**JDK8(2014-03, LTS)**

1）接口默认方法（常用）

即接口中可以声明一个非抽象的方法作为默认的实现，方法的返回类型前要加上“default”关键字

JDK 8以前的接口：

interface 接口名 {

    静态变量;

    抽象方法;

}

JDK 8的接口：

interface 接口名 {

    静态变量;

    抽象方法;

    默认方法;

    静态方法;

}

public interface JDK8Interface {

    /\*\*

     \* 默认的时候 就是为 public abstract

     \*/

    void add();

    /\*\*

     \* jdk8 开始 提供了默认的方法，接口也可以定义默认的方法体

     \*/

     default void get(){

        System.out.println("6666");

    }

    /\*\*

     \* JDK8开始 可以用static 在 接口中修饰方法

     \*/

     static void staticDel(){

         System.out.println("Del");

     }

}

2）lambda表达式（常用）

(参数列表) -> {

    代码体;

}

lambda表达式相当于是对接口中抽象方法的重写。

匿名内部类

前提：

1. 方法的参数或变量的类型是接口。
2. 接口中有且仅有一个抽象方法。

3）函数式接口（常用）

指仅仅包含一个抽象方法的接口，需要标记@FunctionalInterface

常用的内置函数式接口：

包名：java.util.function

* Supplier
* Consumer
* Function
* Predicate

4）使用 :: 来传递方法引用，是lambda的简写（常用）

* 对象名引用成员方法
* 类名引用静态方法
* 类名引用实例方法

|  |  |  |
| --- | --- | --- |
| **类型** | **语法** | **Lambda表达式** |
| 静态方法引用 | 类名::静态方法名 | (args) -> 类名.静态方法名(args) |
| 实例方法引用 | 实例::实例方法名 | (args) -> 实例.实例方法名(args) |
| 对象方法引用 | 类名::对象方法名 | (inst,args) -> 类名.对象方法名(args) |
| 构建方法引用 | 类名::new | (args) -> new 类名(args) |

5）Stream API(常用)

Stream流式的思想类似于工厂车间的“生产流水线”。

串行和并行流

并行流的顺序问题

并行流的线程安全问题

    加锁

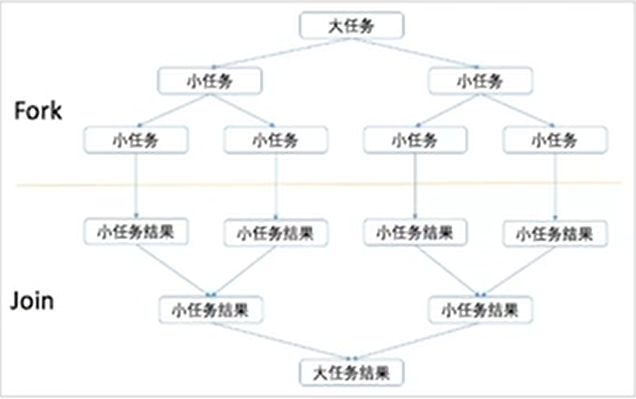
    使用线程安全的集合

    使用toArray()

并行流不适合的场景

parallelStream背后的技术：

Fork/Join框架（From JDK 1.7）



* 分治
* 工作窃取算法

6) Optional 处理

工具类，主要是为了解决避免Null检查，防止NullPointerException.

使用orElse, ifPresent, ifPresentOrElse, map等方法避免对null的判断，写出更加优雅的代码。

Optional<String> userName = Optional.of("dk");

userName.ifPresentOrElse(s -> {

    System.out.println("有值");

}, () -> {

    System.out.println("为空");

});

Optional<User> user = Optional.of(u);

String wu = u.map(UserDto::getName).map(String::toUpperCase).orElse("没有值");

7）新的日期工具类(常用)

旧版的日期时间API存在的问题。

1. 设计不合理
2. 时间格式化和解析是线程不安全的
3. 时区处理麻烦

**JDK9(2017-09)**

1）模块化

* JAVA运行环境代码臃肿，效率低。
* 无法隐藏内部API和类型

即只加载应用运行需要的模块

2) 交互式编程

JShell，即时反馈

3）改进的Java doc

生成的Java doc符合H5标准，支持API搜索

4）集合工厂方法，优化集合初始化（常用）

创建只读集合：

Set<Integer> ints = Set.of(1,2,3);

List<String> strings = List.of("first","second");

Map map = Map.of(1, "AAA", 2 "BBB");

UnsupportedOperationException

5）改进 Stream API

IntStream.iterate(1, i -> i < 100, i -> i + 1).forEach(System.out::println);

6）支持私有接口方法

解决静态或者默认方法代码重复的问题。

7）HTTP/2（常用, 是预览版）

新的方式处理HTTP调用，用于替换 HttpURLConnection，并提供对WebSocket和HTTP/2的支持

8）多版本兼容

9) 释放资源代码优化

10) String，StringBuffer, StringBuilder.

11) 全新的HTTPClient

**JDK10(2018-03)**

1) 局部变量类型推断

var str = "ABC"; //根据推断为 字符串类型

var l = 10L;//根据10L 推断long 类型

var flag = true;//根据 true推断 boolean 类型

var flag1 = 1;//这里会推断boolean类型。0表示false 非0表示true

var list = new ArrayList<String>();  // 推断 ArrayList<String>

var stream = list.stream();          // 推断 Stream<String>

2) 并行full GC的G1

**JDK11(2018-09，LTS)**

1)  增加一些实用API

对于String：

String str = "woshidage";

boolean isblank = str.isBlank();  //判断字符串是空白

boolean isempty = str.isEmpty();  //判断字符串是否为空

String  result1 = str.strip();    //首尾空白

String  result2 = str.stripTrailing();  //去除尾部空白

String  result3 = str.stripLeading();  //去除首部空白

String  copyStr = str.repeat(2);  //复制几遍字符串

long  lineCount = str.lines().count();  //行数统计

对于集合API：

JDK 11 前：

Integer[] nums = list.toArray(new integer[0]);

Integer[] nums2 = list.toArray(Integer[]::new);

对IO的增强：

Files.writeString(Path.of("deal.text"), "Hello");

Files.readString(Path.of("deal.text"), StandardCharsets.UTF\_8);

is.transferTo(os);

2）HttpClient增强（常用，已转换为正式版）

同时提供了同步调用和异步调用实现。JDK对http对调用支持已经足够强，以前apache提供的httpComponents基本可以去除了。

动机：

* URLConnection API设计时考虑了多种协议，而这些协议是现在已经不存在的。
* API早于HTTP/1.1并且过于抽象
* 难于使用
* 仅能在阻塞模式下工作
* 难于维护

4）ZGC

JDK11最瞩目的特性，但目前是实验性质的。目标是GC暂停时间不会超过10ms，既能处理几百兆的小堆，也能处理几个T的大堆。

目标：

* 支持TB量级的堆
* 最大GC停顿时间不超过10ms，停顿时间不随heap大小或存活对象大小增大而增大。
* 奠定未来GC特性的基础
* 最糟糕情况下吞吐量会降低15%

Supported Platforms:

Linux/x64: JDK11

Linux/AArch64: JDK13

macOS: JDK14

Windows: JDK14

5）完全支持Linux容器（常用）

JDK10开始，JVM可以识别当前是否在容器中运行，能接受容器设置的内存限制和CPU限制。

**JDK12(2019-03)**

1) Switch Expression

* 可不写break.
* 可以使用switch作为表达式返回结果
* 多值匹配

传统使用switch的方式：

switch (day) {

    case MONDAY:

    case FRIDAY:

    case SUNDAY:

    System.out.println(6);

    break;

case TUESDAY:

    System.out.println(7);

    break;

case THURSDAY:

case SATURDAY:

    System.out.println(8);

    break;

case WEDNESDAY:

    System.out.println(9);

    break;

}

如今可简化为：

switch (day) {

case MONDAY, FRIDAY, SUNDAY -> System.out.println(6);

case TUESDAY -> System.out.println(7);

case THURSDAY, SATURDAY -> System.out.println(8);

case WEDNESDAY -> System.out.println(9);

}

2) JMH微基准测试套件

**JDK13(2019-09)**

1) 字符串拼接

String html = """

              <html>

                  <body>

                      <p>Hello, world</p>

                  </body>

              </html>

              """;

**JDK14(2020-03)**

1)  instanceof 扩展 （preview）

传统instanceof使用方法：

if (o instanceof String) {

    String s = (String)o;

    System.out.println(s.length);

}

现在可以增强为：

if (o instanceof String s) {

    System.out.println(s.length);

}

2)  异常信息打印，针对链式调用时发生

**JDK15(2020-09)**

引入了Record类型，解决定义大量get/set方法的问题，可一定程度替代掉lomobok

示例如下：

public record Address(

        String addressName,

        String city

) {}

public record CustUser(

        String firstName,

        String lastName,

        Address address,

        int age

) {}

Record和普通的类的区别就在于Record多了一个括号括起来的定义的字段。

Record类默认是final的，里面的字段默认是private final的。、

ZGC可以作为生产使用

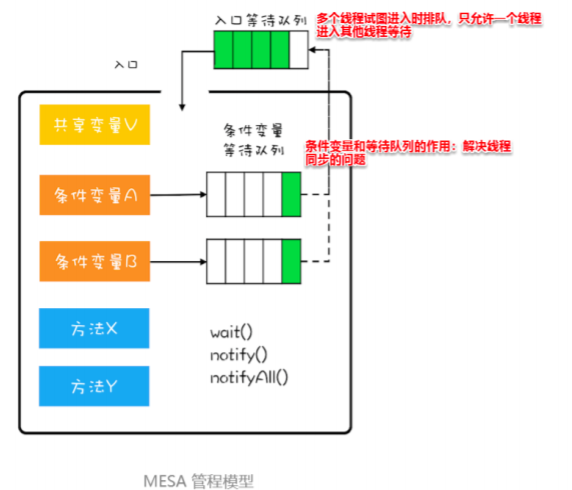
**JDK16(2021-03)**

优化了ZGC的内存使用率，不需要预留内存给重分配。

**JDK17(2021-09，LTS)**

**高并发**

1. 线程基础

* 进程&线程&协程
* java线程的生命周期
* start，sleep，join，yield等方法详解
* java线程中断机制
* Java线程间通信：volatile，等待唤醒机制（cas+park/unpark, sychronized+wait/notify/notifyAll, reetrantLock+Condition(await/signal/signalAll)），管道输入输出流，Thread.join
* 管程：MESI模型，synchronized，Lock+Condition
  + 
* Volatile
* CAS

2. 线程池

为什么要使用线程池？什么场景下用？

线程池选择及线程池的构造参数

线程池原理解析

3. 线程安全问题

问题分类

线程安全问题解决方法：

    无锁实现

    有锁实现

4. 并发工具类

* atomic包下原子类
* 线程池
* 线程协作工具类
* 并发容器
* Future
* Fork/Join

**Java并发知识体系**



并发理论基础

并发&并行



线程基础

线程&进程

Java线程的生命周期（6种状态）

start,sleep,join,yield等方法详解

Java线程的中断机制



Java线程间通信

volatile



等待唤醒机制

cas+park/unpark

sychronized+wait/notify/notifyAll

reentrantLock+Condition(await/singal/singalAll)

管道输入输出流

Thread.join



管程

synchronized

Lock+Condition



多线程模型



共享模型（JMM）



并发编程bug的源头

可见性问题

有序性问题

原子性问题

volatile原理

cas原理

happens-before原则



主存和工作内存交互8大原子操作

lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占状态。

unlock（解锁）：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。

read（读取）：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用

load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。

use（使用）：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。

assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。

store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。

write（写入）：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。



CPU高速缓存架构



多种缓存一致性协议

MSI protocol, the basic protocol from which the MESI protocol is derived.

Write-once (cache coherency), an early form of the MESI protocol.

MESI protocol

MOSI protocol

MOESI protocol

MESIF protocol

MERSI protocol

Dragon protocol

Firefly protocol

伪共享问题

非共享模型



线程安全问题



分类

运行结果错误

活跃性问题（死锁，饥饿，活锁）

对象发布和初始化



线程安全问题解决方案



无锁

局部变量



不可变对象

final关键字

ThreadLocal



cas

atomic包下原子类



有锁实现



锁分类

synchronized



Lock



基于AQS同步器(AbstractQueuedSynchronizer)

ReentrantLock

ReentrantReadWriteLock

StampedLock



并发工具类



atomic包下原子类

AtomicInteger、AtomicLong、AtomicBoolean

AtomicReference、AtomicStampedRerence、AtomicMarkableReference

AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray

AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater

DoubleAccumulator、DoubleAdder、LongAccumulator、LongAdder、Striped64



线程池

ThreadPoolExecutor

ScheduledThreadPoolExecutor



线程协作

CountDownLatch

Semaphore

CyclicBarrier

Exchanger

Phaser



并发容器



Map

ConcurrentHashMap

ConcurrentSkipListMap



List

CopyOnWriteArrayList



Set

CopyOnWriteArraySet

ConcurrentSkipListSet



Queue



BlockingQueue

ArrayBlockingQueue

LinkedBlockingQueue

SynchronousQueue

PriorityBlockingQueue

DelayQueue

LinkedTransferQueue

ConcurrentLinkedQueue



BlockingDeque

LinkedBlockingDeque

ConcurrentLinkedDeque



Future

FutureTask

CompletionService

CompletableFuture



Fork/Join

ForkJoinPool

ForkJoinTask

高性能队列Disruptor

。。。



并发设计模式



Immutability模式

想破坏也破坏不了

Copy-on-Write模式



线程本地存储（Thread-Specific Storage）模式

没有共享就没有伤害



Guarded Suspension模式

等我准备好哦



Balking 模式

不需要就算了



Thread-Per-Message 模式

最简单实用的分工方法



Worker Thread模式

如何避免重复创建线程



两阶段终止（Two-phase Termination）模式

如何优雅的终止线程



生产者-消费者模式

用流水线的思想提高效率

。。。

**反应式编程**

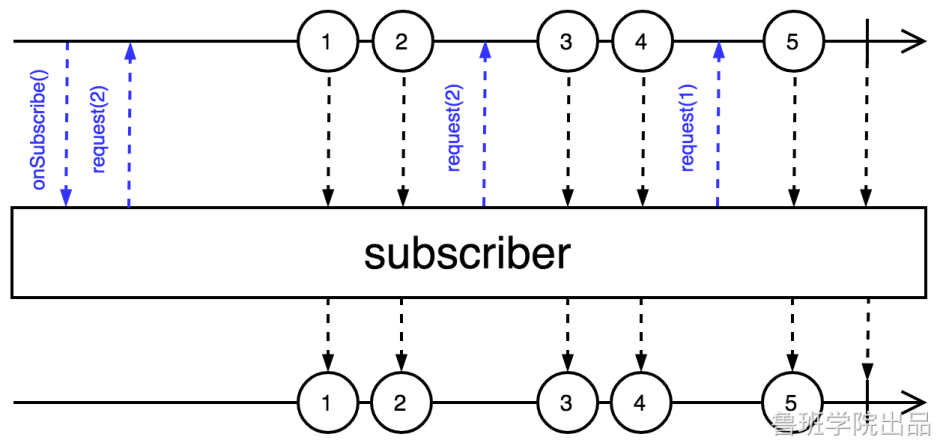
JDK8 Lambda Function Stream

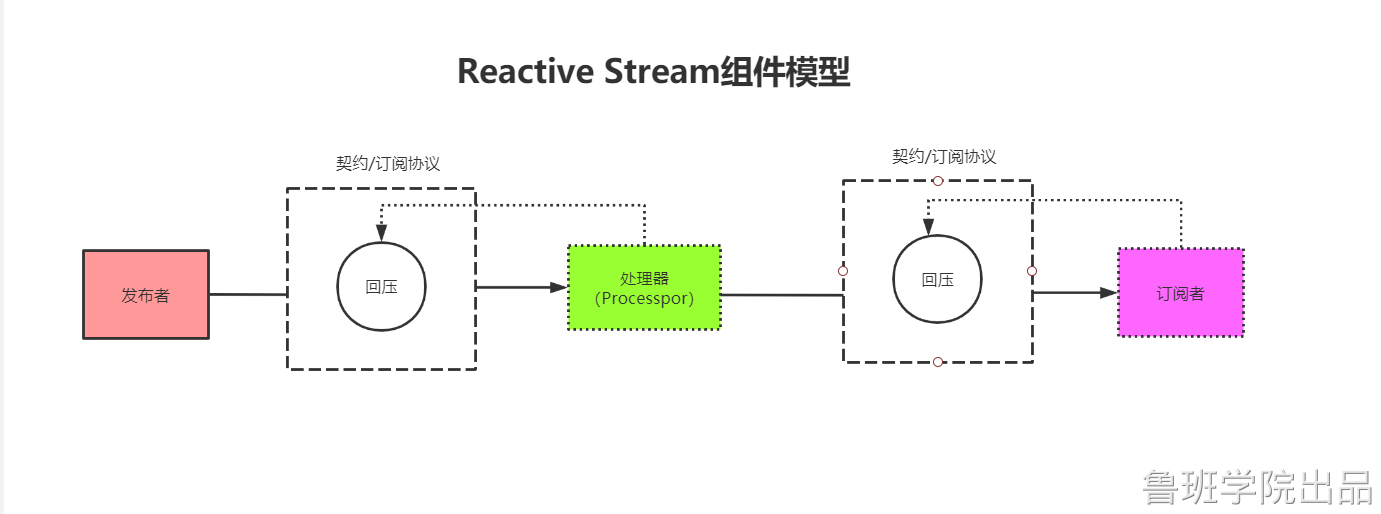
Reactive Stream(JDK9)

     基于发布订阅者模式处理规范

<https://github.com/reactive-streams/reactive-streams-jvm>

**背压**指的发布者和订阅者之间的互动





    主要接口Flow

* Publisher<T>接口发布者

            subscribe(Subscriber<? super T>):void:保证发布者和订阅者之间通过此方法建立订阅关系

* Subscriber<T>订阅者

            onSubscribe(Subscription):void:第一次签署订阅关系，输入就是Subscription对象

            onNext(T):void:接收到一条数据

            onError(Throwable):void:数据出错

            onComplete():void:数据处理完了

* Subscription发布者与订阅者之间的关系

            request(long):void:告诉发布者需要多少资源

            cancel():void

* Processor<T,R>接口继承了Publisher和Subscriber，表示既可以做消费者，又可以做发布者，承担中间角色

            defaultBufferSize():int

Project Reactor

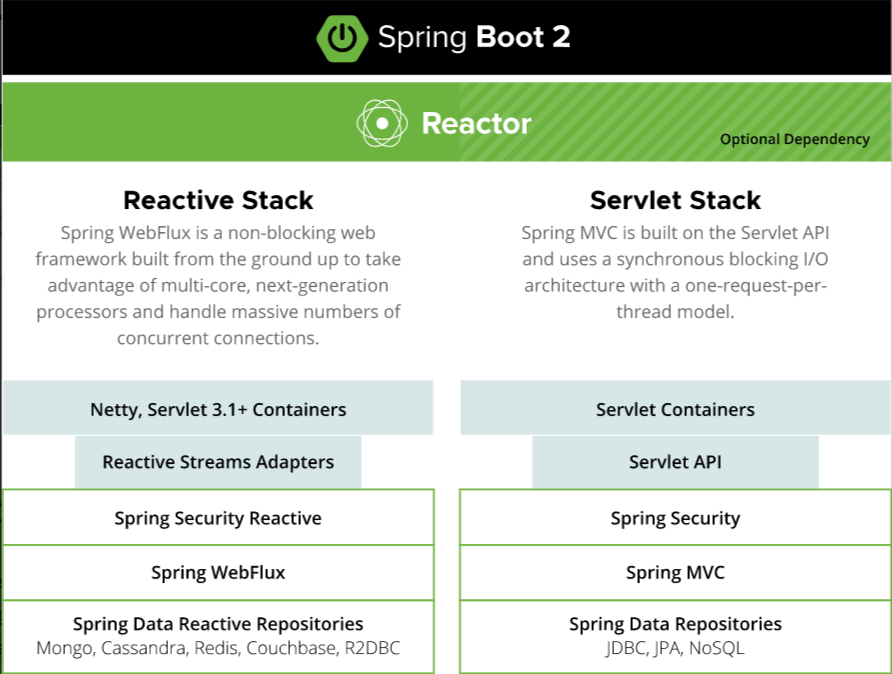
<https://projectreactor.io/docs/core/release/reference/>

    Reactor is a fourth-generation reactive library, based on the Reactive Streams specification, for building non-blocking applications on the JVM

**Flux,** **Mono，Future**,  **CompletableFuture,**

WebFlux+SpringBoot+r2dbc

    Servlet线程模型的比较：同步阻塞，异步阻塞，异步非阻塞



    注解方式。

    非注解方式：RouterFunction