

1

StringBuilder

- 

```
StringBuilder sb = new StringBuilder();
//     
sb.append("aa");
//     
sb.reverse();
//       
String str = sb.toString();
```

ArrayList

- **java.lang.Object**(**Object**)
 - **java.lang.Number**(**Number**)
 - **java.lang.Integer**(**int**)
 - **java.lang.Double**(**double**)
 - **java.lang.Character**(**char**)
 - **java.lang.String**(**String**)

```
// ArrayList<String>
ArrayList<String> list = new ArrayList<String> ();
// 添加
list.add("1234");
// 移除
list.remove("1234");
// 替换
String str = list.remove(1);
// 获取
String str = list.get(1);
// 替换
String str = list.set(1,"3445");
// 大小
int size = list.size();
```

static

- `function`
 - `return`
 - `if`
 - `this`

11

-  **extends**

```
public class Student extends Person{}
```

- Student Person

- java 语言的访问权限
 - 语义
 - 私有
 - private\static\final 修饰符
 - 语义
super() 调用父类构造方法

1

- **private** | **static** | **final** |
 - **final** | **final** | **final** |

super

1

- `System.out.println(name)`         
 - `System.out.println(this.name)`         
 - `System.out.println(super.name)`         

1

- @Override
 - @Override

```
@Override // 重写  
public void eat(){  
    sout('吃饭了');  
}
```

1

- မြန်မာစာတမ်း
 - မြန်မာစာတမ်း

```
Animal a = new dog();
System.out.println(a.name); // 狗的Animal对象name
a.show(); // 狗对象show
```

1

- 

1

- instanceof
- instanceof

```
Animal a = new dog();
Dog d = (Dog) a;// instanceof
if (a instanceof Dog)// instanceof
```

final

- final
- final
- final

抽象

- abstract
- 抽象
- 抽象
- 抽象

```
// 抽象
abstract class Animal {
    String name; // 抽象

    // 抽象
    public abstract void eat();

    // 抽象
    public void sleep() {
        System.out.println(name + " 睡觉");
    }

    // 抽象
    public Animal(String name) {
        this.name = name;
    }
}

// 抽象
class Dog extends Animal {
    public Dog(String name) {
        super(name); // 抽象
    }

    @Override
    public void eat() {
        System.out.println(name + " 吃饭");
    }
}
```

□□

- Java Interface Java 语言中实现接口的语法
 - 实现接口的类 implements 接口
 - 定义接口的语法
- ```
□ interface 接口名
```
- 定义抽象方法 public abstract 方法名
  - 定义常量 public static final 常量名
  - 其他方法

```
// □□
public interface Animal {
 void eat();
}

// □□
public interface Fly {
 void fly();
}

// □□
public class Bird implements Animal, Fly {
 @Override
 public void eat() {
 System.out.println("□□");
 }

 @Override
 public void fly() {
 System.out.println("□□");
 }
}

// □□
public class Main {
 public static void main(String[] args) {
 Bird bird = new Bird();
 bird.eat(); // □□
 bird.fly(); // □□
 }
}
```

## □□

- 实现接口
- 定义抽象类和抽象方法

```
// クラス
class car{
 string name;
 class enigne{
 string name;
 }
}

car.enigne e = new car().new enigne(); // エンジン
System.out.println(e.name);
```

## まとめ

- ・ クラス内構造体

## まとめ

- ・ クラス内構造体

```
o class Outer{
 string name;
 class Inner{
 string name;
 }
 public Inner getInner(){
 return new Inner(); // インナー
 }
}
// new Outer() インナーオブジェクト
Outer.Inner oi = new Outer().getInner();
```

- ・ クラス内構造体

- Outer.Inner.インスタンス = Outer.インスタンス
- Outer.Inner oi = new Outer().new Inner();

## まとめ

- static フィールド
- クラス内構造体のインスタンス化
- クラス内構造体
  - Outer.Inner.インスタンス = new Outer.インスタンス()
  - Outer.Inner oi = new Outer.Inner()
- クラスメソッド
  - Outer.Inner.インスタンス.メソッド()
  - Outer.Inner.show()

```
class Outer {
 private String name; // String タイプのフィールド
 private static int age; // age フィールド static キーワード
```

```

// Outer
static class Inner {
 private String name; // String なまえ

 // Outerのage
 public void getOuterAge() {
 System.out.println(age); // Outerのage
 }

 // Outerのname
 public void getOuterName() {
 System.out.println(new Outer().name); // OuterのOuterのname
 }
}

public class Main {
 public static void main(String[] args) {
 Outer.Inner inner = new Outer.Inner();
 inner.getOuterAge(); // Outerのage 0
 inner.getOuterName(); // Outerのname null
 }
}

```

## まとめ

- ・ クラス内構造
- ・ メンバの親子関係
- ・ メンバの親子関係

```

class Outer{
 int b = 20;
 public void show(){
 int a = 10;
 // Outer
 class Inner{
 String name;
 int age;
 public void method1(){
 System.out.println(a);
 System.out.println(b);
 }
 // Inner
 Inner i = new Inner();
 System.out.println(i.name);
 System.out.println(i.age);
 }
}

```

java

java 语言中抽象类的实现

```
public class abstract Animal{
 public abstract void cry();
}

Animal a = new Animal(){
 @Override
 public void cry(){
 System.out.println("----");
 }
};

a.cry();
```

## API

### String

- 字符串
- new

### System

java

| 方法                                                           | 描述       |
|--------------------------------------------------------------|----------|
| public static void exit(int status)                          | 退出Java程序 |
| public static long currentTimeMillis()                       | 当前时间毫秒数  |
| public static void arraycopy(源数组, 源起始索引, 目标数组, 目标起始索引, 复制长度) | 数组复制     |

status 退出状态

- 0 正常退出
- 1 异常退出
- 2 程序错误
- 1 未知退出

```
import java.util.Arrays;

public class SystemAPIExample {
 public static void main(String[] args) {
 // 1. 退出
 long startTime = System.currentTimeMillis();
 System.out.println("开始时间" + startTime + " ms");

 // 2. 退出
 }
}
```

```

int[] src = {1, 2, 3, 4, 5}; // 1 2 3 4 5
int[] dest = new int[5]; // 0 0 0 0 0

System.arraycopy(src, 1, dest, 2, 3);
// src[1] 2 3 4 dest[2] 3 4 5

System.out.println("输出结果" + Arrays.toString(dest));

// 3. 退出
int status = 1; // 程序正常结束
System.out.println("输出结果" + status);
System.exit(status);

// 程序正常结束 System.exit() 退出 JVM
System.out.println("输出结果");
}
}

```

## Runtime

| 方法                                  | 描述            |
|-------------------------------------|---------------|
| public static getRuntime()          | 返回 Runtime 对象 |
| public long maxMemory()             | JVM 可用内存      |
| public long totalMemory()           | JVM 总内存       |
| public long freeMemory()            | JVM 未使用内存     |
| public Process exec(String command) | 执行命令          |
| public int availableProcessors()    | 可用 CPU 核数     |
| public void exit(int status)        | JVM 退出状态      |

## Object

| 方法                                | 描述      |
|-----------------------------------|---------|
| public String toString()          | 对象字符串表示 |
| public boolean equals(Object obj) | 对象相等性比较 |
| protected Object clone(int a)     | 克隆对象    |

- `toString` 和 `equals` 方法位于 `Object` 类中
- `clone` 方法位于 `Object` 类中，实现了 `Cloneable` 接口的 `clone` 方法。在 `Object` 类中实现的 `clone` 方法是浅拷贝。
- 深拷贝实现：通过重写 `clone` 方法并调用 `Object.clone(true)` 实现。
- 通过 `Object.clone(true)` 调用父类的 `clone` 方法。

## BigInteger

| 方法名                                     | 功能                                 |
|-----------------------------------------|------------------------------------|
| public BigInteger(int num, Random rnd)  | 生成指定范围内的随机数[0~2 <sup>num</sup> -1] |
| public BigInteger(String val)           | 字符串转为大整数                           |
| public BigInteger(String val,int radix) | 字符串转为大整数                           |

大整数类

## Lambda

- Lambda表达式
- 简化了匿名内部类的使用，实现了FunctionalInterface

```
interface Swim{
 void swimming();
}

// 传统写法
Swim s1 = new Swim(){
 @Override
 public void swimming(){
 System.out.println("游泳");
 }
}

/*
lambda写法
(参数) ->{
 语句
}
*/

// 传统写法
Swim s1 = ()->{
 System.out.println("游泳");
}
```

Lambda表达式优势

- 方便简洁
- 可以直接调用方法体"()"或方法名
- 在Lambda表达式中使用return语句时，不能使用return语句

```
// 按钮btn
btn.addActionListener(new ActionListener(ActionEvent e) ->{
 System.out.println("按钮");
})

// 传统写法

btn.addActionListener((ActionEvent e) ->{
```

```

 System.out.println("按钮1");
 })

btn.addActionListener(e ->{
 System.out.println("按钮2");
})

btn.addActionListener((e) ->{
 System.out.println("按钮3");
})

btn.addActionListener(e ->{
 System.out.println("按钮4");
})

btn.addActionListener(e-> System.out.println("按钮5"))

```

按钮1

- 按钮2

按钮3按钮4->按钮5

按钮 1::按钮2

```

// Student按钮1
public static int compare(studnets o1,students o2){
 // 按钮2
 return o1.getAge() - o2.getAge();
}

Arrays.sort(students,(o1,o2) -> o1.getAge() - o2.getAge());

Arrays.sort(students,(o1,o2) -> Student.compare(o1,o2));
//按钮3
Arrays.sort(students,Student::compare)

```

- 按钮4

按钮1按钮2->按钮3

按钮 1::按钮2

```

// Student按钮1
public int compareheight(studnets o1,students o2){
 // 按钮2
 return o1.getHeight() - o2.getHeight();
}

Student student = new Student();

Arrays.sort(students,(o1,o2) -> o1.getHeight() - o2.getHeight());

```

```
Arrays.sort(students,(o1,o2) -> student.compareheight(o1,o2));
//
Arrays.sort(students,student::compareheight)
```

- **Lambda** **方法引用**
- **方法引用**

方法引用**Lambda**方法引用"->"方法引用

方法 方法::new

方法

- **方法引用** **方法名**
- **方法引用** **方法名**
- **方法引用** **方法名**
  - **参数int|double** **方法名****Integer|Double**
  - **方法名**

```
// 方法
// -126~127
Integer i1 = Integer.valueOf(100);
Integer i2 = Integer.valueOf(12);

// 方法
Integer i1 = 100;

// 方法
int i = i1;

// 方法
// 100~1000000000
String str1 = Integer.toString(20);

// 200~1000000000
String str = "23";
int i1 = Integer.parseInt(str);
```

方法

```
// 方法
public class myArrayList<E>{
 private ArrayList list = new ArrayList();

 public boolean(E e){
 list.add(e);
 return true;
```

```
 }

 public void remove(E e){
 return list.remove(e);
 }
}
```

## ArrayList

- ArrayList

```
// ArrayList
public interface Data <T>{
 void add(T t);
 void delete(T t);
}
```

## ArrayList实现类

- ArrayList

```
/*
 * <ArrayList...> 实现类 例 (ArrayList){}

// ArrayList
public E get(int index){ // 返回元素
 return (E)arr[index];
}

// ArrayList
public static <T> T test(T t){

}
```

- ArrayList
  - "?"表示泛型约束
- ArrayList
  - ?extends car 表示子类car的集合
  - ?super car 表示父类car的集合

```
// ArrayList
public static void go(ArrayList <?> cars){
}
```

## Collection(集合)



## Collection

- List
- Set

接口

| 方法                                  | 功能       |
|-------------------------------------|----------|
| public boolean add(E e)             | 添加元素     |
| public void clear()                 | 清空集合     |
| public boolean remove(E e)          | 移除元素     |
| public boolean contains(Object obj) | 判断是否包含   |
| public boolean isEmpty()            | 判断是否为空   |
| public int size()                   | 返回集合大小   |
| public Object[] toArray()           | 将集合转换为数组 |

实现类

- CollectionIterator

```
Collection<String> names = new ArrayList<>();
names.add("1");
names.add("2");
names.add("3");
names.add("4");
names.add("5");

// 遍历
Iterator<String> it = names.iterator();
// 移除
it.next();
// 打印
it.remove();

// 循环
while(it.hasNext()){
 String ele = it.next();
 System.out.println(ele);
}
```

- for

```

/*
for(;;){
}

// []
for(String s : c){
 System.out.println(s);
}

```

- **Lambda**
  - **foreach**

```

Collection<String> names = new ArrayList<>();
names.add("1");
names.add("2");
names.add("3");
names.add("4");
names.add("5");

names.forEach(new Consumer<String>(){
 public void accept(String s){
 System.out.println(s);
 }
});

// []
names.forEach((String s)->{
 System.out.println(s);
});

// []
names.forEach(s->System.out.println(s));

```

## List

接口

| 方法                            | 描述               |
|-------------------------------|------------------|
| void add(int index,E element) | 在指定索引处插入元素       |
| E remove(int index)           | 移除指定索引处的元素       |
| E set(int index,E element)    | 将指定索引处的元素替换为新的元素 |
| E get(int index)              | 获取指定索引处的元素       |

实现类 List <String> names = new ArrayList()

## ArrayList

- 亂序
- **插入时间复杂度O(1)** 删除时间复杂度 $O(n)$
- **空间复杂度1.5倍**

## LinkedList

- **插入和删除时间复杂度O(1)**

线性表

| 方法                        | 功能   |
|---------------------------|------|
| public void addFirst(E e) | 头部插入 |
| public void addLast(E e)  | 尾部插入 |
| public E getFirst()       | 头部取出 |
| public E getLast()        | 尾部取出 |
| public E removeFirst()    | 头部移除 |
| public E removeLast()     | 尾部移除 |

## Set

无序不重复

集合Collection

### HashSet

- 乱序
- 插入和删除时间复杂度 $O(1)$ 
  - **16进制数的基数0.75**  $16 * 0.75 = 12$  表示桶数20
  - **null**
  - **null**
  - **8进制数64**
- int hashCode() 方法

### LinkedHashSet

- 有序
- 插入和删除时间复杂度 $O(n)$

### TreeSet

- 有序
- **TreeSet**
  - Comparable Comparable 比较器 compareTo 方法重写`this > o`

```
public int compareTo(Teacher o){
 if(this.getAge() > o.getAge()) return 1;
```

```
 if(this.getAge() < o.getAge()) return -1;
 return 0;
}
```



2. TreeSet의 Comparator는 Comparable을 implements하는 new로 Comparator를 정의하는 compare 메서드를 정의

```
Set<Teacher> teachers = new TreeSet<>(new Comparator<Teacher>(){
 @Override
 public int compare(Teacher o1, Teacher o2){
 return o1.getAge() - o2.getAge(); //升序
 }
});

Set<Teacher> teachers = new TreeSet<>((o1,o2)->{
 return o1.getAge() - o2.getAge();
});
```

- 

## Map(□□□□)

1

-  Set
  -  Collection

|                                            |                                 |
|--------------------------------------------|---------------------------------|
| public void put(K key, V value)            | 将键值对存入Map                       |
| public int size()                          | 返回Map中键值对的个数                    |
| public void clear()                        | 清空Map                           |
| public boolean isEmpty()                   | 判断Map是否为空<br>true:空<br>false:不空 |
| public V get(Object key)                   | 根据键获取值                          |
| public V remove(Object key)                | 根据键删除键值对                        |
| public boolean containsKey(Object key)     | 判断键是否存在                         |
| public boolean containsValue(Object value) | 判断值是否存在                         |
| public Set<K> keySet()                     | 返回所有键的集合                        |
| public Collection<V> values()              | 返回所有值的集合                        |



## Map

- Map

### HashMap

- HashMap
- new HashSet[] new HashMap[]

### LinkedHashMap

- LinkedHashMap
- new LinkedHashSet[]

### TreeMap

- TreeMap
- TreeSet

```
// TreeMap
Map<Teacher, String> map = new TreeMap<>((o1, o2)->{
 return o2.getAge() - o1.getAge();
});

// TreeSet
Map<Teacher, String> map = new TreeMap<>((o1, o2)->{
 return Double.compare(o2.getSalary()-o1.getSalary());
});
```

## Map

- Map

```
Map<String, Integer> map = new Hashmap<>();
// 1. keySet
Set<Integer> keyset = map.keySet();
// 2. value
for(String key: keyset){
 Integer value = map.get(key);
 System.out.println(value);
}
```

- entrySet
- entrySet() map.entrySet() set
- Map.Entry<K, V> getKey() getValue()

```
Map<String, Integer> map = new Hashmap<>();
Set<Map.Entry<K, V>> sets = map.entrySet();

for(Map.Entry<K, V> set: sets){
```

```
String key = set.getKey();
Integer value = set.getValue();
System.out.println(key + ":" + value);
}
```

- Lambda  JDK8 



```
map.forEach(new BiConsumer<String, Integer>(){
 @Override
 public void accept(String key, Integer value){
 System.out.println(key + ":" + value);
 }
});

// 二
map.forEach((k,v) -> {
 System.out.println(k + ":" + v);
});
```

## Stream

□ □ □ □ □ □ □ □ □ □

- `Stream`
  - `Stream .filter()` `Stream`
  - `Stream .stream()` `Stream`
  - `Map` `Map.entrySet()` `Set`
    - `Map.Entry<K,V>` `map` `set`

```
// ȿȿȿȿ
Set<K> keySet = map.keySet(); // ȿȿmapȿȿȿȿ
Stream<K> keyStream = keySet.stream(); // ȿȿȿȿ

// ȿȿȿȿ
Collection<V> values = map.values(); // ȿȿmapȿȿȿȿ
Stream<V> valueStream = values.stream();

// ȿȿȿȿȿȿ
Set<Map.Entry<K, V>> entrySet = map.entrySet(); // ȿȿmapȿȿȿȿȿȿsetȿȿ
Stream<Map.Entry<K, V>> entryStream = entrySet.stream();
```

- `Arrays.stream()`  $\Rightarrow$  `stream.of()`  $\Rightarrow$  `stream()`
    - Stream API

```

Integer []ages = {1,3,46,23,56,18};

Stream<Integer> stream = Arrays.stream(ages);

Stream<Integer> stream = Stream.of(ages);

```

Stream API

方法

| 方法                             | 描述     |
|--------------------------------|--------|
| Stream <T> filter(Predicate)   | 过滤流    |
| Stream <T> sorted()            | 排序流    |
| Stream <T> sorted(Comparator)  | 自定义排序流 |
| Stream <T> limit(long maxSize) | 限制流    |
| Stream <T> skip(long n)        | 跳过流    |
| Stream <T> distinct()          | 去重流    |
| Stream <T> map(Function)       | 映射流    |

```

// scores를 Stream으로 map하는 예제: 10씩 더해주는 예제
scores.stream().map(s -> "10" + (s + 10));

```

Collector

方法

Stream API

| 方法                                         | 描述             |
|--------------------------------------------|----------------|
| void forEach(Consumer action)              | 对流中的元素执行操作     |
| long count()                               | 统计流中的元素数量      |
| Optional<I> max(Comparator)                | 找到流中最大的元素      |
| Optional<I> min(Comparator)                | 找到流中最小的元素      |
| R collect(Collector collector)             | 将流中的元素收集到一个容器中 |
| Object[] toArray()                         | 将流中的元素转换为数组    |
| public static <T> Collector<T> toList()    | 将流转换为列表        |
| public static <T> Collector<T> toSet()     | 将流转换为集合        |
| public static Collector<T> toMap(Function) | 将流转换为映射        |

.reduce(

Optional<Teacher>

- max/min Optional null
- Optional .get()
- reduce

```
// teachers
//
Optional<Teacher> max = teachers.stream().max((t1,t2)-
>Double.compare(t1.getSalary(),t2.getSalary()));
//
Teacher maxteacher = max.get();

// s
// list
List<User> list = s.collect(Collector.toList());
// set
Set<User> set = s.collect(Collector.toSet());
//
Object[] array = s.toArray();
// map
Map<User> map = s.collect(toMap(t->t.getName(),t->getSalary()));
```

## File

File

```
//
File f1 = new File("");

//
f1.length();

//
f1.getName();

//
f1.isFile();

//
f1.isDirectory();

//
f1.createNewFile();

//
f1.mkdir();

//
f1.mkdirs();
```

```
// 删除文件(目录)
f1.delete();

// 列出目录,文件
f1.list();

// 列出所有File对象
f1.listFiles();

// 绝对路径
f1.getAbsolutePath();
```

## 字符

### 编码

- ASCII 0~127 8bit
- GBK 128~255 ASCII 128~255
- 128~199 ASCII 128~199
- Unicode 16bit

### UTF-8

- UTF-8 1~4字节  
◦ 1 2 3 4

## 线程

### 线程

#### Thread

- run run run
- start

```
public class Demol extends Thread{
 @Override
 public void run(){
 System.out.println("线程");
 }
}

public class Test{
 public static void main(String[] args){
 Thread t1 = new Demol();
 // 启动
 t1.start();
 }
}
```

#### Runnable

- `run`
- `Runnable r` `Thread`
- `start`

```
// 1
public class Demo2 implements Runnable{
 @Override
 public void run(){
 System.out.println("1");
 }
}

public class Test{
 public static void main(String[] args){
 // 2
 Runnable r = new Demo2();
 // 3
 Thread t2 = new Thread(r);
 // 4
 t2.start();
 }
}

// 5
public class Test{
 public static void main(String[] args){
 // 6
 Runnable r = ()->{
 System.out.println("5");
 };
 // 7
 Thread t2 = new Thread(r);
 // 8
 t2.start();
 }
}
```

## Callable

- `Callable`
- `Callable` `call`
- `call`
- `FutureTask`
- `call`
- `FutureTask` `Runnable`
- `Thread`
- `Thread start`
- `FutureTask get`

```

public class Demo3 implements Callable<Integer>{
 @Override
 public Integer call(){
 int a = 0;
 for(int i = 1;i <= 10;i++){
 a += i;
 }
 return a;
 }
}

public class Test{
 public static void main(String[] args){
 // ...
 Callable<Integer> c = new Demo3();
 // ...
 FutureTask<Integer> f = new FutureTask<>(c);
 Thread t2 = new Thread(f);
 // ...
 t2.start();
 // ...
 System.out.println(f.get());
 }
}

```

线程池

- public String getName() 线程名称
- public void setName(String name) 线程名称
- public static Thread currentThread() 线程对象
- public final void join() 等待线程结束

线程池

线程池的线程复用

线程池的线程复用

线程池

线程池的线程复用

```

synchronized(对象){
 //...
}

```

- 同步语句块
- 同步方法
  - this' 对象
  - 同步方法所在类 .class 对象

线程

- `java -jar` this `.jar`
  - `javac -d` `src`.`class` `main`

```
 synchronized void run() {
 System.out.println("...");
 }
}
```

**Lock**

Lock ReentrantLock

- void lock() 
  - void unlock() 

```
public final Lock l1 = new ReentrantLock();
```

三

## ExecutorService

8 / 8

1. `ExecutorService` `ThreadPoolExecutor` `new ThreadPoolExecutor`
  2. `Executors` `new ThreadPoolExecutor`

## ThreadPoolExecutor

- Runnable

```
public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long
keepAliveTime,TimeUnit unit,BlockingQueue <Runnable> workQueue,ThreadFactory
threadFactory,RejectedExecutionHandler handler);

// 例程
ExecutorService pool = new ThreadPoolExecutor(3,5,10,TimeUnit.SECONDS,new
ArrayBlockingQueue<>(3),Executors.defaultThreadFactory(),new
ThreadPoolExecutor.AbortPolicy());
```

- corePoolSize ۰۰۰۰۰۰۰۰۰۰۰
  - maximumPoolSize ۰۰۰۰۰۰۰۰۰۰۰
  - keepAliveTime ۰۰۰۰۰۰۰۰۰۰۰
  - unit ۰۰۰۰۰۰۰۰۰۰۰
  - workQueue ۰۰۰۰۰۰۰۰۰۰۰
  - threadFactory ۰۰۰۰۰۰۰۰۰۰۰
  - handler ۰۰۰۰۰۰۰۰۰۰۰

## ExecutorService 介绍

- void execute(Runnable command) — Runnable

- `Future<T> submit(Callable<T> task)` 用于 Callable 任务的提交
- `void shutdown()` 停止线程池的运行
- `List<Runnable> shutdownNow()` 停止线程池的运行\*\*返回队列\*\*

线程池的关闭

- `AbortPolicy` 强制中断线程

线程池的关闭

- `DiscardPolicy` 强制丢弃线程

线程池的关闭

- `AbortPolicy` 强制中断线程
- `DiscardPolicy` 强制丢弃线程
- `DiscardOldestPolicy` 强制丢弃最老的线程
- `CallerRunsPolicy()` 线程池的线程在 run 方法中运行

线程池

线程CPU使用情况的统计

线程池的线程数CPU使用情况

## IO

线程池的IO操作

- `submit(Runnable task)`
- `submit(Callable<T> task)`

线程池的IO操作

- `read`
- `write`

线程池

线程池的线程数CPU使用情况

## Junit

- `assertEqual` 断言两个对象相等
- `assertNotEqual` 断言两个对象不相等
- `@Test` 测试注解

线程

线程池的线程数Class

- `Thread.currentThread().getStackTrace()`

线程池的线程数Class

### 1. 线程池的线程数Class

- `Class c1 = .class`
- `Class.forName(String package)` package

- Class clazz = Class.forName("Student")
- Class c1 = clazz.getClass()

2. **构造器Constructor**

3. **字段Field**

4. **方法Method**

```
// 构造器
Class c1 = Student.class; // 获取类对象
System.out.println(c1.getName()); // 获取类名
System.out.println(c1.getSimpleName()); // 简单类名

// 字段
Constructor [] cons = c1.getDeclaredConstructors(); // 获取构造器
for(Constructor con: cons){
 System.out.println(con.getName() + "("+con.getParameterCount()+"")"); // 打印构造器参数个数
}

// 方法
Constructor con2 = c1.getDeclaredConstructor(String.class,int.class); // 获取方法

// 字段
Field field = c1.getDeclaredField("hobby");
System.out.println(field.getName() + "("+field.getType().getName()+"")");

// 方法
Method method = c1.getDeclaredMethod("eat"); // 获取方法eat
Method method1= c1.getDeclaredMethod("eat",String.class); // 获取方法eat,参数String
System.out.println(method.getName() + "("+method.getParameterCount()+"")");
```

构造器的使用

```
Class c1 = Student.class; // 获取类对象
Constructor con = c1.getDeclaredConstructor(); // 获取构造器

// 构造器
con.Accessible(true); // 设置构造器为可访问
Student c1 = (Student) con.newInstance(); // 调用构造器
System.out.println(c1);
```

方法的使用

```
Student c1 = new Student("张三",12); // new
Field field = c1.getDeclaredField("hobby"); // 获取字段
field.setAccessible(true);
field.set(c1,"篮球"); // c1赋值
```

```
String hobby = (String) field.get(c1); // 获取 c1.hobby
```

1

```
Method method = c1.getDeclaredMethod("eat");
Student c1 = new Student("王五",12); // new对象

method.setAccessible(true);
method.invoke(c1); // 调用方法，相当于 c1.eat() 打印输出
```

5

- မြန်မာစာတမ်းပေါ်မှု
  - မြန်မာစာ
  - မြန်မာစာတမ်းပေါ်မှု**(class)** မြန်မာစာတမ်းပေါ်မှု

```
ArrayList <String> list = new ArrayList<>();
list.add("二");
list.add("三");
list.add(88); // 二

// 二三八
Class c1 = list.getClass();
Method m1 = c1.getDeclaredMethod("add",Object);
m1.invoke(list,11); // 二三
m1.invoke(list,true); // 二三
```

1

- `@Override` `@Test`
  - `public void`
  - `private void`

1

- **value**

```
/*
 public @interface Mybook{
 public String value() default " ";
 }

 public @interface Mybook{
 String value() ;
 }
*/
```

```
}
@Mybook("delete") // ลบ
@Mybook(value = "delete") // ลบ
```

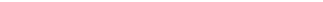
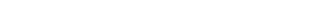
1

10

- @Target  
◦ TYPE  
◦ FIELD  
◦ METHOD  
◦ PARAMETER  
◦ CONSTRUCTOR  
◦ LOCAL\_VARIABLE
  - @Retention  
◦ RUNTIME

```
@Retention(RetentionPolicy.RUNTIME) // 保留策略
@Target({ElementType.METHOD})
public @interface Mybook{ // 定义注解
 String value();
}
```

1

- 
  - 

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
public @interface Mybook{ // 书本
 String value();
 double height () default 100;
 String []address();
}

} // 定义注解

@Mybook(value ="中文",address = {"北京", "上海"})
public class Demo{

}

// 定义类

Class c1 = Demo.class; // Demo类
if(c1.isAnnotationPresent(Mybook.class)) { // 判断Mybook注解是否
 // 存在
 Mybook b = (Mybook) c1.getAnnotation(Mybook.class);
 // 取出
 String []address = b.address();
}
```

```
 double height = b.height();
 String value = b.value();
 }
```

- 
- 
- 
- - 
  - 
  - -

```
public interface Mess{
 public void sing(String name);
 public String dance();
}
```

```
@AllArgsConstructor
public class Real implements Mess{
 private String name;
 @Override
 public void sing(String name){
 System.out.println(this.name + " " + name);
 }
 @Override
 public String dance(){
 System.out.println(this.name + " 亂世Style");
 return " 亂世 ";
 }
}
```

```
// 亂世
import java.lang.reflect.Proxy
// 亂世
import java.lang.reflect.InvocationHandler
```

```

public class ProxyUtil{
 public static Mess createproxy(Real r)){
 // ...
 // ...
 // ...
 Mess m = (Mess)Proxy.newProxyInstance(
 ProxyUtil.class.getClassLoader(),
 r.getClass().getInterfaces(),
 new InvocationHandler(){
 @Override
 public Object invoke(Object proxy , Method method,Object []args)
throws Throwable{

 // proxy...
 // method...
 // args...
 // ...
 System.out.println("调用方法");
 Object result = method.invoke(r,args);

 // ...
 System.out.println("方法返回值");
 return result;
 }
 });
 return m;
 }
}

```

。。。

```

public class Test(){
 public static void main(String []args){
 Real r = new Real("真实");
 // ...
 Mess proxy = ProxyUtil.createProxy(r);
 // ...
 proxy.sing("唱歌");
 System.out.println(proxy.dance());
 }
}

```