

StringBuilder

StringBuilder

- 문자열을 변경할 수 있는 클래스

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

ArrayList

- 문자열, 정수, 실수, 객체 등 다양한 데이터 타입을 저장할 수 있는 클래스 (Integer, Character)
- 문자열
- 정수

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

static

- 클래스에 속한 변수, 메서드, 블록 등
- 클래스에 속한 변수
- 클래스에 속한 메서드
- 클래스에 속한 블록

extends

- 클래스 extends

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

- Student 클래스는 Person 클래스를 상속받는다

- java `String` 클래스는 `String` 클래스를 상속받는다
- `String` 클래스는 `String` 클래스를 상속받는다
 - `String` 클래스
 - `private static final` 클래스
- `String` 클래스는 `String` 클래스를 상속받는다 `super()` 메서드를 호출한다

클래스

- 클래스는 `private static final` 클래스
- 클래스는 `String` 클래스를 상속받는다

super

클래스

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

클래스

- 클래스는 `String` 클래스를 상속받는다
- 클래스는 `@Override` 클래스

```
@Override // 클래스
public void eat(){
    sout('클래스');
}
```

클래스

클래스는 `String` 클래스를 상속받는다

- 클래스는 `String` 클래스를 상속받는다
 - 클래스
- 클래스는 `String` 클래스를 상속받는다
 - 클래스

```
Animal a = new dog();
System.out.println(a.name); // 클래스Animal 클래스name
a.show(); // 클래스dog 클래스show
```

클래스

- 클래스는 `String` 클래스를 상속받는다

클래스

- `instanceof` 연산자
- `instanceof` 연산자

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

final

- `final` 변수
- `final` 메서드
- `final` 클래스

abstract

- `abstract` 클래스
- `abstract` 메서드
- `abstract` 클래스
- `abstract` 클래스

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

    // eat
    public abstract void eat();

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

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

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

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

❏

- `interface`는 Java에서 클래스와 유사하게 사용할 수 있는 구조
- 클래스는 `implements` 키워드로 인터페이스를 구현
- 인터페이스는 추상적인 메서드와 속성을 정의
- `interface` 키워드로 인터페이스를 선언
- 클래스는 `public abstract` 키워드로 인터페이스를 구현
- 클래스는 `public static final` 키워드로 인터페이스를 구현
- 인터페이스는 추상적인 메서드와 속성을 정의

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

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

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

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

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

❏

- 인터페이스
- 인터페이스를 구현하는 클래스

```
// 测试
class car{
    string name;
    class engine{
        string name;
    }
}

car.engine e = new car().new engine(); // 创建engine
System.out.println(e.name);
```

测试

- 测试

测试

- 测试

- ```
class Outer{
 string name;
 class Inner{
 string name;
 }
 public Inner getInner(){
 return new Inner(); // 创建Inner
 }
}
// 创建 Outer()
Outer.Inner oi = new Outer().getInner();
```

- 测试
  - 测试.测试.测试 = 测试.测试
  - 测试 Outer.Inner oi = new Outer().new Inner();

测试

- 测试
- 测试
- 测试
  - 测试.测试.测试 = new 测试.测试()
  - 测试 Outer.Inner oi = new Outer.Inner()
- 测试
  - 测试.测试.测试()
  - 测试 Outer.Inne.show()

```
class Outer {
 private String name; // String 测试
 private static int age; // age 测试 static 测试
```

```

// 静态内部类
static class Inner {
 private String name; // String 静态内部类

 // 静态内部类 age
 public void getOuterAge() {
 System.out.println(age); // 静态内部类
 }

 // 静态内部类 name
 public void getOuterName() {
 System.out.println(new Outer().name); // 静态内部类 Outer 静态
 }
}

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

```

静态内部类

- 静态内部类
- 静态内部类
- 静态内部类

```

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

```

예제

아래 코드를 실행하면 다음과 같은 결과가 출력됩니다.

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

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

API

String

- `"..."` 문자열 리터럴을 사용하여 문자열을 생성합니다.
- `new` 키워드를 사용하여 문자열 객체를 생성합니다.

System

출력

| 메서드                                                                               | 설명                         |
|-----------------------------------------------------------------------------------|----------------------------|
| <code>public static void exit(int status)</code>                                  | 현재 실행 중인 Java 프로그램을 종료합니다. |
| <code>public static long currentTimeMillis()</code>                               | 현재 시간(밀리초)을 반환합니다.         |
| <code>public static void arraycopy(원본 배열, 시작 인덱스, 대상 배열, 시작 인덱스, 복사할 요소 수)</code> | 배열 복사                      |

`status` 값은 다음을 나타냅니다.

- `0`: 성공적으로 종료되었습니다.
- `1`: 오류가 발생했습니다.
- `2`: 강제 종료되었습니다.
- `-1`: 알 수 없는 이유로 종료되었습니다.

```
import java.util.Arrays;

public class SystemAPIExample {
 public static void main(String[] args) {
 // 1. 시간 출력
 long startTime = System.currentTimeMillis();
 System.out.println("시작 시간: " + startTime + " ms");

 // 2. 배열 복사
```

```

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

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

System.out.println("Arrays.toString(dest)");

// 3. status
int status = 1; // status
System.out.println("status" + status);
System.exit(status);

// status System.exit() JVM
System.out.println("status");
}
}

```

### Runtime

| Method                              | Description      |
|-------------------------------------|------------------|
| public static getRuntime()          | Runtime object   |
| public long maxMemory()             | JVM max memory   |
| public long totalMemory()           | JVM total memory |
| public long freeMemory()            | JVM free memory  |
| public Process exec(String command) | exec             |
| public int availableProcessors()    | CPU count        |
| public void exit(int status)        | JVM exit         |

### Object

| Method                            | Description |
|-----------------------------------|-------------|
| public String toString()          | toString    |
| public boolean equals(Object obj) | equals      |
| protected Object clone(int a)     | clone       |

- toString equals 方法 Object
- clone 方法 Cloneable 接口 clone 方法 Object
  - Cloneable
  - clone

### BigInteger



| 구현                                                    | 비고                    |
|-------------------------------------------------------|-----------------------|
| <code>public BigInteger(int num, Random rnd)</code>   | 0~2 <sup>num</sup> -1 |
| <code>public BigInteger(String val)</code>            | 10진법                  |
| <code>public BigInteger(String val, int radix)</code> | radix진법               |

구현

## Lambda

- Lambda
- `FunctionalInterface`

```
interface Swim{
 void swimming();
}
// 구현
Swim s1 = new Swim(){
 @Override
 public void swimming(){
 System.out.println("수영");
 }
}

/*
lambda
() -> {
 }
*/

// 구현
Swim s1 = () -> {
 System.out.println("수영");
}
```

구현

- 
- 
- Lambda

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

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

□□□□

- Lambda**"->"

- |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|

□□□ □□□ : : □□□□

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

- 람다식
- 람다식

람다식 **Lambda**는 "함수"->"객체"로 변환하는 것

객체 생성 :: new

객체

- 객체지향 프로그래밍
- 객체지향 프로그래밍
- 객체지향 프로그래밍
  - **int** **double** **Integer** **Double**
  - 객체지향 프로그래밍

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

// 객체지향
Integer i1 = 100;

// 객체지향
int i = i1;

// 객체지향
// 10까지의 정수
String str1 = Integer.toString(20);

// 20까지의 정수
String str = "23";
int i1 = Integer.parseInt(str);
```

객체

```
// 객체지향
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是List的子类

```

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

```

## LinkedList

- LinkedList

```

/*
 * 链表 <LinkedList> 是 List 的子类
 */

// LinkedList
public E get(int index){ // 获取指定索引的元素
 return (E)arr[index];
}

// LinkedList
public static <T> T test(T t){
}

```

- 泛型
  - 使用泛型可以防止类型转换错误
- 泛型
  - 使用泛型可以防止类型转换错误
  - 使用泛型可以防止类型转换错误

```

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

```

## Collection(接口)



Collection

- List
- Set

| public boolean add(E e)             |  |
|-------------------------------------|--|
| public void clear()                 |  |
| public boolean remove(E e)          |  |
| public boolean contains(Object obj) |  |
| public boolean isEmpty()            |  |
| public int size()                   |  |
| public Object[] toArray()           |  |

- Iterator

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

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

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

- for

```

/*
for(Iterator iter : list){

}
*/

// 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);
 }
});

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

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

```

## List

API

| Method                        | Description                                                                            |
|-------------------------------|----------------------------------------------------------------------------------------|
| void add(int index,E element) | Adds the specified element at the specified position in the list.                      |
| E remove(int index)           | Removes the element at the specified position in the list.                             |
| E set(int index,E element)    | Replaces the element at the specified position in the list with the specified element. |
| E get(int index)              | Gets the element at the specified position in the list.                                |

```

// Create a List <String> names = new ArrayList()

```

ArrayList

- 可變長度陣列
- 陣列長度從0開始到10000000000
- 陣列長度從1.5開始

LinkedList

- 雙向鏈表

方法

| 方法                        | 說明         |
|---------------------------|------------|
| public void addFirst(E e) | 在列表前面添加元素  |
| public void addLast(E e)  | 在列表後面添加元素  |
| public E getFirst()       | 獲取列表第一個元素  |
| public E getLast()        | 獲取列表最後一個元素 |
| public E removeFirst()    | 刪除列表第一個元素  |
| public E removeLast()     | 刪除列表最後一個元素 |

Set

集合

Collection

HashSet

- 可變長度集合
- 基於哈希表實現
  - 初始容量為16，負載因子為0.75，即16\*0.75=12，當元素數量超過12時，會自動擴容
  - 元素不能為null
  - 元素不能重複
  - 元素不能為null
  - 元素不能重複
- 使用int作為哈希碼，Object的hashCode()方法返回的哈希碼

LinkedHashSet

- 可變長度集合
- 基於鏈表實現，元素按照插入順序排列

TreeSet

- 可變長度集合
  - 基於紅黑樹實現，元素按照自然順序排列
- 1. 實現Comparable接口，實現compareTo方法，比較元素的年齡

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

```

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

```

- 00000000 00000
- 00000000 00000
- 00000000 000

2. TreeSet0000 Comparator 000000000000new00 Comparator 00000000  
compare 000000000

```

Set<Teacher> teachers = new TreeSet<>(new Comparator<Teacher>(){
 @Override
 public int compare(Teacher o1,Teacher o2){
 return o1.getAge() - o2.getAge();// 00
 }
})

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

```

- 0000000000


## Map(0000)

0000

- 0000000000 Set
- 0000000000 Collection

| 0000                                       | 00                          |
|--------------------------------------------|-----------------------------|
| public V put(K key, V value)               | 0000                        |
| public int size()                          | 00000000                    |
| public void clear()                        | 0000                        |
| public boolean isEmpty()                   | 000000000000 true0000 false |
| public V get(Object key)                   | 00000000                    |
| public V remove(Object key)                | 0000000000                  |
| public boolean containsKey(Object key)     | 0000000000                  |
| public boolean containsValue(Object value) | 0000000000                  |
| public Set<K> keySet()                     | 00000000                    |
| public Collection<V> values()              | 00 Map 000000               |



 image-20250822144303974

구분

- 구분

**HashMap**

- 구분
- new** `HashSet` `new` `HashMap`

**LinkedHashMap**

- 구분
- `LinkedHashSet`

**TreeMap**

- 구분
- 구분 `TreeSet`

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

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

구분

- 구분 `Map`

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

- 구분
  - `entrySet()` `map` `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**은 JDK8에서 도입
  - `map.forEach()` 메서드를 사용하면
    - `BiConsumer` 인터페이스를 `map`에 전달
    - `accept` 메서드를 호출
  - 메서드 참조

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

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

## Stream

Stream은 컬렉션의

- 메서드 참조
- `filter()` 메서드를 사용하여
- `stream()` 메서드로 `stream` 생성
- `entrySet()` 메서드로 `set` 생성
  - `Map.Entry<K,V>` 인터페이스를 `set`에 전달

```
// 키 스트림
Set<K> keySet = map.keySet(); // map의 키
Stream<K> keyStream = keySet.stream(); // 키 스트림

// 값 스트림
Collection<V> values = map.values(); // map의 값
Stream<V> valueStream = values.stream();

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

- `Arrays.stream()` 또는 `stream.of()` 메서드로 `Stream` 생성
  - `Stream`은 `Iterable` 인터페이스를 구현

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

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

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

Stream API

Stream API

| API                            | Description                             |
|--------------------------------|-----------------------------------------|
| Stream <T> filter(Predicate)   | Filter elements based on a predicate    |
| Stream <T> sorted()            | Sort elements in ascending order        |
| Stream <T> sorted(Comparator)  | Sort elements using a custom comparator |
| Stream <T> limit(long maxSize) | Limit the number of elements to maxSize |
| Stream <T> skip(long n)        | Skip the first n elements               |
| Stream <T> distinct()          | Remove duplicate elements               |
| Stream <T> map(Function)       | Map each element to a new value         |

```
// scores array map to new array: 10 less + 10
scores.stream().map(s -> "10" + (s + 10));
```

Stream API

Stream API

Stream API

| API                                     | Description                        |
|-----------------------------------------|------------------------------------|
| void forEach(Consumer action)           | Perform an action on each element  |
| long count()                            | Count the number of elements       |
| Optional<I> max(Comparator)             | Find the maximum element           |
| Optional<I> min(Comparator)             | Find the minimum element           |
| R collect(Collector collector)          | Collect elements into a collection |
| Object[] toArray()                      | Convert the stream to an array     |
| public static <T> Collector toList()    | Collect elements into a list       |
| public static <T> Collector toSet()     | Collect elements into a set        |
| public static Collector toMap(Function) | Collect elements into a map        |

`.reduce(())`

`Optional`

- `max` `min` `Optional` `null`
- `Optional` `.get()`
- `Optional`

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

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

## File

`File`

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

// length
f1.length();

// getName
f1.getName();

// isFile
f1.isFile();

// isDirectory
f1.isDirectory();

// createNewFile
f1.createNewFile();

// mkdir (create directory)
f1.mkdir();

// mkdirs
f1.mkdirs();
```

```
// 删除文件(目录)
f1.delete();

// 列出文件及目录,返回数组
f1.list();

// 列出所有File对象
f1.listFiles();

// 绝对路径
f1.getAbsolutePath();
```

## 编码

### ASCII

- ASCII 用 0 到 127 的 8bit 表示
- GBK 用 0 到 255 的 8bit 表示
  - 0 到 127 是 ASCII 码
- Unicode 用 0 到 65535 的 16bit 表示

### UTF-8

- **UTF-8** 用 1 到 4 个字节表示一个字符
  - 0 到 127 用 1 字节表示

## 线程

### Thread

#### Thread

- 通过 `run` 方法运行线程
- 通过 `start` 方法启动线程

```
public class Demo1 extends Thread{
 @Override
 public void run(){
 System.out.println("线程");
 }
}

public class Test{
 public static void main(String[] args){
 Thread t1 = new Demo1();
 // 启动
 t1.start();
 }
}
```

### Runnable

- 调用 run 方法
- 创建 Runnable r 对象 Thread 对象
- 调用 start 方法

```
// 第一种
public class Demo2 implements Runnable{
 @Override
 public void run(){
 System.out.println("Runnable");
 }
}

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

// 第二种
public class Test{
 public static void main(String[] args){
 // 创建 Runnable
 Runnable r = ()->{
 System.out.println("Runnable");
 };
 // 创建 Thread
 Thread t2 = new Thread(r);
 // 启动 Thread
 t2.start();
 }
}
```

## Callable

- 与 Runnable 类似
- Callable 接口 call 方法返回结果
  - 调用 call 方法
- Callable 接口返回 FutureTask 对象
  - 调用 call 方法
  - FutureTask 实现 Runnable 接口
- 调用 FutureTask 的 Thread 方法
- 调用 Thread 的 start 方法
- 调用 FutureTask 的 get 方法

```

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){
 // 创建
 Callable<Integer> c = new Demo3();
 // 包装
 FutureTask<Integer> f = new FutureTask<>(c);
 Thread t2 = new Thread(f);
 // 启动
 t2.start();
 // 等待
 System.out.println(f.get());
 }
}

```

方法

- `public String getName()` 获取名称
- `public void setName(String name)` 设置名称
- `public static Thread currentThread()` 获取当前线程
- `public final void join()` 等待线程结束

同步

线程同步

线程同步

同步

线程同步

```

synchronized(锁){
 同步代码块
}

```

- 线程同步
- 线程同步
  - 使用 `this` 锁
  - 使用 `类名.class` 锁

同步

스레드 동기화 메서드

- 메서드 내부 `this` 객체
- 메서드가 속한 클래스 객체

```
 synchronized 메서드 호출(객체){
 메서드 실행
}
```

## Lock

스레드 동기화 메서드

스레드 동기화 Lock 메서드 `ReentrantLock` 객체

- `void lock()` 메서드
- `void unlock()` 메서드

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

스레드

스레드 관리자 `ExecutorService`

스레드 관리자

1. `ExecutorService` 객체 `ThreadPoolExecutor` 객체 생성
2. `Executors` 클래스를 사용하여 객체 생성

`ThreadPoolExecutor` 객체

- `Runnable` 인터페이스

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

// 예제

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

- `corePoolSize` 기본 스레드 수
- `maximumPoolSize` 최대 스레드 수
- `keepAliveTime` 스레드 유효 시간
- `unit` 시간 단위
- `workQueue` 작업 큐
- `threadFactory` 스레드 팩토리
- `handler` 실행 거부 핸들러

`ExecutorService` 객체

- `void execute(Runnable command)` 메서드 `Runnable` 객체



- `Future<T> submit(Callable<T> task)` 返回 Callable 返回的 Future 对象
- `void shutdown()` 关闭线程池
- `List<Runnable> shutdownNow()` 立即关闭线程池\*\*并返回正在运行的任务\*\*

线程池的构造方法

- `ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue)`

线程池的构造方法

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

线程池的构造方法

- `AbortPolicy` 拒绝策略
- `DiscardPolicy` 丢弃策略
- `DiscardOldestPolicy` 丢弃最老的未处理的任务
- `CallerRunsPolicy()` 由调用者来运行此任务

线程池的构造方法

线程池的构造方法

线程池的构造方法

## IO

线程池的构造方法

- `FileInputStream`
- `FileOutputStream`

线程池的构造方法

- `File`
- `FileChannel`

线程池的构造方法

线程池的构造方法

线程池的构造方法

- `FileChannel`
- `FileChannel`
- `@Test`

线程池的构造方法

线程池的构造方法

- `FileChannel`

线程池的构造方法

1. 线程池的构造方法
  - `Class c1 = .class`
  - `Class.forName(String package)` **package**

- `Class clazz = Class.forName("测试")`
  - `Class c1 = clazz.getClass()`
2. 测试构造**Constructor**
  3. 测试**Field**
  4. 测试**Method**

```
// 测试
Class c1 = Student.class; //测试
System.out.println(c1.getName()); // 测试
System.out.println(c1.getSimpleName()); // 测试

// 测试构造
Constructor [] cons = c1.getDeclaredConstructors(); //测试
for(Constructor con: cons){
 System.out.println(con.getName() + "("+con.getParmeterCount()+")"); // 测试
}

// 测试
Constructor con2 = c1.getDeclaredConstructor(String.class,int.class); // 测试

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

// 测试
Method method = c1.getDeclaredMethod("eat"); // 测试eat
Method method1= c1.getDeclaredMethod("eat",String.class); // 测试eatString
System.out.println(method.getName() + "("+method.getParmeterCount()+")");
```

测试构造

```
Class c1 = Student.class; //测试
Constructor con = c1.getDeclaredConstructor(); // 测试

// 测试
con.Accessible(true); // 测试
Student c1 = (Student) con.newInstance(); // 测试
System.out.println(C1);
```

测试Field

```
Student c1 = new Student("测试",12); // new
Field field = c1.getDeclaredField("hobby"); // 测试
field.setAccessible(true);
field.set(c1,"测试"); // c1测试
```

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

測試

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

method.setAccessible(true);
method.invoke(c1); // 執行c1.eat() 方法
```

說明

- 取得c1.hobby
- 取得c1
- 取得c1.getDeclaredMethod("eat",12) (class)方法

```
ArrayList <String> list = new ArrayList<>();
list.add("1");
list.add("2");
list.add(88); // 1 2 88

// 取得c1
Class c1 = list.getClass();
Method m1 = c1.getDeclaredMethod("add",Object o);
m1.invoke(list,11); // 1 2 88 11
m1.invoke(list,true); // 1 2 88 11 true
```

說明

- @Override 或 @Test 方法
- 取得c1
- 取得c1

測試

- 取得value
  - 取得value

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

```

}

@Mybook("delete") // 删除
@Mybook(value = "delete") // 删除

```

总结

常用注解

- @Target 标注目标
  - TYPE 类型
  - FIELD 域
  - METHOD 方法
  - PARAMETER 参数
  - CONSTRUCTOR 构造器
  - LOCAL\_VARIABLE 局部变量
- @Retention 标注保留策略
  - **RUNTIME** 运行时

```

@Retention(RetentionPolicy.RUNTIME) // 运行时
@Target({ElementType.METHOD})
public @interface Mybook{ // 接口
 String value() ;
}

```

应用

- 使用注解
- 使用反射

```

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

@Mybook(value ="测试",address = {"测试","测试"})
public class Demo{

}

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

```

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

□□□□

□□□□□□

- 代理類實現了被代理類的所有方法，並將這些方法調用委派給被代理類實現
- 代理類實現了被代理類的所有方法，並將這些方法調用委派給被代理類實現
- 代理類實現了被代理類的所有方法，並將這些方法調用委派給被代理類實現
  - 代理類實現了被代理類的所有方法，並將這些方法調用委派給被代理類實現
  - 代理類實現了被代理類的所有方法，並將這些方法調用委派給被代理類實現
  - 代理類實現了被代理類的所有方法，並將這些方法調用委派給被代理類實現
    - 代理類實現了被代理類的所有方法，並將這些方法調用委派給被代理類實現
    - 代理類實現了被代理類的所有方法，並將這些方法調用委派給被代理類實現

□□□□□

□□□□

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

111

```
@ArgsConstructor
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 + " " + "Style");
 return " ";
 }
}
```

111

```
// 代理
import java.lang.reflect.Proxy

// 代理實現
import java.lang.reflect.InvocationHandler
```

```

public class ProxyUtil{
 public static Mess createproxy(Real r){
 // 创建代理对象
 // 创建目标对象
 // 创建代理类
 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{

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

 // 返回结果
 System.out.println("返回结果");
 return result;
 }
 }
);
 return m;
 }
}

```

测试

```

public class Test(){
 public static void main(String []args){
 Real r = new Real("测试");
 // 创建代理对象
 Mess proxy = ProxyUtil.createProxy(r);
 // 调用方法
 proxy.sing("测试");
 System.out.println(proxy.dance());
 }
}

```