @Override
    public void doSomething() {
        System.out.println("ServiceImpl1.doSomething()");
    }
}

```

```

}

@Component("serviceImpl2")
public class ServiceImpl2 implements OneService {
    @Override
    public void doSomething() {
        System.out.println("ServiceImpl2.doSomething()");
    }
}

```

```

@Component
public class MyComponent {
    @Autowired
    @Qualifier("serviceImpl1")
    private OneService service;

    public void doSomething() {
        service.doSomething();
    }
}

```

@Resource注解

在成员变量上直接标记@Resource注解即可完成自动装配。

默认是byName方式，如果匹配不上就会byType。

区别：

- Autowired是先byType,如果唯一则注入，否则byName查找。resource是先byname,不符合再继续byType
- 都是用来自动装配的，都可以放在属性字段上
- @Autowired通过byType的方式实现，而且必须要求这个对象存在！【常用】
- @Resource默认通过byname的方式实现，如果找不到名字，则通过byType实现！如果两个都找不到的情况下，就报错！【常用】
- 执行顺序不同：@Autowired通过byType的方式实现。@Resource默认通过byName的方式实现


```
@Component
public class MyComponent {
    @Resource(name = "userService1")
    private UserService userService1;

    @Resource(type = UserServiceImpl2.class)
    private UserService userService2;
}
```

@Bean注解

单例Bean的线程安全问题

补充一个概念：

有状态的bean就是有数据存储功能。无状态的bean就是不会保存数据。

singleton表示该bean全局只有一个实例，Spring中bean的scope默认也是singleton。

prototype表示该bean在每次被注入的时候，都要重新创建一个实例。

- 如果有状态的Bean配置为singleton，便会出现以下情况：用户a想要修改有状态bean中保存的值，但是这时用户b也在修改bean中保存的值，b完成修改后a提交他修改好的值。此时b的操作就被a完全覆盖了。
- 如果使用的prototype的方式，每次注入都产生一个实例。就不存在数据共享的情况，也就不存在线程安全的问题。

Spring中的单例与设计模式里面的单例略有不同，设计模式的单例是在整个应用中只有一个实例，而Spring中的单例是在一个IOC容器中就只有一个实例。

解决方法：

- 将Bean的作用域由“singleton”单例 改为“prototype”多例。
- 在类中定义 ThreadLocal 的成员变量，并将需要的可变成员变量保存在 ThreadLocal 中，ThreadLocal 本身就具备线程隔离的特性，这就相当于为每个线程提供了一个独立的变量副本，每个线程只需要操作自己的线程副本变量，从而解决线程安全问题。

当创建一个ThreadLocal变量时，访问这个变量的每个线程都有这个变量的一个本地副本，当多个线程操作这个变量时，实际上就是操作自己本地内存里面的变量，从而避免了线程安全问题。

```
public class ThreadLocalTest {
```

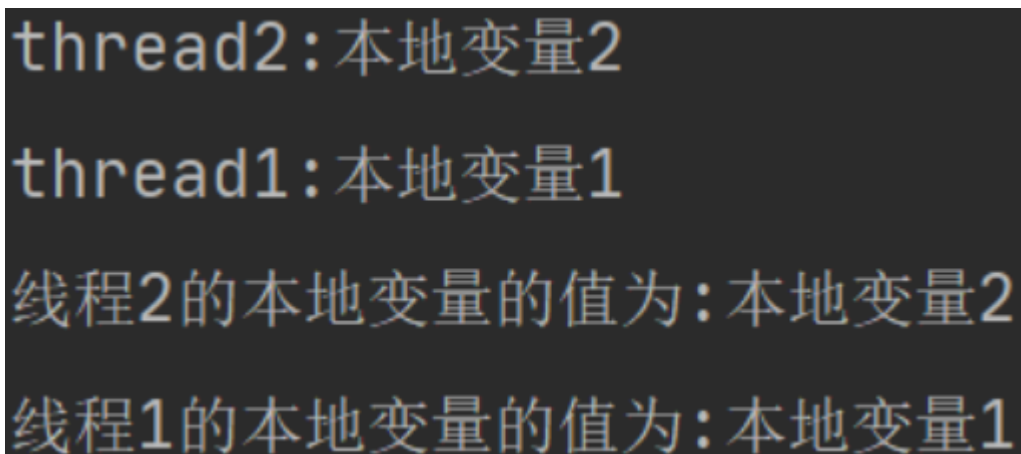
```
private static ThreadLocal<String> threadLocal = new
ThreadLocal<>();

public static void main(String[] args) {
    Thread thread1 = new Thread(() -> {
        threadLocal.set("本地变量1");
        print("thread1");
        System.out.println("线程1的本地变量的值
为:"+threadLocal.get());
    });

    Thread thread2 = new Thread(() -> {
        threadLocal.set("本地变量2");
        print("thread2");
        System.out.println("线程2的本地变量的值
为:"+threadLocal.get());
    });

    thread1.start();
    thread2.start();
}

public static void print(String s){
    System.out.println(s+": "+threadLocal.get());
}
}
```



```
thread2:本地变量2
thread1:本地变量1
线程2的本地变量的值为:本地变量2
线程1的本地变量的值为:本地变量1
```

大部分 Bean 实际都是无状态（没有实例变量）的（比如 Dao、Service），这种情况下，Bean 是线程安全的。

SpringBean的生命周期

实例化:Spring容器根据配置文件或注解创建一个Bean定义，这个定义描述了Bean的类，依赖关系等。然后容器使用Java反射机制创建一个Bean的实例。

属性赋值:Spring容器将在Bean实例化后，通过调用Setter方法等方式来将属性赋值给Bean。这个过程可以通过XML配置或注解来实现。

三，初始化:在Bean的属性赋值完成后，Spring容器会调用Bean的初始化方法，这个方法可以是自定义的，需要在Bean的配置文件或注解中进行定义。

四，使用: Bean初始化完成后，就可以使用了。这个时候Bean已经被完全构建并准备好被其他组件使用。

五，销毁:当Bean不再需要使用时，Spring容器会调用Bean的销毁方法，这个方法可以是自定义的，需要在Bean的配置文件或注解中进行定义。

各种接口方法分类

Bean的完整生命周期经历了各种方法调用，这些方法可以划分为以下几类：

1、Bean自身的方法：这个包括了Bean本身调用的方法和通过配置文件中的**init-method****和**destroy-method**指定的方法

2、Bean级生命周期接口方法：这个包括了**BeanNameAware**、**BeanFactoryAware**、**InitializingBean**和**DiposableBean**这些接口的方法

- 当一个 Bean 实现了 `BeanNameAware` 接口时，容器在实例化该 Bean 后，会通过调用 `setBeanName()` 方法来将该 Bean 的名字注入到该方法中，以供 Bean 使用。
- 当一个 Bean 实现了 `BeanFactoryAware` 接口时，容器在实例化该 Bean 后，会通过调用 `setBeanFactory()` 方法将该 Bean 的工厂对象注入到该方法中，以供 Bean 使用。这样，该 Bean 就能够通过该工厂对象来获取其他 Bean 对象了。

`InitializingBean` 接口是 Spring 提供的一个回调接口，用于在 Bean 初始化之后执行一些定制化的操作。这个接口定义了一个方法 `afterPropertiesSet()`，当 Spring 容器完成了对 Bean 的依赖注入和属性赋值之后，会自动调用该方法，以完成 Bean 的初始化工作。

实现 `InitializingBean` 接口的 Bean 可以在 `afterPropertiesSet()` 方法中进行一些初始化操作，比如检查 Bean 的属性是否已被正确设置、创建一些辅助对象、连接一些外部资源等等。通过实现 `InitializingBean` 接口，我们可以确保在 Bean 初始化完成之后，该 Bean 的依赖关系已被正确地设置和初始化，从而保证 Bean 在运行时具备完整的状态和功能。

`DisposableBean` 接口是 Spring 提供的一个回调接口，用于在 Bean 销毁之前执行一些清理操作。这个接口定义了一个方法 `destroy()`，当 Spring 容器销毁 Bean 的时候，会自动调用该方法，以完成 Bean 的清理工作。

实现 `DisposableBean` 接口的 Bean 可以在 `destroy()` 方法中进行一些清理操作，比如释放一些资源、关闭一些连接、保存一些状态等等。通过实现 `DisposableBean` 接口，我们可以确保在 Bean 销毁之前，该 Bean 所占用的资源已被正确地释放和清理，从而避免资源泄漏和其他潜在的问题。

需要注意的是，使用 `DisposableBean` 接口进行 Bean 销毁操作同样是一种侵入式的方式。因为该接口的实现依赖于 Spring 框架，因此它会将 Bean 与 Spring 框架紧密耦合在一起。如果我们不希望 Bean 和 Spring 框架之间存在强耦合性，可以考虑使用其他方式来实现 Bean 的销毁，比如 `@PreDestroy` 注解、自定义的销毁方法等。

3、容器级生命周期接口方法：这个包括了 `InstantiationAwareBeanPostProcessor` 和 `BeanPostProcessor` 这两个接口实现，一般称它们的实现类为“后处理器”。

`InstantiationAwareBeanPostProcessor` 是 `BeanPostProcessor` 接口的子接口之一，它在 Spring 容器实例化 Bean 对象时提供了一个扩展点，用于在实例化 Bean 对象之前或之后对其进行自定义处理，比如对 Bean 对象进行属性赋值、更改对象类型、修改构造函数参数等等。

`InstantiationAwareBeanPostProcessor` 中定义了如下方法：

1. `postProcessBeforeInstantiation`：在实例化 Bean 对象之前进行自定义处理。
2. `postProcessAfterInstantiation`：在实例化 Bean 对象之后进行自定义处理，这个方法是在对象实例化后，属性填充之前调用的，因此如果需要通过这个方法改变 Bean 对象的属性，需要在 `postProcessPropertyValues` 方法中进行。
3. `postProcessPropertyValues`：在对象的属性填充完毕后，对 Bean 对象进行进一步的自定义处理，比如更改属性值等等。
4. `postProcessBeforeInitialization`：在 Bean 对象进行初始化之前进行自定义处理，可以对 Bean 对象进行一些预处理，比如添加一些自定义注解，修改属性等等。
5. `postProcessAfterInitialization`：在 Bean 对象进行初始化之后进行自定义处理，可以对 Bean 对象进行一些后处理，比如添加一些切面、修改属性等等。

`BeanPostProcessor` 接口是 Spring 中一个重要的扩展点，用于在 Bean 的生命周期中对 Bean 进行一些处理。具体来说，`BeanPostProcessor` 接口定义了两个方法 `postProcessBeforeInitialization()` 和 `postProcessAfterInitialization()`，这两个方法分别在 Bean 初始化之前和初始化之后被调用，允许我们在 Bean 初始化的过程中对 Bean 进行自定义的操作。

`BeanPostProcessor` 接口可以被用来扩展 Spring 容器的功能，比如在 Bean 初始化之前进行一些额外的操作（例如修改属性值、检查依赖关系等等），或者在 Bean 初始化之后进行一些后处理（例如修改返回的对象、添加代理对象等等）。常见的使用场景包括：AOP 代理、属性占位符替换、消息源国际化等。

4、工厂后处理器接口方法：这个包括了 **BeanFactoryPostProcessor** (**AspectJWeavingEnabler**, **ConfigurationClassPostProcessor**, **CustomAutowireConfigurer**) 等等非常有用的工厂后处理器接口的方法。工厂后处理器也是容器级的。在应用上下文装配配置文件之后立即调用。

Spring Bean的完整生命周期从创建Spring容器开始，直到最终Spring容器销毁Bean

- 开始初始化容器
- 工厂后处理

BeanFactoryPostProcessor实现类构造器

BeanFactoryPostProcessor调用postProcessBeanFactory方法

- 容器级生命周期接口方法

这是BeanPostProcessor实现类构造器！！

这是InstantiationAwareBeanPostProcessorAdapter实现类构造器！！

Bean实例化之前的方法：InstantiationAwareBeanPostProcessor调用postProcessBeforeInstantiation方法

- 【构造器】调用Person的构造器实例化
- 在对象的属性填充完毕后，对 Bean 对象进行进一步的自定义处理
InstantiationAwareBeanPostProcessor调用postProcessPropertyValues方法
- 【注入属性】注入属性address
【注入属性】注入属性name
【注入属性】注入属性phone
- 【BeanNameAware接口】调用BeanNameAware.setBeanName()
【BeanFactoryAware接口】调用BeanFactoryAware.setBeanFactory()
- 初始化之前的方法：BeanPostProcessor接口方法postProcessBeforeInitialization对属性进行更改！
- 【InitializingBean接口】调用InitializingBean.afterPropertiesSet()
【init-method】调用的init-method属性指定的初始化方法
- 初始化之后的方法：BeanPostProcessor接口方法postProcessAfterInitialization对属性进行更改！

- 实例化之后的方法：InstantiationAwareBeanPostProcessor调用postProcessAfterInitialization方法
- 容器初始化成功
- 现在开始关闭容器！
 - 【DisposableBean接口】调用DisposableBean.destory()
 - 【destroy-method】调用的destroy-method属性指定的初始化方法

AOP

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

Spring AOP 就是基于动态代理的，如果要代理的对象，实现了某个接口，那么 Spring AOP 会使用 **JDK Proxy**，去创建代理对象，而对于没有实现接口的对象，就无法使用 JDK Proxy 去进行代理了，这时候 Spring AOP 会使用 **Cglib** 生成一个被代理对象的子类来作为代理

Spring AOP 已经集成了 AspectJ，AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。

AOP 切面编程设计到的一些专业术语：

术语	含义
目标(Target)	被通知的对象
代理(Proxy)	向目标对象应用通知之后创建的代理对象
连接点(JoinPoint)	目标对象的所属类中，定义的所有方法均为连接点
切入点(Pointcut)	被切面拦截 / 增强的连接点（切入点一定是连接点，连接点不一定是切入点）
通知(Advice)	增强的逻辑 / 代码，也即拦截到目标对象的连接点之后要做的事情
切面(Aspect)	切入点(Pointcut)+通知(Advice)
Weaving(织入)	将通知应用到目标对象，进而生成代理对象的过程动作

Spring AOP 属于运行时增强，而 AspectJ 是编译时增强。 Spring AOP 基于代理 (Proxying)，而 AspectJ 基于字节码操作(Bytecode Manipulation)。

AspectJ 定义的通知类型有哪些？

- **Before**（前置通知）：目标对象的方法调用之前触发
- **After**（后置通知）：目标对象的方法调用之后触发
- **AfterReturning**（返回通知）：目标对象的方法调用完成，在返回结果值之后触发
- **AfterThrowing**（异常通知）：目标对象的方法运行中抛出 / 触发异常后触发。
AfterReturning 和 AfterThrowing 两者互斥。如果方法调用成功无异常，则会有返回值；如果方法抛出了异常，则不会有返回值。
- **Around**（环绕通知）：程式控制目标对象的方法调用。环绕通知是所有通知类型中可操作范围最大的一种，因为它可以直接拿到目标对象，以及要执行的方法，所以环绕通知可以任意的在目标对象的方法调用前后搞事，甚至不调用目标对象的方法

相关注解介绍、

AOP的注解

@Aspect

前置注解，被注解的方法将会在目标方法之前执行

@Before(value="execution(* demo1.userdao.add(..))")

后置注解，被注解的方法将会在目标方法执行之后执行

@AfterReturning(value = "execution(* demo1.userdao.add(..))")

环绕注解，被注解的方法将会之前和之后都进行执行

@Around(value = "execution(* demo1.userdao.add(..))")

异常注解，被注解的方法只会在目标方法发生异常的时候才会执行

@AfterThrowing(value = "execution(* demo1.userdao.add(..))")

最终注解，被注解的方法将在执行完所有的注解方法后再执行

@After(value="execution(* demo1.userdao.add(..))")

实现AOP

@Aspect//增强注解

```
public class user {  
    @Before(value="execution(* demo1.userdao.add(..))")  
    public void before(){  
        System.out.println("====前置====");  
    }  
    @AfterReturning(value = "execution(* demo1.userdao.add(..))")  
    public void AfterReturning(){  
        System.out.println("====后置====");  
    }  
    @After(value="execution(* demo1.userdao.add(..))")  
    public void after(){  
        System.out.println("====最终====");  
    }  
}
```

```

@AfterThrowing(value = "execution(* demo1.userdao.add(..)")
public void AfterThrowing(){
    System.out.println("=====异常====");
}

@Around(value = "execution(* demo1.userdao.add(..)")
    public void around(ProceedingJoinPoint proceedingJoinPoint)
throws
    Throwable {
        System.out.println("环绕之前.....");
        //被增强的方法执行
        proceedingJoinPoint.proceed();
        System.out.println("环绕之后.....");
    }
}

```

对相同切入点的公共化

对于一个切入点，如果需要进行多次增强，就是使用多个注解对这个方法进行增强，我们可以将这个切入点表达式，也就是这个路径(value = "execution(* demo1.userdao.add(..)")进行公共化，将设置一个这个路径获取的方法，就可以直接对路径进行获取。

```

//相同切入点抽取
@Pointcut(value = "execution(*
com.atguigu.spring5.aopanno.User.add(..)")
public void pointdemo() {
}

//前置通知
//@Before 注解表示作为前置通知
@Before(value = "pointdemo()")
public void before() {
    System.out.println("before.....");
}

```

设置多个增强类的优先级

对于多个类对同一个方法的增强时，想要对这个些类的执行顺序进行设置，我们可以通过添加@Order注解来实现，（）中的数字越小优先级越高。

```
@Component
@Aspect
@Order(1)
public class PersonProxy
```

Spring MVC

MVC 是一种设计模式，Spring MVC 是一款很优秀的 MVC 框架。Spring MVC 可以帮助我们进行更简洁的 Web 层的开发，并且它天生与 Spring 框架集成。Spring MVC 下我们一般把后端项目分为 Service 层（处理业务）、Dao 层（数据库操作）、Entity 层（实体类）、Controller 层（控制层，返回数据给前台页面）。

Spring MVC 的核心组件有哪些？

DispatcherServlet：核心的中央处理器，负责接收请求、分发，并给予客户端响应。

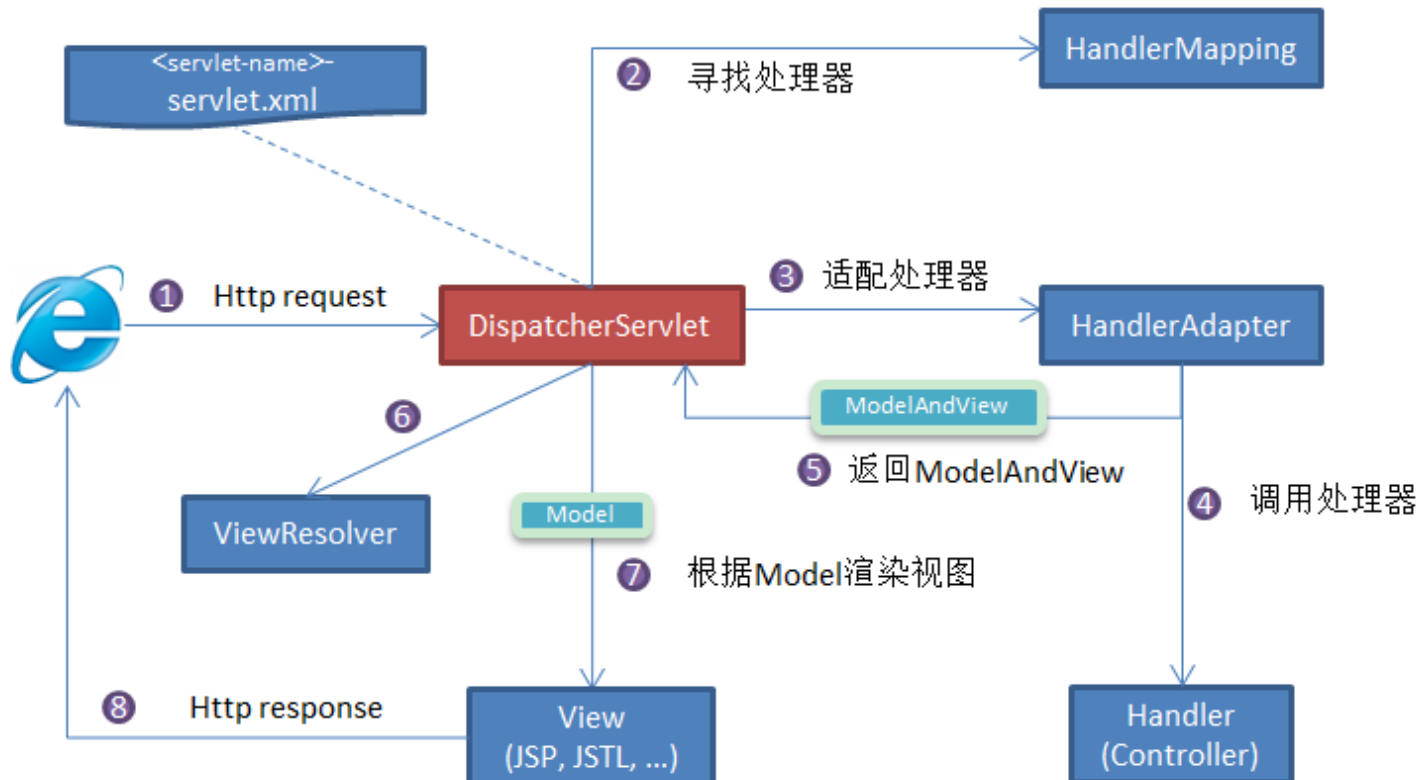
HandlerMapping：处理器映射器，根据 uri 去匹配查找能处理的 **Handler**，并会将请求涉及到的拦截器和 **Handler** 一起封装。

HandlerAdapter：处理器适配器，根据 **HandlerMapping** 找到的 **Handler**，适配执行对应的 **Handler**；

Handler：请求处理器，处理实际请求的处理器。

ViewResolver：视图解析器，根据 **Handler** 返回的逻辑视图 / 视图，解析并渲染真正的视图，并传递给 **DispatcherServlet** 响应客户端

SpringMVC 工作原理



流程说明（重要）：

1. 客户端（浏览器）发送请求， `DispatcherServlet` 拦截请求。
2. `DispatcherServlet` 根据请求信息调用 `HandlerMapping`。`HandlerMapping` 根据 uri 去匹配查找能处理的 `Handler`（也就是我们平常说的 `Controller` 控制器），并会将请求涉及到的拦截器和 `Handler` 一起封装。
3. `DispatcherServlet` 调用 `HandlerAdapter` 适配执行 `Handler`。
4. `Handler` 完成对用户请求的处理后，会返回一个 `ModelAndView` 对象给 `DispatcherServlet`，`ModelAndView` 顾名思义，包含了数据模型以及相应的视图的信息。`Model` 是返回的数据对象，`view` 是个逻辑上的 `view`。
5. `ViewResolver` 会根据逻辑 `view` 查找实际的 `View`。
6. `DispatcherServlet` 把返回的 `Model` 传给 `View`（视图渲染）。
7. 把 `view` 返回给请求者（浏览器）

MySQL

InnoDB

<https://dev.mysql.com/doc/refman/8.0/en/innodb-introduction.html>

DML

数据库存储引擎的DML（Data Manipulation Language）指的是用于操作数据的SQL语句，主要包括以下几种：

1. SELECT：用于查询数据库中的数据。

2. INSERT：用于向数据库中插入新的数据。
3. UPDATE：用于更新数据库中的数据。
4. DELETE：用于删除数据库中的数据。

这些DML操作是数据库中最常用的操作之一，它们可以帮助用户实现各种复杂的数据操作和查询。具体的语法和用法可以根据不同的数据库系统和存储引擎进行调整，但是大体上是类似的。

数据库ACID模型

ACID模型是数据库系统中用于描述事务特性的一个概念，它包括四个关键特性：

1. 原子性 (Atomicity)：事务应该被视为不可分割的单元，要么全部执行成功，要么全部失败回滚。在事务执行期间，如果发生错误或异常，所有的操作都将被撤消，将数据恢复到事务开始之前的状态。
2. 一致性 (Consistency)：事务执行前后，数据应该处于一致状态，满足数据库定义的各种约束和完整性条件。在事务执行期间，如果有任何冲突或错误发生，所有的操作都将被回滚，以保证数据的一致性。
3. 隔离性 (Isolation)：多个事务并发执行时，各个事务之间应该互相隔离，保证它们之间的操作不会相互影响。这可以通过锁机制来实现。
4. 持久性 (Durability)：一旦事务被提交，它的结果应该被永久地保存在数据库中，即使系统崩溃或重启也不会丢失。

ACID模型的目的是确保数据库的数据一致性和可靠性。数据库管理系统必须支持ACID模型，以便保证事务执行的正确性和安全性。在实际应用中，开发人员可以根据应用的需求，选择合适的数据库存储引擎来支持ACID模型，以保证数据的正确性和稳定性。

一致性

在数据库中，一致性(Consistency)指的是在事务执行前和执行后，数据库的状态应该保持一致。

具体来说，一致性要求数据库中的数据必须符合预定义的规则和约束，包括实体完整性、参照完整性、域完整性等。实体完整性要求每个表中的每个实体都有唯一的标识符，参照完整性要求每个引用表中的实体在被引用前必须已经存在，域完整性要求每个列的取值必须满足特定的数据类型和范围等限制条件。

事务的一致性要求在事务执行期间，数据库中的数据必须遵守这些约束和规则。如果事务违反了这些约束或规则，事务会被回滚并恢复到事务开始前的状态，以确保数据库的一致性。

隔离性

在数据库中，事务的隔离级别定义了并发执行的事务之间的隔离程度。通常情况下，隔离级别越高，数据的一致性和事务的独立性就越好，但是并发性能就会相应地降低。常见的隔离级别包括以下四个：

1. 未提交读取（Read Uncommitted） 未提交读取是最低的隔离级别，它允许一个事务读取其他事务尚未提交的数据。在这个级别下，一个事务可能会读取到脏数据，即另一个事务所修改但尚未提交的数据，这样可能导致脏读问题。未提交读取隔离级别的优点是并发性能高，但是一致性和数据完整性得不到保证。
2. 已提交读取（Read Committed） 已提交读取是默认的隔离级别，它要求一个事务只能读取其他事务已经提交的数据。在这个级别下，一个事务只会读取到其他事务已经提交的数据，因此可以避免脏读问题。但是，由于其他事务可能在读取之后修改了数据并提交，所以同一个事务多次读取同一个数据可能会得到不同的结果，这就是不可重复读问题。
3. 可重复读取（Repeatable Read） 可重复读取隔离级别要求一个事务在执行期间多次读取同一个数据时得到的结果保持一致。为了达到这个目的，当一个事务读取数据时，数据库会对读取的数据加锁，防止其他事务修改这个数据。这样，即使其他事务在执行期间对这个数据进行了修改和提交，这个事务读取的数据仍然保持一致，不会出现不可重复读问题。但是，在这个级别下，一个事务可能会发生幻读问题，即一个事务在执行期间多次执行同一查询，得到的结果集合不同。这是因为在可重复读取级别下，数据库只会对读取的数据加行锁，而无法防止其他事务插入新的数据。
4. 串行化（Serializable） 串行化隔离级别要求所有事务串行执行，即每个事务在执行时需要先获得锁，直到事务结束后才释放锁，这样可以避免脏读、不可重复读和幻读等问题。但是，串行化级别的并发性能非常低，因为每个事务都需要等待其他事务释放锁后才能执行，这对于高并发的应用来说是无法接受的。

B+树

<https://blog.csdn.net/zsz0147/article/details/117985568>