1.基础

多态

JAVA运行时系统根据调用该方法的实例的类型来决定选择调用哪个方法则被称为运行时多态。（我们平时说得多的事运行时多态，所以多态主要也是指运行时多态）

运行时多态存在的三个必要条件：

一、要有继承（包括接口的实现）；

二、要有重写；

三、父类引用指向子类对象

优先级从高到低：

this.show(0)，super.show(0)，this.show((super)0)，super.show((super)0)。

```
class A {
    public String show(D obj)...{
        return ("A and D");
    }
    public String show(A obj)...{
```

```
            return ("A and A");
        }
}
class B extends A{
        public String show(B obj)...{
            return ("B and B");
        }
        public String show(A obj)...{
            return ("B and A");
        }
}
class C extends B...{}
class D extends B...{}

public static void main(String[] args) {
        A a1 = new A();
        A a2 = new B();
        B b = new B();
        C c = new C();
        D d = new D();
        System.out.println(a1.show(b));   //A and A
        System.out.println(a1.show(c));   //A and A
        System.out.println(a1.show(d));   //A and D
        System.out.println(a2.show(b));   //B and A
        System.out.println(a2.show(c));   //B and A
        System.out.println(a2.show(d));   //A and D
        System.out.println(b.show(b));    //B and B
        System.out.println(b.show(c));    //B and B
        System.out.println(b.show(d));    //A and D
}

class A {
        public String show(D obj)...{
            return ("A and D");
        }
        public String show(A obj)...{
            return ("A and A");
        }
}
class B extends A{
        public String show(B obj)...{
            return ("B and B");
        }
        public String show(A obj)...{
            return ("B and A");
        }
}
```

```
class C extends B...{}
class D extends B...{}

public static void main(String[] args) {
        A a1 = new A();
        A a2 = new B();
        B b = new B();
        C c = new C();
        D d = new D();
        System.out.println(a1.show(b));   //A and A
        System.out.println(a1.show(c));   //A and A
        System.out.println(a1.show(d));   //A and D
        System.out.println(a2.show(b));   //B and A
        System.out.println(a2.show(c));   //B and A
        System.out.println(a2.show(d));   //A and D
        System.out.println(b.show(b));    //B and B
        System.out.println(b.show(c));    //B and B
```

1.1 objeck类方法

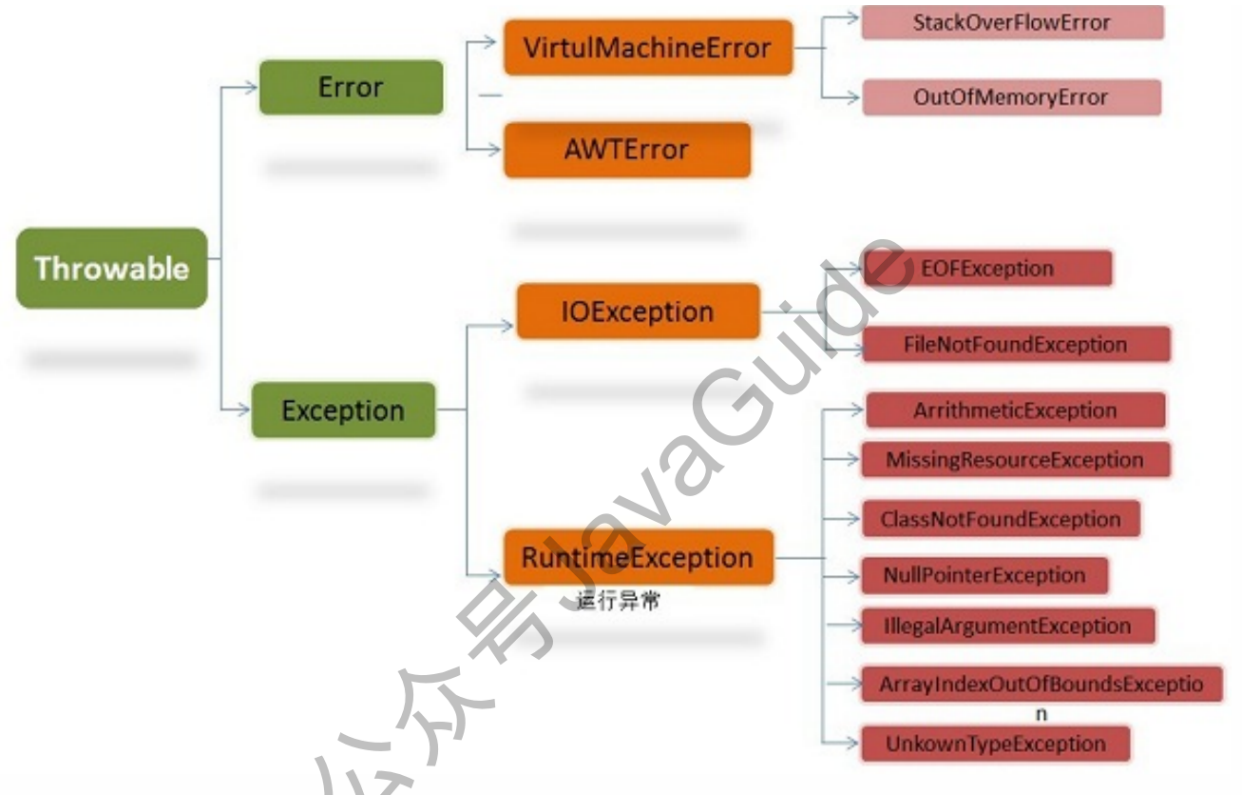
1. clone
2. getClass()
3. equals
4. hashCode()
5. toString()
6. wait(...) / notify() / notifyAll()方法

1.2 equals和==

== 比较2个对象的地址

equals比较2个对象的值,需要重写hashCode方法

1.3 异常处理



即便try语句中已经return ，依旧会执行finally,finally如果含有return语句,会覆盖返回

throw和throws

throws

throws是方法可能抛出异常的声明。(用在声明方法时，表示该方法可能要抛出异常)

```
public void function() throws Exception{.....}
```

throw

throw是语句抛出一个异常

```
throw new RuntimeExcetion("xxx");
```

1.4 Static关键字

修饰成员变量和成员方法： 被 static 修饰的成员属于类，不属于单个这个类的某个对象，被类中所有对象共享.

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

静态内部类（static修饰类的话只能修饰内部类）： 静态内部类与非静态内部类之间存在一个最大的区别：非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围类，但是静态内部类却没有。没有这个引用就意味着：1. 它的创建是不需要依赖外围类的创建。2. 它不能使用任何外围类的非static成员变量和方法。

静态导包(用来导入类中的静态资源，1.5之后的新特性)： 格式为：这两个关键字连用可以指定导入某个类中的指定静态资源，并且不需要使用类名调用类中静态成员，可以直接使用类中静态成员变量和成员方法。

1.5 final关键字

final关键字主要用在三个地方：变量、方法、类。

1. 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。
2. 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。
3. 使用final方法的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；第二个原因是效率。在早期的Java实现版本中，会将final方法转为内嵌调用。但是如果方法过于庞大，可能看不到内嵌调用带来的任何性能提升（现在的Java版本已经不需要使用final方法进行这些优化了）。类中所有的private方法都隐式地指定为final。

1.6 java数据类型

8种基本数据类型

byte：8位 1字节，最大存储数据量是255，存放的数据范围是-128~127之间。

short：16位 2字节，最大数据存储量是65536，数据范围是-32768~32767之间。

int：32位 4字节,最大数据存储容量是2的32次方减1,数据范围是负的2的31次方到正的2的31次方减1。

long：64位 8字节,最大数据存储容量是2的64次方减1,数据范围为负的2的63次方到正的2的63次方减1。

float：32位 4字节，数据范围在3.4e-45~1.4e38，直接赋值时必须在数字后加上f或F。

double：64位 4字节，数据范围在4.9e-324~1.8e308，赋值时可以加d或D也可以不加。

boolean：只有true和false两个取值。

char：16位 2字节，存储Unicode码，用单引号赋值。

double和float精度

分为符号位,指数位和尾数位

double=1+11+52

float=1+8+23

二进制表示小数

将该数字乘以2，取出整数部分作为二进制表示的第1位；然后再将小数部分乘以2，将得到的整数部分作为二进制表示的第2位；以此类推，知道小数部分为0。

特殊情况：　小数部分出现循环，无法停止，则用有限的二进制位无法准确表示一个小数，这也是在编程语言中表示小数会出现误差的原因

1.7 访问修饰符

下表为Java访问控制符的含义和使用情况

	类内部	本包	子类	外部包
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

1.8 接口和抽象类

1. 抽象类可以有默认的方法实现完全是抽象的。接口根本不存在方法的实现。

抽象类中可以有已经实现了的方法，也可以有被abstract修饰的方法（抽象方法），因为存在抽象方法，所以该类必须是抽象类。但是接口要求只能包含抽象方法，抽象方法是指没有实现的方法。所以就不能像抽象类那么无赖了，接口就根本不能存在方法的实现。(jdk8之后接口也有默认方法)

2.抽象类可以有构造器，而接口不能有构造器

3.抽象方法可以有public、protected和default这些修饰符;接口方法默认修饰符是public。你不可以使用其它修饰符。

4.抽象类在java语言中所表示的是一种继承关系，一个子类只能存在一个父类，但是可以存在多个接口。

总结:接口是对于方法的统一规范,而抽象类是对类的抽象

2.集合

2.1. 集合的继承关系

2个顶级接口 Collection和Map

Set和List 继承于 Collection

HashMap和TreeMap 继承于Map

2.2. List,Set,Map特点

List:　可以储存重复元素

Set: 不允许重复

Map: 键值对

2.3. ArrayList和LinkedList 区别

1. 均属于非不同步的集合
2. 底层数据结构:

   Object数组

   双向链表
3. 增删改效率
4. 快速随机访问: ArrayList支持,LinkedList不支持

2.4.HashMap和HashTable(重点)

- HashMap没有考虑同步，是线程不安全的；Hashtable使用了synchronized关键字，是线程安全的；
- HashMap允许null作为Key和Value；Hashtable不允许null作为Key，Hashtable的value也不可以为null

2.5. HashMap为什么不安全(重点)

- HashMap线程不安全主要是考虑到了多线程环境下进行扩容可能会出现HashMap死循环
- Hashtable线程安全是由于其内部实现在put和remove等方法上使用synchronized进行了同步，所以对**单个方法的使用是线程安全**的。但是对多个方法进行**复合操作时，线程安全性无法保证。** 比如一个线程在进行get操作，一个线程在进行remove操作，往往会导致下标越界等异常。

fast-fail机制

快速失败是Java集合的一种**错误检测机制，**当多个线程对集合进行结构上的改变的操作时，有可能会产生fail-fast。

例如：

假设存在两个线程（线程1、线程2），线程1通过Iterator在遍历集合A中的元素，在某个时候线程2**修改了集合A的结构**（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就**可能**会抛出 ConcurrentModificationException异常，从而产生fast-fail快速失败。

2.6.ConCurrentHashMap

ConcurrentHashMap和Hashtable的区别

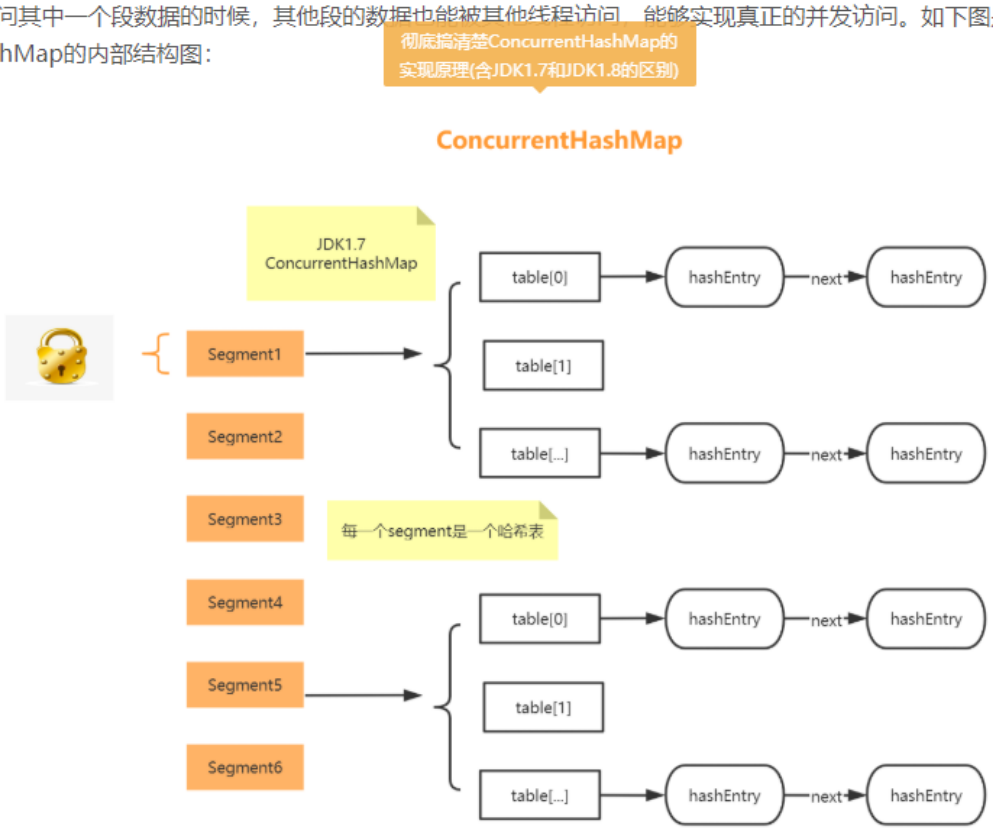
ConcurrentHashMap结合了HashMap和Hashtable二者的优势。HashMap没有考虑同步，Hashtable考虑了同步的问题。但是Hashtable在每次同步执行时都要锁住整个结构。

1.Segment(分段锁)

ConcurrentHashMap中的**分段锁称为Segment**，它即类似于HashMap的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表,同时又是一个ReentrantLock（Segment继承了ReentrantLock）。

2.内部结构

ConcurrentHashMap使用分段锁技术，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。如下图是ConcurrentHashMap的内部结构图：



面的结构我们可以了解到，ConcurrentHashMap定位一个元素的过程需要进行两次Hash操作。
第一次Hash定位到Segment，第二次Hash定位到元素所在的链表的头部。

3.该结构的优劣势

坏处

这一种结构的带来的副作用是Hash的过程要比普通的HashMap要长

好处

写操作的时候可以只对元素所在的Segment进行加锁即可，不会影响到其他的Segment，这样，在最理想的情况下，ConcurrentHashMap可以最高同时支持Segment数量大小的写操作（刚好这些写操作都非常平均地分布在所有的Segment上）。

所以，通过这一种结构，ConcurrentHashMap的并发能力可以大大的提高。

2.7.HashMap的长度为什么是2的幂次方?

- 我们将一个键值对插入HashMap中，通过**将Key的hash值与length-1进行&运算**，实现了当前Key的定位，2的幂次方可以减少冲突（碰撞）的次数，提高

HashMap查询效率

- **如果length为2的幂次方，则length-1 转化为二进制必定是11111……的形式，在与h的二进制与操作效率会非常的快，而且空间不浪费**
- **如果length不是2的幂次方，比如length为15，则length-1为14，对应的二进制为1110，在与h与操作，最后一位都为0，而0001，0011，0101，1001，1011，0111，1101这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！这样就会造成空间的浪费。**

总结：

也就是说2的N次幂有助于减少碰撞的几率，空间利用率比较大。

```
//Hashtable
Map<String, String> hashtable = new Hashtable<>();
//synchronizedMap
Map<String, String> synchronizedHashMap = Collections.synchronizedMap(new
HashMap<String, String>());
//ConcurrentHashMap
Map<String, String> concurrentHashMap = new ConcurrentHashMap<>();
```

2.8.TreeMap

TreeMap底层使用红黑树实现，TreeMap中存储的键值对按照键来排序。

- 如果Key存入的是字符串等类型，那么会按照字典默认顺序排序
- 如果传入的是自定义引用类型，比如说User，那么该对象必须实现Comparable接口，并且覆盖其compareTo方法；或者在创建TreeMap的时候，我们必须指定使用的比较器。

Comparable接口和Comparator接口有哪些区别呢?

- Comparable实现比较简单，但是当需要重新定义比较规则的时候，**必须修改源代码**，即修改User类里边的compareTo方法
- Comparator接口不需要修改源代码，只需要在创建TreeMap的时候**重新传入一个具有指定规则的比较器**即可。

3.多线程

3.1java实现同步的方式

1.同步方法-synchronized关键字

2.同步代码块-synchronized关键字

3.volatile关键字实现同步

4.ReentrantLock可重入锁

ReentrantLock() ： 创建一个ReentrantLock实例

lock() : 获得锁

unlock() : 释放锁

```
class Bank {
        private int account = 100;
        //需要声明这个锁
        private Lock lock = new ReentrantLock();
        public int getAccount() {
            return account;
        }
        //这里不再需要synchronized
        public void save(int money) {
            lock.lock();
            try{
                account += money;
            }finally{
                lock.unlock();
            }
        }
    }
```

5.使用局部变量实现线程同步

```
public class Bank{
        //使用ThreadLocal类管理共享变量account
        private static ThreadLocal<Integer> account = new ThreadLocal<Integer>(){
            @Override
            protected Integer initialValue(){
                return 100;
            }
        };
        public void save(int money){
            account.set(account.get()+money);
        }
        public int getAccount(){
            return account.get();
        }
    }
```

3.2 synchronized 关键字

使用方式

修饰实例方法

修饰静态方法

修饰代码块

synchronized 和 ReentrantLock都是可重入锁

非公平锁,依赖jvm实现

3.3 ReentrantLock

可重入锁

不同于synchronized ,依赖于api实现 lock(),unlock()

可以指定是公平锁还是非公平锁

3.4 volatile 关键字

主要作用:

保持变量的可见性,保证变量在读写的时候,主内存和工作内存保持一致

防止指令重排序,保证有序性

轻量级锁,不会阻塞,主要解决多线程之间变量共享

指令重排序

指令重排序
比如:创建对象的时候,分为三步执行:
1.为xx实例分配内存空间
2.初始化 实例xx
3.将实例指向分配的内存空间
jvm在执行过程中,具有指令重排序的功能,比如1-3-2的执行顺序
单线程环境下,指令重排不会有问题,但是在多线程环境下,比如执行过1-3后,
cpu去执行其他线程指令,比如调用xx实例,这是时候xx实例还没有初始化,就会出现问题

3.5 java并发容器

1. ConcurrentHashMap：并发版HashMap
2. CopyOnWriteArrayList：并发版ArrayList 锁写，不锁读，会出现脏读
3. CopyOnWriteArraySet：并发Set
4. ArrayBlockingQueue：阻塞队列(基于数组)
5. LinkedBlockingQueue：阻塞队列(基于链表)
6. LinkedBlockingDeque：阻塞队列(基于双向链表)

3.6 start()和run()

调用start方法可启动线程进入就绪状态,调用run方法,只不过是在main函数里执行run方法,并不是开启新的线程

3.7 sleep()和wait()

sleep方法不释放锁,wait方法会释放锁

都可以暂停线程的执行

wait通常用于线程间的交互和通信,sleep用于暂停执行

wait()方法需要notify()来唤醒,或者超时自动苏醒,sleep方法执行完成后自动苏醒

3.8 CAS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**

乐观锁常见的2种机制

1. 版本号机制

一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

2. CAS算法

即compare and swap（比较与交换），是一种有名的**无锁算法**。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。**CAS算法**涉及到三个操作数

- 需要读写的内存值 V
- 进行比较的值 A
- 拟写入的新值 B

当且仅当 V 的值等于 A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个**自旋操作**，即**不断的重试**。

乐观锁的问题

1 ABA 问题

2 循环时间长开销大

3 只能保证一个共享变量的原子操作

3.9 Runnable和Callable的区别和用法

1. Runnable执行方法是run(),Callable是call()
2. 实现Runnable接口的任务线程无返回值；实现Callable接口的任务线程能返回执行结果
3. call方法可以抛出异常，run方法若有异常只能在内部消化

3.10 线程池

1 线程池的定义和作用

线程池的基本思想是一种对象池，在程序启动时就开辟一块内存空间，里面存放了众多(未死亡)的线程，池中线程执行调度由池管理器来处理。当有线程任务时，从池中取一个，执行完成后线程对象归池，这样可以避免反复创建线程对象所带来的性能开销，节省了系统的资源。

2.几个核心参数的作用:

```
public ThreadPoolExecutor(
    int corePoolSize,       //核心线程的数量
    int maximumPoolSize,    //最大线程数量（核心线程+核心意以外的线程）
    long keepAliveTime,     //超出核心线程数量以外的线程空闲时的存活时间
    TimeUnit unit,          //存活时间的单位
    //存放来不及处理的任务的队列，是一个BlockingQueue。
    BlockingQueue<Runnable> workQueue,
    //生产线程的工厂类，可以定义线程名，优先级等。
    ThreadFactory threadFactory,
    RejectedExecutionHandler handler // 当任务无法执行时的处理器
    ) {...}
```

3.工作原理

1. 任务请求时,当前池中线程比核心数少，新建一个线程执行任务
2. 如果核心线程池已经满了,而任务队列未满,添加到任务队列
3. 如果核心线程池和任务队列都满了,会增加核心以为的线程进行任务执行
4. 如果线程池到达最大线程数量,无法进行线程创建,交给handler处理

4.handler处理策略

1、CallerRunsPolicy：只要线程池没关闭，就直接用调用者所在线程来运行任务

2、AbortPolicy：直接抛出 RejectedExecutionException 异常

3、DiscardPolicy：悄悄把任务放生，不做了

4、DiscardOldestPolicy：把队列里待最久的那个任务扔了，然后再调用 execute() 试试看能行不

我们也可以实现自己的 RejectedExecutionHandler 接口自定义策略，比如如记录日志什么的

5.jdk提供的线程池

1、newSingleThreadExecutor

1个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。

2、newFixedThreadPool

FixedThreadPool 的核心线程数和最大线程数都是指定值，也就是说当线程池中的线程数超过核心线程数后，任务都会被放到阻塞队列中。(只有核心线程数)

3、newCachedThreadPool

CachedThreadPool 没有核心线程，非核心线程数无上限，也就是全部使用外包，但是每个外包空闲的时间只有 60 秒，超过后就会被回收。

4、newScheduledThreadPool

核心和外包都有，此线程池支持定时以及周期性执行任务的需求。

6.线程池策略

所以**对于任务耗时短的情况，要求线程尽量少，如果线程太多，有可能出现线程切换和管理的时间**，大于任务执行的时间，那效率就低了；

对于耗时长的任务，要分是cpu任务，还是io等类型的任务。如果是cpu类型的任务，线程数不宜太多；但是如果是io类型的任务，线程多一些更好，可以更充分利用cpu。

高并发，低耗时的情况：建议少线程，只要满足并发即可；例如并发100，线程池可能设置为10就可以

低并发，高耗时的情况：建议多线程，保证有空闲线程，接受新的任务；例如并发10，线程池可能就要设置为20；

高并发高耗时：1要分析任务类型，2增加排队，3、加大线程数

4. 反射

4.1 反射定义

反射就是**动态加载对象**，并对对象进行剖析。在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法，这种动态获取信息以及动态调用对象方法的功能成为Java反射机制。

4.2 获取类的方法

1.通过对象获取类

```
Class<String> stringClass = String.class;
```

2.通过实例获取类

```
String str="String";
Class<? extends String> stringClass = str.getClass();
```

3.通过全类名获取类

```
Class<?> aClass = Class.forName("java.lang.String");
```

4.3 反射机制的作用

1)在运行时判断任意一个对象所属的类

2)在运行时构造任意一个类的对象

3)在运行时判断任意一个类所具有的成员变量和方法

4)在运行时调用任意一个对象的方法

4.4反射的优缺点

优点

- 反射提高了程序的灵活性和扩展性,在底层框架中用的比较多，业务层面的开发过程中尽量少用。

缺点

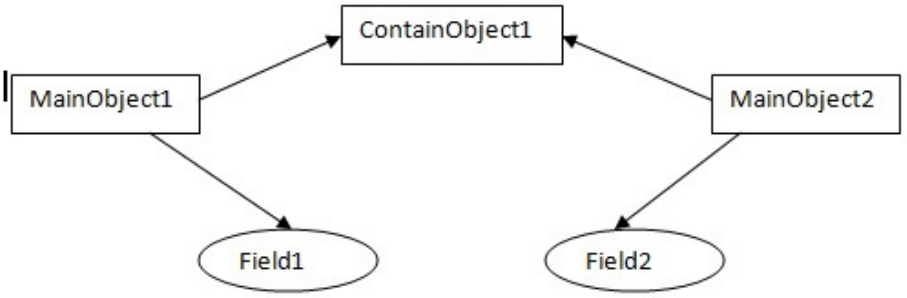
- 性能不好 反射是一种解释操作,用于字段和方法接入时要远慢于直接代码，下面通过2段简单的代码来比较下执行的时间就可以体现出性能的问题

5.深拷贝和浅拷贝

如果在拷贝这个对象的时候，只对基本数据类型进行了拷贝，而对引用数据类型只是进行了引用的传递，而没有真实的创建一个新的对象，则认为是**浅拷贝**。

反之，在对引用数据类型进行拷贝的时候，创建了一个新的对象，并且复制其内的成员变量，则认为是**深拷贝**。

1、浅拷贝：对基本数据类型进行值传递，对引用数据类型进行引用传递般的拷贝，此为浅拷贝。



2、深拷贝：对基本数据类型进行值传递，对引用数据类型，创建一个新的对象，并复制其内容，此为深拷贝。