

# Java后端学习笔记

whq

## Java后端学习笔记

Java基础

Java特点

JVM JRE JDK

Java和C++

基本类型

基本类型和包装类型

自动包箱拆箱

浮点数精度丢失：BigDecimal

方法的重写

访问修饰符

equals()

== 和 equals()

equals() 和 hashCode()

final

String

String不可变

StringBuilder StringBuffer

字符串常量池

String.intern()

+

编译器优化

异常

finally

try-with-resources

泛型

反射

代理

静态代理

动态代理

JDK动态代理

CGLIB动态代理

注解

序列化

集合

Comparable和Comparator

Comparable

Comparator

Arrays.sort()

List/Deque

ArrayList 与 LinkedList

ArrayDeque 与 LinkedList

BlockingQueue

当HashMap以Object为key

HashMap遍历

ConcurrentHashMap

对比Hashtable

实现线程安全：JDK1.7与1.8

ConcurrentHashMap不允许null  
 集合list - 数组array 转换  
 ArrayList扩容  
 HashMap源码  
 JDK1.7 vs 1.8  
 LinkedHashMap

**并发**

- 进程与线程
- 如何创建线程
- 线程生命周期和状态
- Thread#sleep()和Object#wait()
- 死锁
- 单核CPU运行多线程
- JMM内存模型
  - 并发编程三特性 (问题)
  - JMM的用处
  - JMM和JVM内存结构
  - happens-before原则
- volatile关键字
- synchronized关键字
  - synchronized原理与锁升级
  - synchronized和volatile
- 乐观锁、悲观锁
  - CAS
  - CAS的局限性
- 公平锁、非公平锁
- 可重入锁、不可重入锁
- 共享锁、独占锁
- 读锁、写锁
- ReentrantLock
  - ReentrantLock和synchronized
- ThreadLocal
- 线程池
  - 创建线程池的方法
  - 线程池的拒绝策略
  - 阻塞队列
  - 线程池中线程异常后，销毁新建 or 复用
  - 针对CPU密集型/IO密集型的线程池优化
- Future
  - CompletableFuture
- AQS
  - AQS原理

## Adapter Pattern

NIO

NIO实现原理

JVM

运行时内存区域

线程私有

程序计数器

JVM栈 (栈)

本地方方法栈

堆

字符串常量池

方法区

常量池

直接内存

垃圾回收GC

死亡对象判断方法

**引用类型**

GC算法

GC算法-分代回收

HotSpot GC类型

分配与回收流程

垃圾收集器Collector

类加载器ClassLoader

ClassLoader作用

ClassLoader种类

`ClassLoader#getParent()`

双亲委派模型

## RDBMS - MySQL

数据库范式

MySQL字段

SQL语句在MYSQL的执行过程

MySQL运行逻辑架构

SQL语句执行

SQL语法执行顺序

MySQL索引

索引划分

**常见索引数据结构**

聚集索引和非聚集索引

覆盖索引

联合索引

索引下推

正确建立索引

索引失效场景

MySQL执行计划EXPLAIN

MySQL日志

redo log

binlog

**两阶段提交**

undo log

MySQL事务

**ACID特性**

事务并发问题

并发控制方式

事务隔离级别

MVCC多版本并发控制

- 一致性非锁定读和锁定读
- InnoDB实现MVCC
- RC和RR隔离等级下MVCC不同
- MVCC的局限性
- 临键锁Next-key Lock
- MySQL锁
  - 行级锁和表级锁
  - Next-key Lock加锁范围
  - 意向锁Intension Lock
- MySQL性能优化
- 分库分表
- 不推荐用外键和级联
- drop truncate delete
- NoSQL - Redis**
  - Redis
  - Redis为什么速度快
  - Redis和Memcached
  - Redis数据类型
    - String
    - List
    - Hash
    - Set
    - Sorted Set
      - SortedSet底层: skipList跳表
    - Bitmap位图
    - HyperLogLog基数统计
    - Geospatial地理空间
  - Redis持久化
    - RDB
    - AOF
    - RDB + AOF: 混合持久化
  - Redis线程模型与IO
  - 内存管理: 过期时间
  - Redis性能优化
    - 慢查询
    - key集中过期
    - bigkey
    - hotkey
    - 持久化阻塞
    - Swap内存交换
    - 网络阻塞
  - Redis生产问题
    - 缓存穿透
    - 缓存击穿 (热点key问题)
    - 缓存雪崩
  - 缓存策略与数据库一致性
    - 缓存策略
    - Cache Aside Pattern 旁路缓存的一致性分析
  - Redis应用场景
  - Redis分布式锁
    - 简易分布式锁 setnx
    - setnx分布式锁的问题
  - Redisson
    - 可重入的Redis分布式锁
    - 解决不可重入问题

## Maven

## Spring

Spring基础

Spring模块

Spring / SpringMVC / SpringBoot

Spring IoC

IoC/DI简介

Spring IoC容器

Bean

声明一个类为Bean的注解

注入Bean的注解

Bean注入的场景 (在代码中的注入方式)

Bean作用域

Bean线程安全

Bean生命周期

Spring AOP

AOP简介

SpringAOP 原理

Advice通知类型

实战

SpringMVC

过滤器和拦截器

Spring设计模式

Spring循环依赖

三级缓存解决循环依赖

@Lazy解决循环依赖

Spring事务

Spring两种事务管理方式

Spring事务管理接口

事务属性

传播行为

isReadOnly: 为什么数据库查询 (读) 还需要开启事务?

@Transactional原理

AOP自调用问题

@Transactional使用注意事项

JPA

JDBC与DataSource

JPA、Spring Data JPA、Hibernate关系

JPA常用注解

SpringBoot自动装配

原理

如何实现一个Starter

总结

## MyBatis

MyBatis整体架构

MyBatis+使用文档

MyBatis原理与使用

为什么说MyBatis是半自动ORM框架

MyBatis XML映射工作原理

DAO接口工作原理

DAO接口能重载吗

不同XML文件内的 id 可以重复吗

sqlSession

## 工作例子

`{} 和 #{} 的区别`

MyBatis分页

SQL手动分页

MyBatis RowBounds

分页插件PageHelper

MyBatis插件原理

MyBatis关联查询

MyBatis执行器

如何指定执行器?

MyBatis缓存

## 消息队列

基础知识

消息队列特点

JMS

JMS五种消息数据类型

JMS两种消息模型

AMQP

AMQP核心概念

JSM vs AMQP

RPC vs MQ

MQ技术选型

RabbitMQ

特性

核心概念与架构

交换机类型与工作模式

Exchange Type 交换机类型

RabbitMQ的工作模式

死信 DL

死信与延迟队列

RabbitMQ通信方式: Channel

RabbitMQ实现高可用: 集群

普通集群模式

镜像集群模式

如何保证消息传输的可靠性

## 设计模式

OS

用户态和内核态

用户态切换到内核态

进程与线程

线程间通信

进程间通信

进程调度算法

上下文 context

虚拟内存

分段映射

分页映射

硬链接和软链接

磁盘调度算法

## 计网

网络分层模型

URL访问网页的过程

TCP

三次握手

如何保证传输可靠?

TCP vs UDP  
HTTPS加密机制  
**手撕**  
  算法  
    快排  
    堆排序  
  Java  
    多线程交替打印ABC  
      semaphore  
      synchronized  
      ReentrantLock  
    线程池的使用  
  MySQL  
    连接  
    函数  
  配置文件  
    DockerFile  
    CI/CD  
**其他**  
  鉴权与授权  
    JWT  
    Spring Security  
    越权访问问题  
  代码沙箱的实现  
  finalize  
  IO多路复用机制  
  Dubbo - RPC  
  分布式CAP问题  
  排查CPU占用(Java)  
  大模型与RAG, CoT、ToT

## Java基础

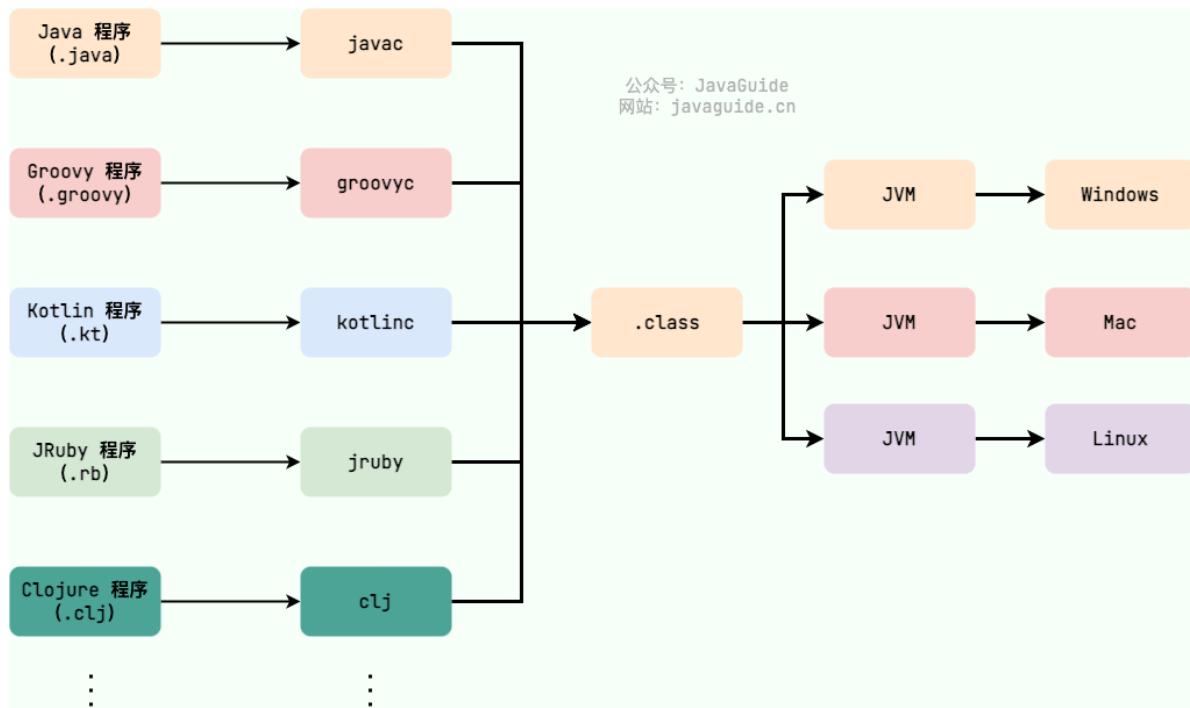
---

### Java特点

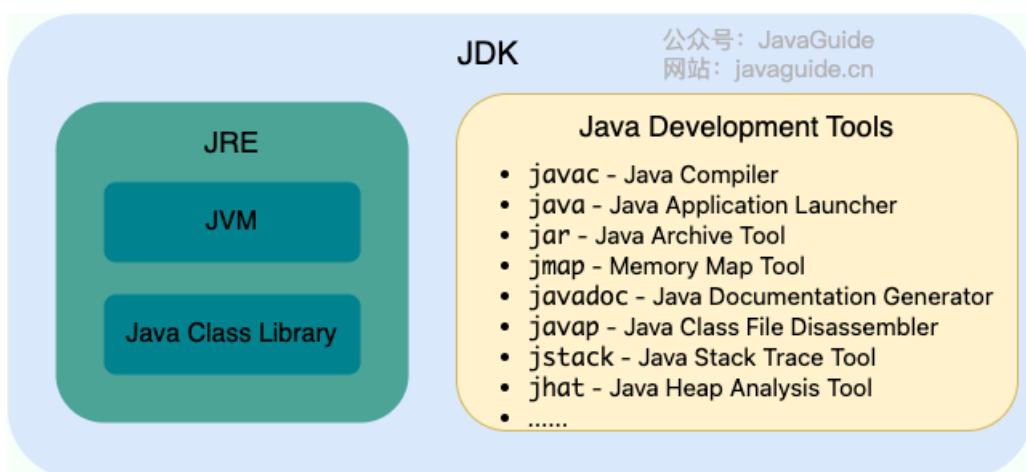
1. 面向对象 (封装 继承 多态)
  2. 跨平台 (Write Once, Run Anywhere)
  3. 提供内置的多线程机制
  4. 安全：访问权限修饰符
  5. 编译+解释，效率+移植
- 
- Java SE 标准版本Standard Edition
  - Java EE 企业版本 (java se + Servlet、JSP、EJB、JDBC、JPA、JTA、JavaMail、JMS...)

### JVM JRE JDK

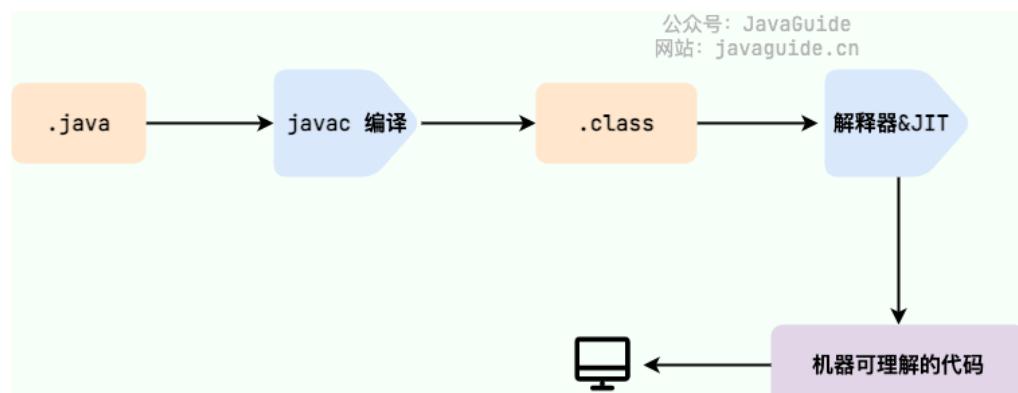
- JVM: java虚拟机，运行.class字节码，跨平台的关键。jvm是一种规范，有多种实现，常见的hotspotVM



- JRE: java运行时环境。JRE = JVM+Java Class Library (网络通信、I/O、数据结构)
- JDK: Java开发工具包



java9开始不再单独区分JRE和DK，取而代之模块化JDK，使用Jlink根据当前依赖创建所需的小JRE



JIT (just in time) 编译模式，Jvm自带，对高频热点代码进行运行时编译，编译成机器码加快其运行速度

AOT (ahead of time) 编译模式java9出现，包小快启动，但能力受限，适合云原生

## Java和C++

都是面向对象，封装继承多态

- Java无指针，保护内存
- Java类只能单继承，接口可以多继承；C++类支持多继承
- Java内存自动回收
- Java只能方法重载（除了字符串 + +=）；C++还支持操作符重载

## 基本类型

基本类型	位数	字节	默认值
byte	8	1	0
short	16	2	0
int	32	4	0
long	64	8	0L
char	16	2	'\u0000'
float	32	4	0f
double	64	8	0d
boolean	1		false

## 基本类型和包装类型

- 包装类型可以用于泛型
- 基本类型局部变量在栈中，成员变量在堆中。对象实例都在堆中
- 包装类型重写了 equals 方法使得它能够比较内容，而 == 则会比较包装类型（Object 类型）的内存地址
- 包装类型在一个较小的范围(128)内不会重新 new，而是使用提前创建好的缓存

## 自动包箱拆箱

- Integer i = 10 等价于 Integer i = Integer.valueOf(10)
- int n = i 等价于 int n = i.intValue();

## 浮点数精度丢失：BigDecimal

浮点数存储机制导致了精度丢失。在银行场景下，需要完全精确的浮点数运算，使用 BigDecimal。用 String 创建 BigDecimal。

BigDecimal 进行等值比较时应该使用 compareTo()。这是因为 equals() 方法不仅仅会比较值的大小 (value)，还会比较精度 (scale)，而 compareTo() 方法比较的时候会忽略精度。

# 方法的重写

方法的重写要遵循“**两同两小一大**”，核心思想是**保证向上转型的可用**（子类转成父类时能正常作为父类使用）

- “**两同**”：即方法名相同、形参列表相同；
- “**两小**”：指的是子类方法返回值类型应比父类方法返回值类型更小或相等，子类方法声明抛出的异常类应比父类方法声明抛出的异常类更小或相等；
- “**一大**”：指的是子类方法的访问权限应比父类方法的访问权限更大或相等。

## 访问修饰符

Access Modifiers

修饰符	同类	同包	子类(即使不同包)	其他包
<b>public</b>	✓ 可以访问	✓ 可以访问	✓ 可以访问	✓ 可以访问
<b>protected</b>	✓ 可以访问	✓ 可以访问	✓ 可以访问	✗ 不能访问
<b>默认(无修饰符)</b>	✓ 可以访问	✓ 可以访问	✗ 不能访问	✗ 不能访问
<b>private</b>	✓ 可以访问	✗ 不能访问	✗ 不能访问	✗ 不能访问

## equals()

`==` 和 `equals()`

因为 Java 只有**值传递**，所以，对于 `==` 来说，不管是比较基本数据类型，还是引用数据类型的变量，其**本质比较的都是值**，只是引用类型变量存的值是对象的地址。

- 对于基本数据类型来说，`==` 比较的是值。
- 对于引用数据类型来说，`==` 比较的是对象的内存地址。

类才有equals方法。若类没有重写 `equals()` 方法则等价于通过 `==` 比较这两个对象，使用的默认是 `Object.equals()` 方法。

`String` 重写了 equals 方法，使其先尝试 `==`，再去比较值。这是因为 `String` 有一个常量池机制。

`equals()` 和 `hashCode()`

`hashCode()` 用于在 `HashMap` 和 `HashSet` 中快速查找和添加：在查找当前元素是否存在于 Map 中时，会先用 `hashCode()`，若命中则继续 `equals()` 比较。（`equals` 更加精确）

因此 `equals()` 相等时 `hashCode()` 也必须相等，重写 `equals()` 也必须重写 `hashCode()`。

## final

- final修饰的类无法被继承
- final修饰的变量无法被修改
  - 基本类型不能改
  - 引用类型无法改变指向 (因为引用类型存的值，就是内存地址)
- final修饰的方法不能被重写

## String

### String不可变

```
public final class String implements java.io.Serializable, Comparable<String>,  
CharSequence {  
    private final char value[];  
    //...  
}
```

保存String字符串的数组是final且private，且String没有暴露setter；String类是final，防止被子类继承  
破环值。

### StringBuilder StringBuffer

StringBuilder线程不安全；StringBuffer使用了同步锁synchronized，线程安全。

字符串+和+=是java中唯二的运算符重载，字符串直接量相加会在编译时优化为StringBuilder

### 字符串常量池

JVM为了提升性能和减少内存消耗针对字符串（String类）专门在堆中开辟的一块区域，主要目的是为了避免字符串的重复创建。

String a = new String("abc") 创建了几个字符串对象？

1个或2个

1. 若字符串常量池里没有"abc"，则先创建一个字符串对象保存在常量池里，然后再new String指向"abc"
2. 若已有，则直接new String指向"abc"

### String.intern()

返回改字符串在常量池里的引用

```
String s1 = "Java"; // s1指向常量池的"Java"  
String s3 = new String("Java"); // s3指向堆内存中的String对象  
String s4 = s3.intern(); // s4指向常量池的"Java"  
// s1 == s4  
// s3 != s4
```

+

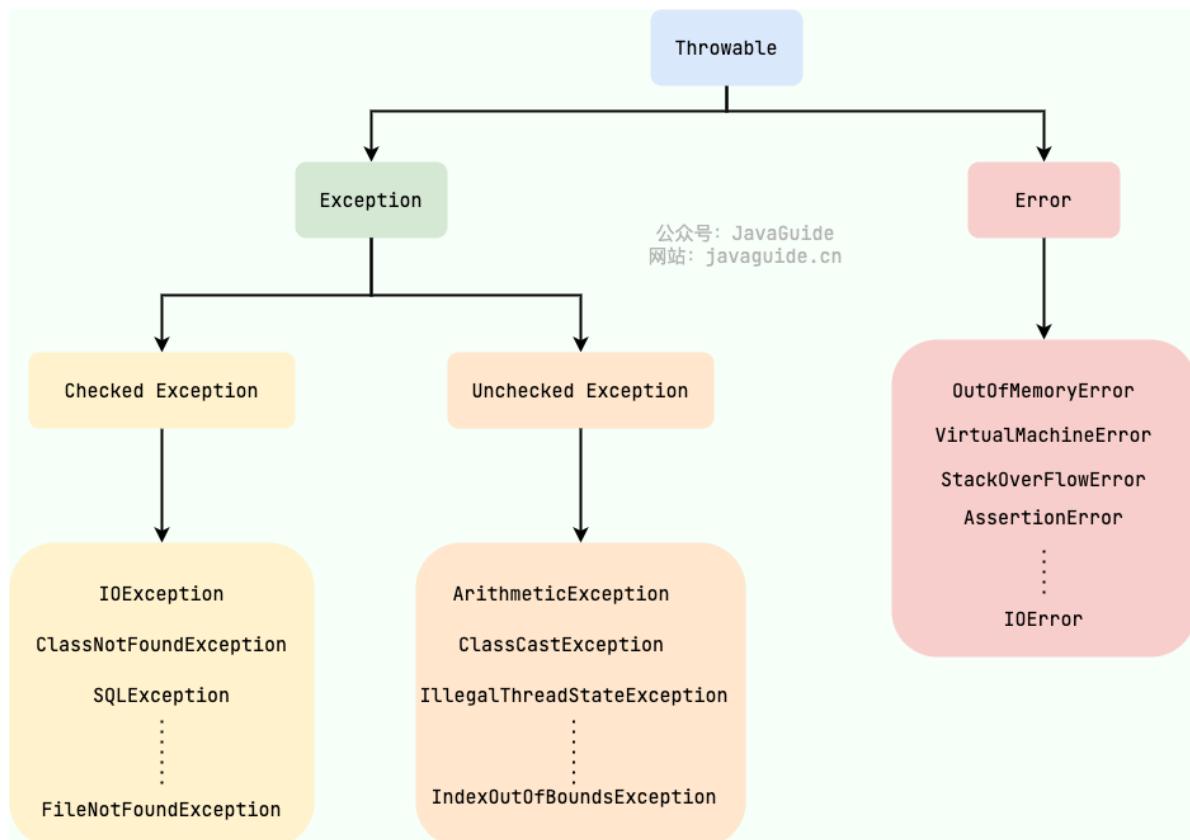
```
String str1 = "str";
String str2 = "ing";
String str3 = "str" + "ing"; // 指向常量池里的"string"
String str4 = str1 + str2; // 在堆上创建新String对象
String str5 = "string";
System.out.println(str3 == str4); // false
System.out.println(str3 == str5); // true
```

## 编译器优化

- 常量折叠：字符串常量（或者final修饰的）直接相加，会在编译时优化为结果。
- 引用（变量）相加`+`：优化为`StringBuilder.append().toString()`。
- `String.concat()`：拼接单个字符串比`+`高效，但是`+`已经会转为`StringBuilder`，所以实际使用较少。

但是for循环里的`+`不会优化，每次都会创造一个新的String对象！

## 异常



- `Error`无法处理，不建议`catch`，JVM会终止线程
- `Exception`是程序本身可以处理`catch`的异常
  - `Checked Exception`: 可以在编译时检查的异常，若不用`try catch throw`处理则无法通过编译
  - `Unchecked Exception`: 即`RuntimeException`及其子类

## finally

不能在finally用return，它会覆盖try catch的return

若在finally执行前，线程终止（比如 `System.exit()`）则不会执行finally

## try-with-resources

Java 中类似于 `InputStream`、`OutputStream`、`Scanner`、`PrintWriter` 等的资源都需要我们调用 `close()` 方法来手动关闭，推荐使用try-with-resources写法。资源会自动关闭。`catch finally` 在资源关闭后执行。

```
try (Scanner scanner = new Scanner(new File("test.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException fnfe) {
    fnfe.printStackTrace();
} finally {}
```

通过使用分号分隔，可以在 try-with-resources 块中声明多个资源。

## 泛型

使用泛型类来实现一个自定义接口的通用返回结果 `CommonResult<T>`，通过参数 `T` 来动态指定返回结果的数据类型。这种设计模式非常常见，用于统一 API 的返回结构，使其更加灵活和可扩展。

```
public class CommonResult<T> {
    private int code;           // 返回状态码
    private String message;     // 返回信息
    private T data;             // 泛型数据，具体类型由使用时决定

    // 构造器
    public CommonResult(int code, String message, T data) {
        this.code = code;
        this.message = message;
        this.data = data;
    }

    public CommonResult(int code, String message) {
        this(code, message, null);
    }

    // 静态方法 - 成功
    public static <T> CommonResult<T> success(T data) {
        return new CommonResult<>(200, "Success", data);
    }

    // 静态方法 - 失败
    public static <T> CommonResult<T> failed(String message) {
        return new CommonResult<>(500, message);
    }
}
```

```

// Getter and Setter
...

public T getData() {
    return data;
}

public void setData(T data) {
    this.data = data;
}

@Override
public String toString() {
    return "CommonResult{" +
        "code=" + code +
        ", message='" + message + '\'' +
        ", data=" + data +
        '}';
}
}

public static void main(String[] args) {
    CommonResult<String> result = CommonResult.success("This is a string
result.");
    System.out.println(result);
}

```

- <T> CommonResult<T> : 方法级别的泛型。这是泛型方法的声明，表示这个方法可以独立于类的泛型，动态接收一个泛型类型 T。方法调用时，编译器会根据传入的参数类型推断 T 的实际类型。在静态方法中需要用这种泛型，因为静态方法在类实例化之前就已经加载。
- CommonResult<T> (没有 <T> 声明) : 类级别的泛型。只有类 CommonResult 是泛型类的情况下，非泛型方法才能使用 T，而且 T 必须依赖类的泛型声明。

```

// 泛型类声明了泛型T
public class CommonResult<T> {
    // 此方法是静态的，需要额外声明为方法级别的泛型<T>
    public static <T> CommonResult<T> success(T data) { // 此参数data依赖于类声
明的泛型
        return new CommonResult<>(200, "Success", data);
    }
    public void setData(T data) { // 此参数data依赖于类声明的泛型
        this.data = data;
    }
}

```

# 反射

Java 反射 (Reflection) 是一种运行时的 API，允许程序在**运行时**检查和操作类、接口、方法和字段，而**不需要在编译时知道它们的确切类型**。通过反射，程序可以动态加载、实例化、调用对象的方法、访问和修改对象的字段，以及操作注解等。**常用于动态代理。**

核心类：

- `Class<T>`：类的字节码文件，可以获取类的元数据，比如类名、构造方法、方法、字段等。
  - `Class.forName("com.example.MyClass")` 常用于动态加载类
- `Field`：类的属性(字段)，可以获取或修改类或对象的字段。私有字段需要设置为可访问  
`field.setAccessible(true)`
- `Method`：类中的方法，可以调用类或对象的方法。私有方法需要设置可访问。
- `Constructor<T>`：代表类的构造方法，可以通过它创建类的实例。获取到后调用  
`constructor.newInstance()`

# 代理

使用**代理对象**来代替对**目标对象(real object)**的访问，这样就可以在不修改原目标对象的前提下，**增强目标对象的功能**。

## 静态代理

- 接口类规定需要完成的方法；实现类完成接口的基本实现；代理类在内部注入**实现类实例并加上扩展内容**
- 需要手动实现每一个方法的增强；一旦接口类改动则全盘改动；**每个目标类都要单独实现代理类**
- 编译时JVM就得到代理的class

## 动态代理

- 不需要针对每个目标类都单独创建一个代理类；并且也不必须实现接口类，可以直接代理实现类 (CGLIB)
- 运行时动态生成类字节码，并加载到JVM

### JDK动态代理

通过**接口**实现代理

**不适用于无接口的类**，原因见 `Proxy.newProxyInstance()` 参数

核心是 `InvocationHandler` 接口和 `Proxy` 类。

- 通过 `Proxy` 类的 `Proxy.newProxyInstance()` 方法对目标对象创建动态代理
- `Proxy` 类含有 `InvocationHandler` 接口；`InvocationHandler` 注入目标对象实例，重写 `invoke()` 方法，该方法的作用是利用**反射**调用目标对象的原生方法并执行自定义增强。

步骤：

1. 定义一个接口及其实现类；
2. 自定义 `InvocationHandler` 注入目标对象，重写 `invoke` 方法调用目标对象的原生方法并自定义一些处理逻辑；
3. 通过 `Proxy.newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)` 方法创建代理对象

```

public class DebugInvocationHandler implements InvocationHandler {
    private final Object target; // 代理类中的真实对象
    public DebugInvocationHandler(Object target) { this.target = target; }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
    InvocationTargetException, IllegalAccessException {
        System.out.println("before method " + method.getName());
        Object result = method.invoke(target, args);
        System.out.println("after method " + method.getName());
        return result;
    }
}

```

```

public static Object getProxy(Object target) {
    return Proxy.newProxyInstance(
        target.getClass().getClassLoader(), // 目标类的类加载器
        target.getClass().getInterfaces(), // 代理需要实现的接口，可指定多个
        new DebugInvocationHandler(target) // 代理对象对应的自定义
        InvocationHandler
    );
}

```

## CGLIB动态代理

通过生成目标类的子类并重写目标类的方法来实现代理的，不依赖接口。基于字节码生成。

**不适用于final修饰的类和方法、性能慢**，原因是它是通过生成目标子类来代理

核心是 `MethodInterceptor` 接口和 `Enhancer` 类。

步骤：

1. 定义一个类；
2. 自定义 `MethodInterceptor` 并重写 `intercept` 方法，`intercept` 用于拦截增强被代理类的方法，和 JDK 动态代理中的 `invoke` 方法类似；
3. 通过 `Enhancer` 类的 `create()` 创建代理类；

## 注解

注解本质是一个继承了 `Annotation` 的特殊接口。用于修饰类、方法或者变量，提供某些信息供程序在编译或者运行时使用。

1. **编译期**：javac在进行编译时处理一部分注解，如 `@override`、`@Deprecated` 等JDK内置的注解；Lombok的 `@Data`。
2. **类加载**：某些框架会在类加载时扫描注解并做处理，如Spring的依赖收集 `@Component`、`@Service`。
3. **运行时**：使用**反射API**读取注解并执行对应逻辑，如Spring的依赖注入 `@Autowired`、JUnit的 `@Test`、Hibernate的 `@Entity`。

```

// 1. 定义注解
@Retention(RetentionPolicy.RUNTIME) // 注解生命周期。此注解持续到运行时
@Target(ElementType.METHOD) // 指定此注解可以从Target获取的东西。此注解获取到方法
@interface MyAnnotation {

```

```

    String value() default "Hello";
}

// 2. 使用注解
class TestClass {
    @MyAnnotation(value = "Custom Message")
    public void testMethod() { }
}

// 3. 通过反射读取注解
public class AnnotationProcessor {
    public static void main(String[] args) throws Exception {
        Method method = TestClass.class.getMethod("testMethod");
        if (method.isAnnotationPresent(MyAnnotation.class)) {
            MyAnnotation annotation = method.getAnnotation(MyAnnotation.class);
            System.out.println("注解值: " + annotation.value()); // 输出: Custom
Message
        }
    }
}

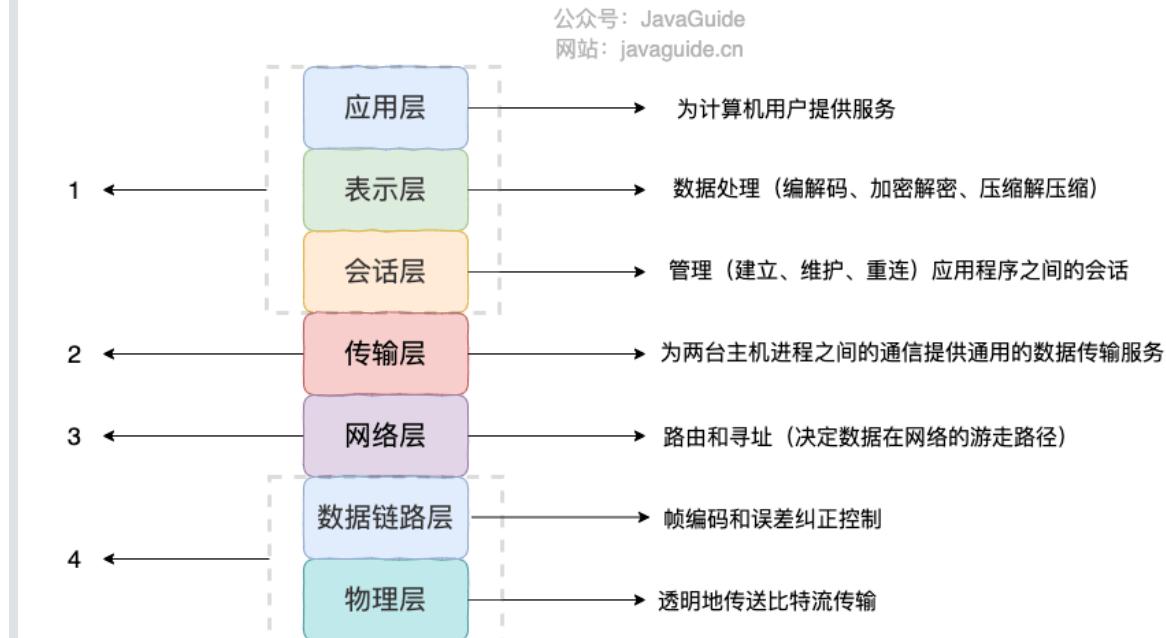
```

## 序列化

- **序列化**: 将数据结构或对象，转换成二进制字节流
- **反序列化**: 将在序列化过程中所生成的二进制字节流，转换成数据结构或者对象

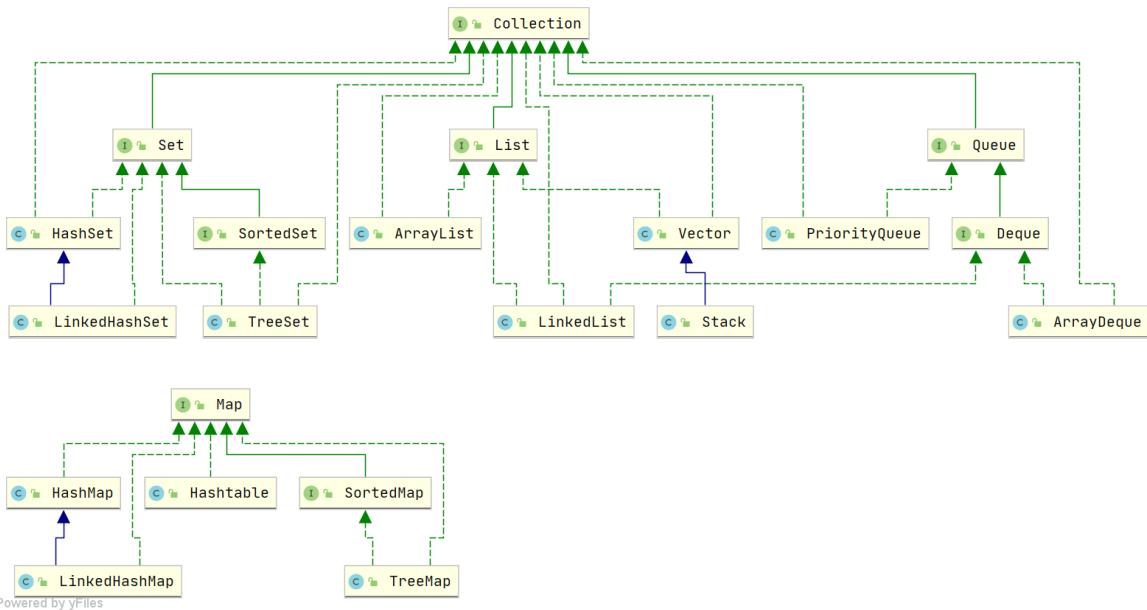
用于文本储存、网络传输等。JSON、XML是一种序列化方法，但是性能很差（可读性很强）

序列化是OSI7层里的表示层、TCP/IP的应用层(应用层、传输层、网络层、网络接口层)



对于不想进行序列化的变量，使用 `transient` 关键字修饰。

## 集合



- 我们需要根据键值获取到元素值时就选用 `Map` 接口下的集合，需要排序时选择 `TreeMap`，不需要排序时就选择 `HashMap`，需要保证线程安全就选用 `ConcurrentHashMap`。
- 我们只需要存放元素值时，就选择实现 `Collection` 接口的集合，需要保证元素唯一时选择实现 `Set` 接口的集合比如 `TreeSet` 或 `HashSet`，不需要就选择实现 `List` 接口的比如 `ArrayList` 或 `LinkedList`，然后再根据实现这些接口的集合的特点来选用。

## Comparable和Comparator

### Comparable

`Comparable` 接口只有一个方法 `compareTo()`，返回 `int` 值

**Comparable** 接口是在类自身实现的，定义了该类的自然排序。实现 `Comparable` 接口的类可以直接填入 `TreeMap`, `PriorityQueue`，否则需要指定 `Comparator`。

`String`、`Integer` 等类已经实现了 `Comparable` 接口

```

class Student implements Comparable<Student> {
    private String name;
    private int score;

    // constructor ...

    @Override
    public int compareTo(Student other) {
        return this.score - other.score; // 按分数排序
    }
}

List<Integer> students = new ArrayList<>();
... // 添加几个数据
Collections.sort(students);

```

## Comparator

`Comparator` 接口只有一个方法 `compare()`，返回 `int`

`Comparator` 在类的外部自定义排序规则。

```
class StudentNameComparator implements Comparator<Student> {  
    @Override  
    public int compare(Student s1, Student s2) {  
        return s1.getName().compareTo(s2.getName()); // 按名字排序  
    }  
}  
  
List<Integer> students = new ArrayList<>();  
... // 添加几个数据  
Collections.sort(students, new StudentNameComparator());
```

```
// 或者直接在sort()里写：  
Collections.sort(students, new Comparator<Student>(){  
    @Override  
    public int compare(...){...}  
})  
// 还可以简写，与上面效果相同  
Collections.sort(students, (s1, s2) -> a.score - b.score);
```

## Arrays.sort()

- `Arrays.sort()` 传入对象类型数组时，用法和 `Collections.sort()` 一致，可自定义排序，采用稳定排序（相同值不改变位置）；
- `Arrays.sort()` 传入基本类型数组时，默认升序排序、不支持自定义排序。它专门为基本类型设计了高效的排序算法，且是不稳定排序。`boolean[]` 不可排序。

## List/Deque

`List` 接口支持随机访问、中间插入、中间删除；`Deque` 接口专门面向首尾操作（双向队列操作）。

### ArrayList 与 LinkedList

- `ArrayList` 基于可变长数组实现，`LinkedList` 基于双向链表实现。
- `ArrayList` 随机访问为  $O(1)$ ，`LinkedList` 为  $O(n)$ 。
- `ArrayList` 在中间插入或删除需要移动元素，而 `LinkedList` 需遍历然后改变指针。虽然二者皆为  $O(n)$ ，但还是链表开销更小。

`ArrayList` 适合读多写少的场景，而 `LinkedList` 则适合频繁修改的场景。

### ArrayDeque 与 LinkedList

`ArrayDeque` 和 `LinkedList` 都实现了 `Deque` 接口，两者都具有双向队列的功能

- `ArrayDeque` 是基于可变长的循环数组和双指针来实现，而 `LinkedList` 则通过双向链表来实现。
- `ArrayDeque` 不支持存储 `NULL` 数据，但 `LinkedList` 支持。

- `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。
- `ArrayDeque` 支持栈操作

## BlockingQueue

阻塞队列`BlockingQueue`接口, 常用于生产-消费者模型, 当空队列消费或满队列生产时阻塞。

实现类`ArrayBlockingQueue`、`LinkedBlockingQueue`

- 一个基于静态数组 (有界、需指定size), 一个基于链表 (默认无界)
- `ArrayBlockingQueue`不支持锁分离 (生产锁、消费锁是同一个锁)

## 当HashMap以Object为key

1. **重写`equals()`和`hashCode()`:** `HashMap`依赖`hashCode()`计算存储位置和哈希冲突, 然后靠`equals()`判断key相等。而默认的`equals()`实际上是 == 判断Object内存地址, `hashCode()`默认也是根据地址计算。

```
class Person {
    String name;
    int age;
    @Override
    public boolean equals(Object obj) { // 重写equals, 判断是否内容相等
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Person person = (Person) obj;
        return age == person.age && Objects.equals(name, person.name);
    }
    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
}
```

2. **不要修改key:** 修改key内容造成hash重新计算, 导致map数据丢失查找失败。使用不可变对象如 `Integer` `String`; 类属性设置为final。
3. **不推荐用null:** 功能上支持用null作key, 固定存在`table[0]`、只能equal不能hashcode, 业务上可能造成空指针异常

# HashMap遍历

## Iterator迭代器

```
// entrySet
Iterator<Map.Entry<Integer, String>> iterator = map.entrySet().iterator();
while (iterator.hasNext()) {
    Map.Entry<Integer, String> entry = iterator.next();
    System.out.println(entry.getKey());
    System.out.println(entry.getValue());
}
// keySet...
```

## 增强for-each

```
// entrySet; keySet; values
for (Map.Entry<Integer, String> entry : map.entrySet()) {
    System.out.println(entry.getKey());
    System.out.println(entry.getValue());
}
```

不要在for-each里进行add/remove操作！！因为for-each底层实际上还是由Iterator执行，它有自己的安全的增删方法；假如在for-each中使用了集合的add/remove，会引起报错。

可以在Iterator里调用 `Iterator.remove()`，安全删除当前元素；或者直接 `list.removeIf(filter -> filter % 2 == 0);`

也可以通过 `map.keySet()`、`map.values()` 获取键或值的集合

## Lambda

```
map.forEach((key, value) -> {
    System.out.println(key);
    System.out.println(value);
});
```

## Stream和parallelStream

```
map.entrySet().stream().forEach(() -> {})
map.entrySet().parallelStream().forEach(() -> {})
```

parallelStream在多线程（存在阻塞）情况下性能最好

# ConcurrentHashMap

## 对比Hashtable

- HashTable
  - 底层结构：数组 + 链表
  - 线程安全实现：使用 `synchronized` 对整个HashTable加锁，阻塞率高、效率低下
- ConcurrentHashMap (JDK1.8)

jdk1.7及之前，ConcurrentHashMap由**分段数组** segment +链表构成（对整个桶数组分段，每个segment都是一个可扩容的桶数组结构；默认16个segment），读写时对当前segment加锁

- 底层结构：Node 数组 + 链表/红黑树
- 线程安全：

## 实现线程安全：JDK1.7与1.8

- JDK1.7：数据结构是纯**拉链法**，由**分段数组** segment +链表构成（对整个桶数组分段，每个segment都是一个可扩容的桶数组结构）。segment继承自 reentrantLock，读写时对当前segment加synchronized锁。最大并发度是segment个数，默认16，**一旦初始化不可变**。
- JDK1.8：使用**拉链法+红黑树**。对当前hash命中的Node加锁（synchronized+CAS），只锁定当前链表或红黑树根。最大并发度是Node个数。

## ConcurrentHashMap不允许null

- 键二义性：执行get(null)时，可能是key不存在，也可能是就是要key=null
- 值二义性：若get(key)=null，可能是值不存在，也可能存的就是null

hashMap单线程中可以通过containsKey()判断；但是多线程环境下，在containsKey()之后、map.get()之前，可能已经有别的线程修改了该键值。

**这也说明，这些方法的复合并不存在原子性。**

ConcurrentHashMap原子复合操作：putIfAbsent()

```
// not Thread-safe:  
if (!map.containsKey(key)) map.put(key, value);  
// Thread-safe:  
map.putIfAbsent(key, value);
```

如果就是想用null：

```
public static final Object NULL = new Object();
```

## 集合list - 数组array 转换

- 集合转数组：`<T> <T>[] toArray(T[] array)`

参数是一个泛型数组，需传入一个类型一致、长度0的空数组；如果无参数则返回的是Object数组。

```
String[] myArr = myList.toArray(new String[0])
```

- 数组转集合：`<T> List<T> asList(T... a)`

```
String[] myArray = {"Apple", "Banana", "Orange"};  
List<String> myList = Arrays.asList(myArray);  
//上面两个语句等价于下面一条语句  
List<String> myList = Arrays.asList("Apple", "Banana", "Orange");
```

asList返回的list不能add、remove、clear

再转为ArrayList

```
List list = new ArrayList<>(Arrays.asList("a", "b", "c"))
```

## ArrayList扩容

三种初始化方法：

- 无参：赋值一个 object[] 空数组，等到添加元素 add() 时再扩容
- 指定初始容量 initCapacity：赋值 new Object[initCapacity]
- 指定Collection：通过 Arrays.copyOf() 初始化

扩容：

- 触发扩容：添加元素操作 add() 触发扩容
- 计算所需最小容量 miniCapacity：若此前为空数组，则所需容量为默认容量 10；否则是当前数组大小+1
- 计算新容量 newCapacity：通过位运算扩大旧容量为1.5倍，取其和 miniCapacity 的较大值
- 内存溢出判断：newCapacity 若大于 Integer.MAX\_VALUE-8，视 miniCapacity 值缩减或报错
- 执行扩容：Arrays.copyOf()

## HashMap源码

底层架构：主体是一个桶数组 table，数组中每个桶 bucket 指向一个由 Node 构成的链表（或 TreeNode 红黑树），代表了hash冲突的多个键值对 entry。

HashMap类属性：

```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>,
Cloneable, Serializable {
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // 默认初始容量是16，即桶数组长度tableSize
    static final float DEFAULT_LOAD_FACTOR = 0.75f; // 默认负载因子(控制扩容的阈值)，entry数量达容量75%触发扩容

    static final int TREEIFY_THRESHOLD = 8; // 树化界限，当bucket上Node数量(链表长度)>=8，树化
    static final int UNTREEIFY_THRESHOLD = 6; // 非树化界限，当bucket上的TreeNode数量<=6，转为链表
    static final int MIN_TREEIFY_CAPACITY = 64; // 桶数组长度达到64才可以进行树化

    transient Node<k,v>[] table; // 存储元素的数组，tableSize总是2的幂次倍
    transient int size; // 存放元素的个数，注意这个size不等于数组的长度tableSize

    int threshold; // 容量阈值(capacity*loadFactor)。当size超过阈值时，会进行扩容。
    final float loadFactor; // 负载因子
}
```

HashMap里没有capacity字段，真正的容量指的是table的长度tablesize。需要注意的是，tableSize往往比size大。

tablesize每次更改，都会调用tablesizeFor()将其调整为2的幂次倍。

### HashMap初始化：

HashMap有4种构造方法：

```
public class HashMap<K,V> {  
    // 默认构造函数。  
    public HashMap() {  
        this.loadFactor = DEFAULT_LOAD_FACTOR;  
    }  
  
    // 包含另一个“Map”的构造函数  
    public HashMap(Map<? extends K, ? extends V> m) {  
        this.loadFactor = DEFAULT_LOAD_FACTOR;  
        putMapEntries(m, false);  
    }  
  
    // 指定“容量大小”的构造函数  
    public HashMap(int initialCapacity) {  
        this(initialCapacity, DEFAULT_LOAD_FACTOR);  
    }  
  
    // 指定“容量大小”和“负载因子”的构造函数  
    public HashMap(int initialCapacity, float loadFactor) {  
        if (initialCapacity < 0)  
            throw new IllegalArgumentException("Illegal initial capacity: " +  
initialCapacity);  
        if (initialCapacity > MAXIMUM_CAPACITY)  
            initialCapacity = MAXIMUM_CAPACITY;  
        if (loadFactor <= 0 || Float.isNaN(loadFactor))  
            throw new IllegalArgumentException("Illegal load factor: " +  
loadFactor);  
        this.loadFactor = loadFactor;  
        // 初始容量暂时存放到threshold，在resize()中再赋值给newCap进行table初始化  
        this.threshold = tablesizeFor(initialCapacity);  
    }  
}
```

### HashMap初始化：

在HashMap创建时，最开始只是赋值loadFactor，将初始容量暂存threshold；此时并没有真正初始化table，它还只是一个null，等到插入元素时再进行resize()（它同时承担了初始化桶数组和扩容的功能）。

对于使用一个现成的Map初始化的情况，也是先保证threshold满足大小，然后挨个插入元素。

### 元素插入putVal()：

0. 触发：调用put(key, value)时触发putVal()
1. 判断初始化：若此前刚进行初始化，桶数组为null或空，先resize()扩容
2. 插入元素：

- **hash冲突计算**: key的 hashCode() + 扰动函数 => hash 值;  $(n - 1) \& hash$  得到该元素存放位置 ( $n$  是数组长度)。
- 若key在桶数组中hash不冲突 (该bucket位置处是个 null)，直接插入该entry;
- 若hash冲突、存在该key，覆盖、结束
- 若hash冲突、不存在key，插入到链表尾部或红黑树中
  - 若插入后，链表长度达到 8 且桶数组长度达到 64，则树化 treeifyBin()
  - 若链表长度达到 8 但桶数组长度未到 64，则直接 resize() 扩容

3. 判断是否需要扩容：元素个数是否超过负载阈值 threshold，超过则 resize() 扩容。

**核心：**元素数量过载则 resize() 扩容，某桶链表过长则 treeifyBin() 树化；但是桶数组长度较短时优先扩容。总的来说都是要使链表不能长，链表太长了优先扩容、再考虑树化。

扩容 resize() :

1. 计算新容量 newCap 和新阈值 newThr :

- 若旧容量 oldCap = 0，说明桶数组未初始化，取上述暂存 threshold (无值则默认16) 作为 newCap，更新 threshold 为当前实际阈值；
- 若 oldCap > 0 则翻倍，更新阈值。

2. 创建新数组：根据算出的新容量，new 一个新的桶数组

3. 重新分配 rehash :

- 若是单节点， $hash \& (newCap - 1)$  计算出新位置，直接移过去；
- 若是树节点，执行 TreeNode#split() 分裂成俩子树 (一留一动，机制同下)，若子树节点小于 6 转为链表；
- 若是链表节点，判断  $hash \& oldCap == 0$  则留在原地，否则移动到 原 hash 位置 + oldCap。
  - 因为 oldCap 是 2 的次幂，只有 1 位是 1，概率上平均将链表分为两块。

## JDK1.7 vs 1.8

查找性能优化、安全性改进、扩容效率提升

对比	JDK1.7	JDK1.8
底层结构	<b>数组+链表（链表散列）</b> 特殊情况下链表少但长、造成搜索时间过长	<b>数组+链表/红黑树</b>
链表插入	<b>头插法</b> 并发插入时可能指向错误、形成环形链表，导致查询死循环	<b>尾插法</b> 并发下只可能覆盖，不会死循环
hash扰动函数	4次扰动较复杂	进行优化、减少计算
扩容 rehash	<b>逐一重新计算hash分配</b>	<b>仅计算一bit分成两桶，一桶移动到新位置</b>
线程安全	不安全，且可能环形链表死循环	也不安全

## LinkedHashMap

LinkedHashMap是基于HashMap的，顺便一起写了。

**LinkedHashMap继承HashMap，并在此基础上维护了一条双向链表**（内部类 Entry 增加 before,after 指针），具备以下特性：

1. 支持遍历时会按照插入顺序有序进行迭代，默认顺序
2. 支持按照元素访问顺序排序迭代（可用于封装LRU缓存），通过 accessOrder 启用

**构造方法入参：** initCapacity, loadFactor, accessOrder。容量和负载因子同HashMap；  
accessOrder 代表是否按访问顺序排序，默认为 false。

**维护双向链表：** afterNodeRemoval、 afterNodeInsertion、 afterNodeAccess，作为访问、添加、删除的后置操作

`afterNodeAccess()`

accessOrder 为 true 时，此函数才会被启用

- get() 或 put() 方法触发 afterNodeAccess()
- 获取当前访问节点 p，将 p 的前驱和后继节点关联起来（注意 p 为首或尾的情况），将 p 独立出来。  
然后将 p 放到末尾，更新 p 的前后指针、当前链表的尾指针

`afterNodeInsertion()`

- put() 方法中，插入的 key 在 map 中存在，直接覆盖并调用 afterNodeAccess()；若是插入新 entry，则再触发 afterNodeInsertion()
- 通过 removeEldestEntry() 判断是否需要移除最老元素。removeEldestEntry() 默认不移除，一般自己重写以实现 LRU

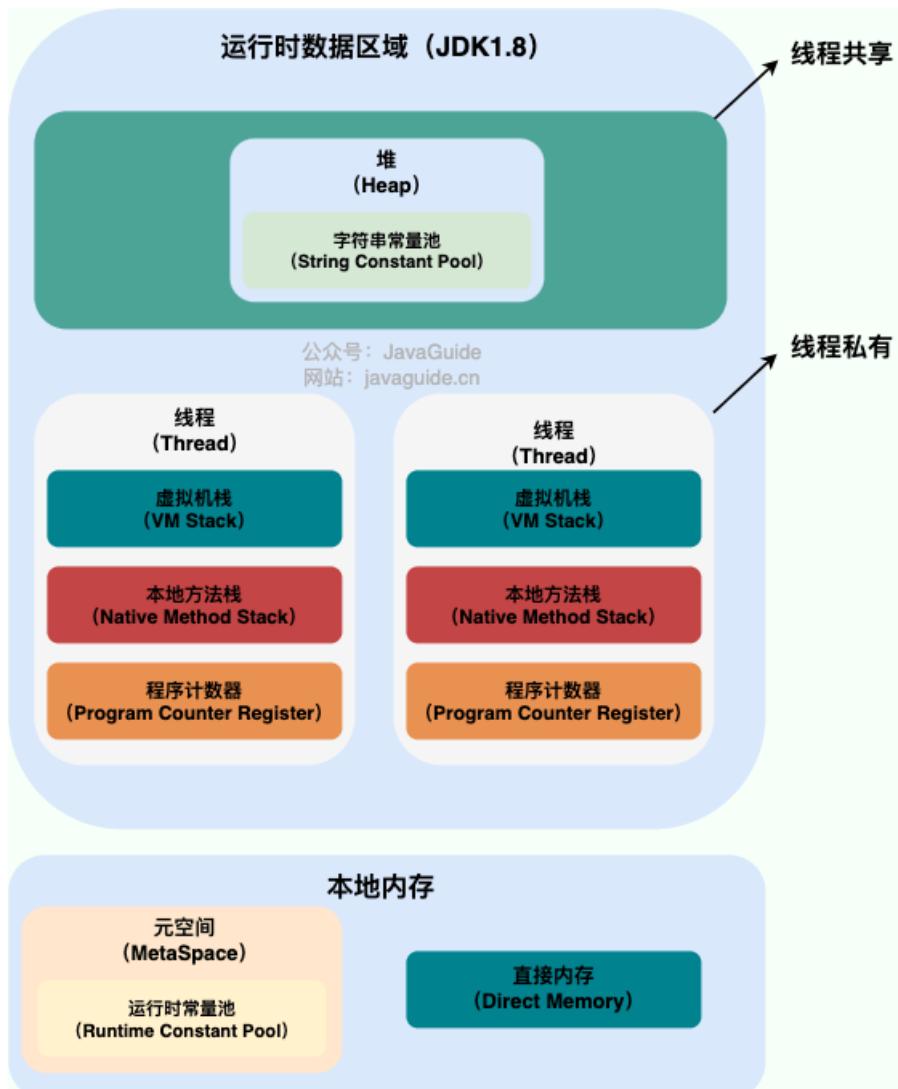
`afterNodeRemoval()`

- 让当前节点 p 的前驱节点、后继节点直接相连，p 断开联系独立出来，等待 gc 回收

## 并发

### 进程与线程

进程是系统运行程序（资源分配）的基本单位；线程是 CPU 调度的基本单位。



程序计数器私有主要是为了线程切换后能恢复到正确的执行位置。

虚拟机栈和本地方法栈是线程私有的，为了保证保证线程中的局部变量不被别的线程访问到。

## 如何创建线程

创建线程体（线程需要运行的任务）的方法：

- 继承 `Thread` 类，重写 `run` 方法
- 实现 `Runnable` 接口，重写 `run` 方法
- 实现 `Callable` 接口，重写 `call` 方法
- `ExecutorService` 线程池

```
ExecutorService poolA = Executors.newFixedThreadPool(2);
poolA.execute(() -> {
    System.out.println("4A.....");
});
poolA.shutdown();
```

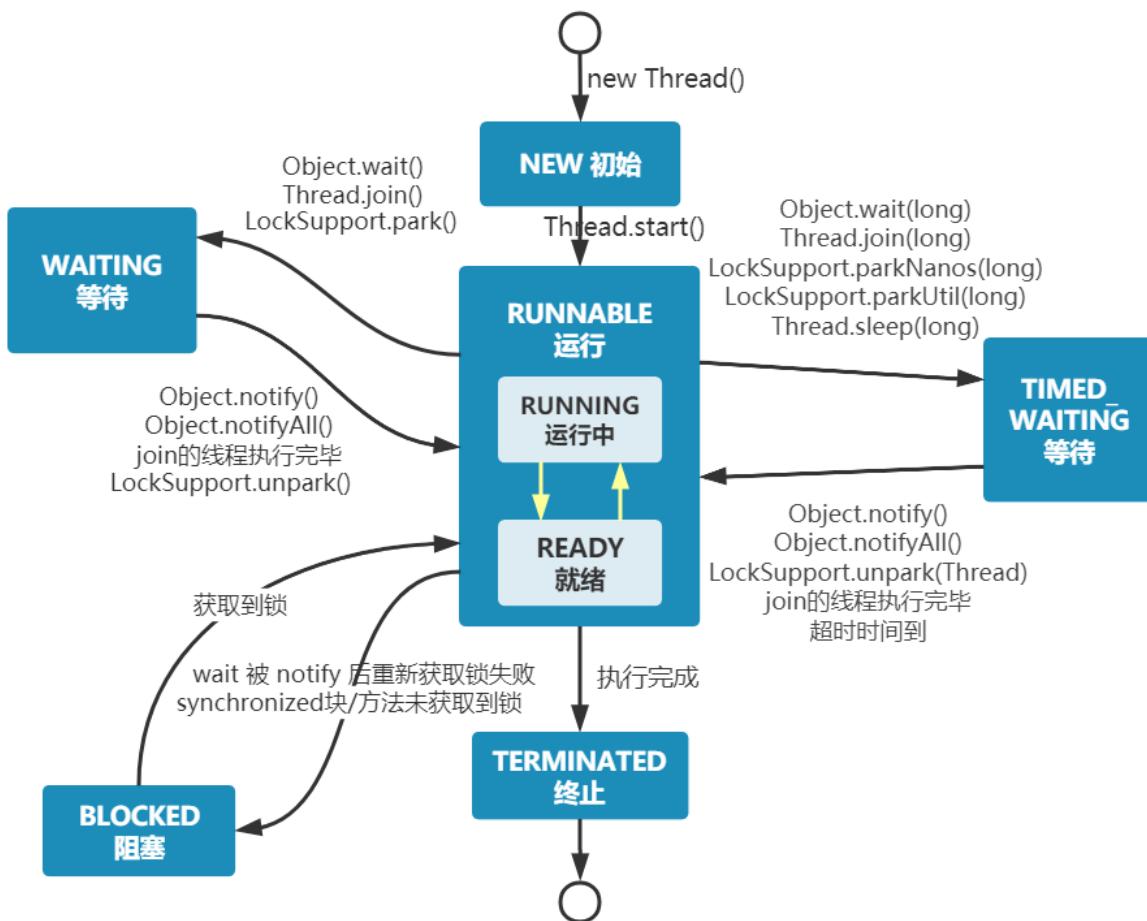
- lambda: `new Thread(() -> {}).start();`

.....

**创建线程的方法是唯一的:** `new Thread().start()`, 其他都会调用这一步

不能直接调用 `run()`: 调用 `start()` 可启动线程并使线程进入就绪状态, 直接执行 `run()` 不会以多线程的方式执行

## 线程生命周期和状态



## Thread#sleep()和Object#wait()

- 两者都可以暂停线程的执行。`sleep()` 方法没有释放锁, 而 `wait()` 方法释放了锁 (只能在 `synchronized` 里用)。
- `wait()` 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()`; `wait(long time)` 和 `sleep()` 会自动苏醒
- `sleep()` 是 `Thread` 类的静态本地方法。因为它是要暂停当前线程, 和对象类、对象锁无关
- `wait()` 是 `Object` 类的本地方法。

每个对象都有一把对象锁, 确保某个时刻只有一个线程可以访问对象的临界区。对象常作为线程共享的资源, 线程间的安全通信和访问基于对象的锁来实现; 要释放当前线程占用的对象锁, 是要操作对象而非线程。 **线程通信的本质是通过对象上的锁进行的, 而不是线程本身。**

# 死锁

形成死锁必须的四个条件：

- **互斥Mutual Exclusion**: 一个资源同时只能被一个线程访问（东西不能共享）  
通常不可破坏
- **请求与等待Hold and Wait**: 线程已经持有至少一个资源，同时又在等待其他资源（等着抢别人的）  
一次性申请全部资源
- **不可剥夺No Preemption**: 线程获得某资源后不可被其他线程剥夺，直到该线程完成任务主动释放（自己拿的不能被人抢）  
申请不到自己释放
- **循环等待Circular Wait**  
按序申请

# 单核CPU运行多线程

- 单核 CPU 是支持 Java 多线程的。操作系统通过**时间片轮转**的方式，将 CPU 的时间分配给不同的线程。
- Java 采用**抢占式调度**：由系统时钟中断（时间片轮转）或其他高优先级事件（如 I/O 操作完成）触发调度、切换进程。
- 但是单核 CPU 运行多线程不一定会提高效率，取决于线程类型：
  - **CPU密集型**：主要进行计算和逻辑处理，需要占用大量的 CPU 资源。多线程同时运行会导致频繁的线程切换，增加了系统的开销，降低了效率
  - **IO密集型**：主要进行输入输出操作，如读写文件、网络通信等，需要等待 IO 设备的响应。多线程同时运行可以利用 CPU 在等待 IO 时的空闲时间，提高了效率

# JMM内存模型

## 并发编程三特性（问题）

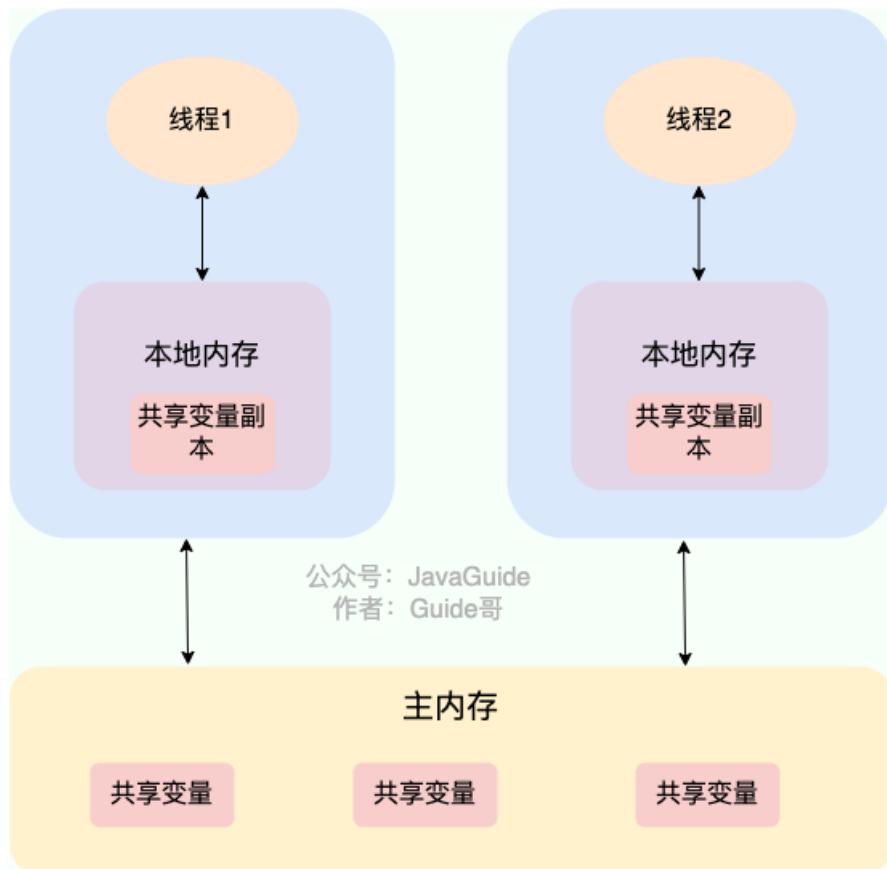
**原子性**：要么全执行、要么不执行，不会被打断。通过 `synchronized`、各种 `Lock` 以及各种原子类（CAS实现）实现原子性。

**可见性**：线程对共享变量修改后，其他线程立刻看到修改后的值。通过 `synchronized`、`volatile` 以及各种锁实现。`volatile` 修饰的变量会在每次访问时都到主存中读取。

**有序性**：指令重排优化后串行语义一致，但多线程不一定。`volatile` 禁止指令重排序。

## JMM的用处

在并发编程下出现了上述问题，JMM定义了并发编程的规范，同时也是一种线程和主内存之间的关系的抽象。JMM简化了多线程编程。



## JMM和JVM内存结构

- JVM内存区域定义了JVM运行时如何存放数据
- JMM抽象了线程和主存的关系，定义了并发编程规范

## happens-before原则

**设计思想：**如果指令重排序改变执行结果，就禁止；否则任凭编译器和处理器重排序优化。这在程序员和编译器/处理器之间取得平衡：不影响效率，也不会增加程序员编程复杂度（正确性）。

**happens-before原则定义：**若一个操作 happens-before 另一个操作，那么**第一个操作的执行结果将对后一个可见，从程序员视角来看二者顺序执行。**（若重排序不影响顺序执行结果，可以不阻止重排序；在程序员视角看不到这样的影响）

**happens-before规则：**不满足规则视为非顺序操作，可以重排序

1. **线程内顺序规则：**一个线程内，按照代码顺序执行，前面的操作 happens before 后面的操作
2. **传递规则：** A happens before B, B happens before C，则A happens before C
3. **解锁在前规则：**解锁 happens before 加锁
4. **线程启动规则：** Thread对象的 `start()` happens before 此线程的任何操作
5. **volatile规则：** volatile变量的写操作，happens before之后对其的读操作

## volatile关键字

- `volatile` 关键字可以保证**变量的可见性**：它指示 JVM，这个变量是共享且不稳定的，每次使用它都到**主存**中进行读取。
  - `volatile` 只能保证可见性，不能保证原子性。`synchronized`两者都能保证。
- `volatile`可以防止**JVM的指令重排序**

**双重校验锁实现懒加载单例（线程安全）**

```

public class Singleton {
    private volatile static Singleton uniqueInstance;
    private Singleton() {}

    public static Singleton getInstance() {
        //先判断对象是否已经实例过，没有实例化过才进入加锁代码
        if (uniqueInstance == null) {
            //类对象加锁
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}

```

`uniqueInstance = new Singleton()` 其实是三步：

1. 为 `uniqueInstance` 分配空间
2. `uniqueInstance` 初始化
3. `uniqueInstance` 指向地址

指令重排序可能扰乱它们的顺序。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 `getInstance()` 后发现 `uniqueInstance` 不为空，因此返回 `uniqueInstance`，但此时 `uniqueInstance` 还未被初始化。

## synchronized关键字

保证修饰的方法或者代码块在同时只能有一个线程执行

1. 修饰**实例方法**（锁当前**对象实例**）

```
synchronized void method() {...}
```

2. 修饰**静态方法**（锁当前**类**）

```
synchronized static void method() {...}
```

3. 修饰**代码块**

- 指定**锁对象**：`synchronized(object) {...被修饰的代码块...}`
- 指定**锁类**：`synchronized(类.class) {...被修饰的代码块...}`

```
synchronized(this) {...}
```

**构造方法不能被 synchronized 修饰。**构造方法本身是线程安全的，但其内部若访问共享资源，可以在内部用 `synchronized` 代码块。

**锁的对象应该尽可能细粒度（根据业务决定），以提高并发性能。**

## synchronized原理与锁升级

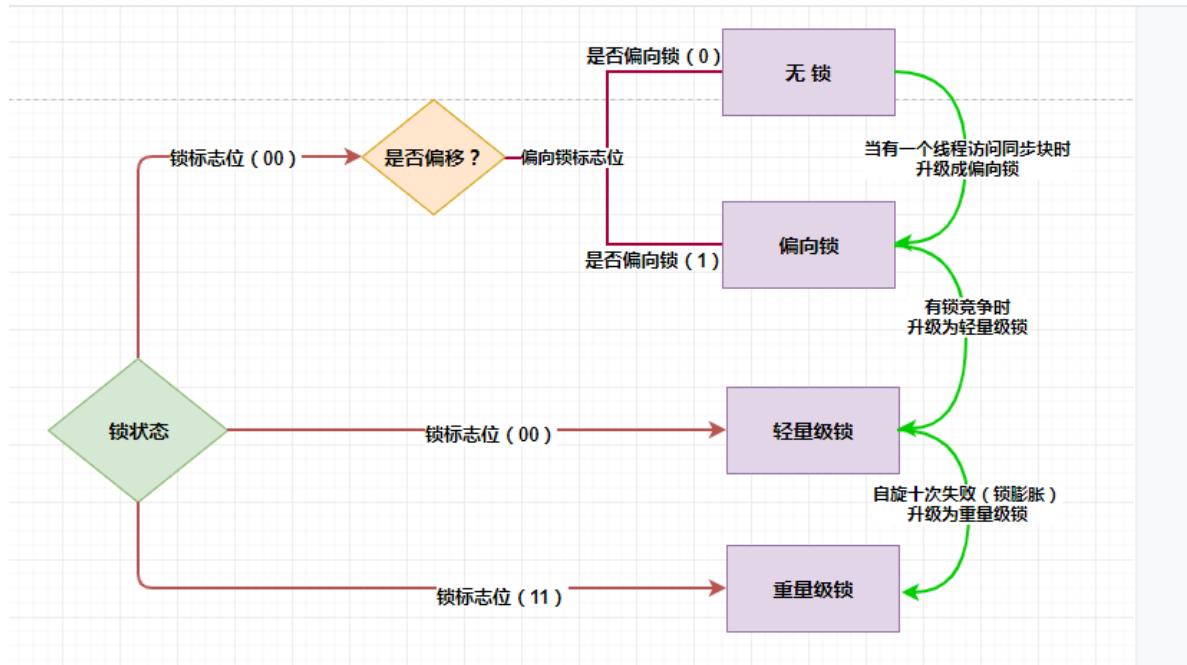
**synchronized**同步语句块：在代码块前后加上 `monitorenter` 和 `monitorexit` 指令。执行 `monitorenter` 时会尝试获取对象的锁（即对对象监视器 `monitor` 的持有权）、获取失败则阻塞等待；执行 `monitorexit` 则释放。

**监视器锁 monitor**：一种同步机制，每个Java对象都有一个对应的 monitor；monitor 中的 owner 字段存放持有该锁的线程ID，表示该锁被该线程占用。

锁升级

JDK1.6前，synchronized效率非常低下：monitor依赖底层OS的MutexLock互斥锁实现，而OS切换线程需要从用户态转到核心态，性能开销极高。**依赖OS Mutex Lock的锁称为重量级锁。**

JDK1.6对synchronized进行了性能优化，加入了**锁升级机制**：**无锁 - 偏向锁 - 轻量级锁 - 重量级锁**，提高了锁的获取与释放的效率。



**synchronized锁的存储位置：Java对象头。**Java对象在内存中由三部分组成：对象头、实例数据、对齐填充；对象头由标记字段 `Markword` 和类型指针 `KlassPointer` 构成，其中 `Markword` 存储了锁标志位信息，是实现 `synchronized` 锁机制的基础。随着锁状态变化，`Markword` 数据也会动态变化，32位的情况如下（64位会扩展25bit的那一部分）：

## 32位虚拟机

锁状态	25bit		4bit	1bit (是否偏向锁)	2bit (锁标志位)
	23bit	2bit			
无锁	对象 hashCode		对象分代年龄	0	01
偏向锁	threadId(偏向锁的线程ID)	Epoch	对象分代年龄	1	01
轻量级锁	指向栈中锁的记录的指针				00
重量级锁	指向重量级锁的指针				10
GC 标志	空				11

## 锁升级的四种锁分类

- **无锁**: 没有synchronized时的情况。没有对该对象进行锁定，所有线程可以访问修改同一资源，但同时只有一个线程成功，其他线程会不断循环尝试修改直至成功。

- **偏向锁BiasedLock**: “偏向于第一个获得它的线程”的锁，专用于单线程访问同步代码的优化。
  - 执行到synchronized代码块时，通过CAS修改Markword加偏向锁，在其中写入占有该锁的线程ID。
  - 无竞争时，锁会在一段时间内保持偏向（表现为ThreadID），后续同一线程再执行同步代码则无需重新获取锁，降低了开销。因此，只有一个线程访问同步代码块时，偏向锁的性能极好。
  - 当其他线程尝试获取偏向锁时，JVM会撤销偏向、进行锁升级；或是等待不活跃的安全时间点退回无锁状态。
- **轻量级锁（自旋锁）**
  - 偏向锁出现竞争时升级为轻量级锁。竞争线程会不断自旋、继续竞争。获取轻量级锁通过CAS修改Markword锁标志位完成。
  - 自旋是一种忙等、空耗CPU，短时间的忙等是可以容忍的，比切换用户态核心态的开销小。
- **重量级锁**
  - 竞争轻量级锁时一直自旋会空耗CPU，自旋一定次数（默认10）则升级为重量级锁。竞争线程遇到重量级锁则直接挂起、阻塞、等待唤醒。
  - 重量级锁依赖OS的Mutex Lock。重量级锁就是把线程调度控制权交给了OS，OS进行线程状态的变更开销较大。

**锁升级是单向的，锁状态不会降级回去。**这是JVM的设计选择，用于简化锁的实现和管理。

## synchronized和volatile

- `volatile`是轻量级的同步工具，性能更好。因此二者可以互补使用。
- `volatile`只能修饰变量，`synchronized`可以修饰变量、方法、代码块
- `volatile`只保证可见性，不保证原子性。`synchronized`两者都能保证。前者注重多线程间的可见，后者注重多线程的同步性。

## 乐观锁、悲观锁

**悲观锁**：任何访问都要加锁，共享资源同时只给一个线程使用。适用于多写场景。

悲观锁的实现：`synchronized` 和 `ReentrantLock` 等独占锁

**乐观锁**：访问资源没问题，只有提交修改才需要验证。适用于多读场景。

乐观锁的实现：

- 版本号机制：用`version`存储版本（修改次数），提交修改时检查`version`是否相等。
- **CAS算法**

## CAS

- *compare and swap*：比较变量V的期望值E和实际值V，相等才更新为新的值N，否则放弃更新。
- CAS是一个原子操作，在硬件侧面得到原子性保证。
- 在Java中，CAS由`unsafe`类实现，实际使用了`native`方法（C/C++），调用操作系统和CPU底层的指令来实现。

**自旋锁机制**：由于CAS操作可能会因为并发冲突而失败，因此通常会与 `while` 搭配使用，在失败后不断重试直到成功。

## CAS的局限性

- **ABA问题**：若V初始为A，中间改为了B，后面又改回了A，此时CAS操作会认为没有修改。
  - 在变量前面追加上版本号或者时间戳
- **长等待问题**：自旋锁while循环时间过长
  - 有限自旋：设置自旋次数上限
  - 退避算法：自旋失败后等待一段递增的时间，减少竞争强度
- **仅能操作单个共享变量**
  - `AtomicReference` 支持引用对象之间的原子性，将多个变量装在一个对象中
  - 加锁

## 公平锁、非公平锁

- **公平锁**：先申请的线程先拿到锁。需要维护队列，增加了切换上下文和调度的开销，性能差。  
适用于高公平性场景，如任务调度。
- **非公平锁**：按照随机顺序抢锁，也有可能是固定的优先级；性能较高。  
适用于高性能，且竞争不激烈的场景。

## 可重入锁、不可重入锁

- **可重入锁**：递归锁，线程可以获取其内部的（已经获得过的）锁
- **不可重入锁**：重入时会死锁。

## 共享锁、独占锁

- **共享锁**：一把锁可以被多个线程同时获得
- **独占锁**：一把锁同时只能给一个线程获得

## 读锁、写锁

- **读锁**：共享锁，允许多个读，但是同时不能有写
- **写锁**：独占锁，不能有其他读或写
- 线程已有读锁时，不能继续获取写锁（写锁是独占锁，获取写锁会先查看是否已有锁，不管是否是自己的）
- 线程有写锁时，可以再获取读锁

读写锁只能**锁降级**（写降为读，用于写后读取、减少阻塞），不能**锁升级**（读升为写不可以，多线程同时锁升级会死锁）

`synchronized`可以**锁升级**，因为它是单一互斥锁、不区分读写，且锁状态由JVM根据竞争情况动态优化。锁升级的本质是JVM改变锁的实现形式（偏向锁→轻量级锁→重量级锁）以适应并发需求。一个锁只能由一个线程持有，避免了死锁。

## ReentrantLock

`ReentrantLock` 实现了 `Lock` 接口，是一个可重入且独占式的锁，和 `synchronized` 关键字类似。但 `ReentrantLock` 更灵活强大，加了轮询、超时、中断、公平锁和非公平锁等高级功能。

### ReentrantLock和synchronized

- `synchronized`在JVM层面实现，`ReentrantLock`在JDK API层面实现
- `ReentrantLock`可以是可中断锁（等待锁的线程可以放弃等待）
- `ReentrantLock`可指定公平锁/非公平锁：`new ReentrantLock(true)` 指定为公平锁；默认非公平锁。  
`synchronized`只能是非公平锁。

## ThreadLocal

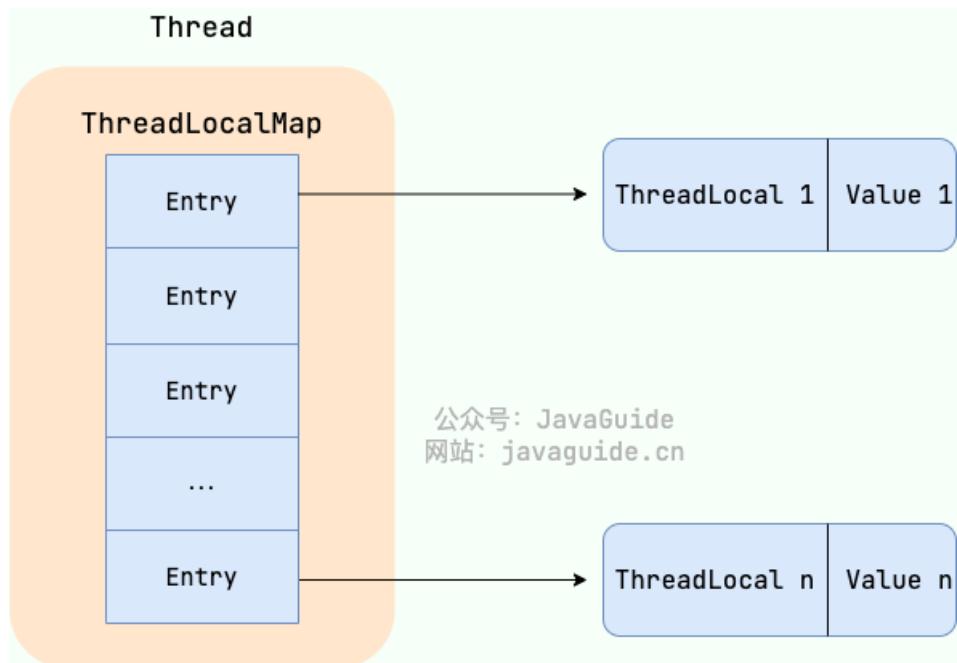
`ThreadLocal` 类实现了线程的局部变量，每个线程都会存储该变量的一个单独的、私有的数据副本，实现线程隔离。

一个 `ThreadLocal` 实例代表了被存储的单个变量。

```
// 这个ThreadLocal是存储当前这个User的线程域变量
class UserHolder {
    private static ThreadLocal<User> t1 = new ThreadLocal<>();
    public static void saveUser(User u) { t1.set(u); }
    public static User getUser() { return t1.get(); }
    public static removeUser() { t1.remove(); }
}
```

原理：

- `Thread` 类中有一个 `ThreadLocalMap` 类型的成员变量 `threadLocals`，保存了当前线程所有的 `ThreadLocal` 变量（以 `<ThreadLocal, Object>` 键值对的形式存储在其中）
- 在 `ThreadLocal#set()` 和 `ThreadLocal#get()` 中，获取当前线程的 `ThreadLocalMap` 来获取自己的局部变量



### ThreadLocal内存泄露问题

`ThreadLocalMap` 中，**key为弱引用，value是强引用**。所以，如果 `ThreadLocal` 没有被外部强引用的情况下，在垃圾回收的时候，key会被清理掉，而value不会被清理掉，造成 `<null, value>` 存在map中。现在线程大都由线程池创建，会进行**线程重用**，也`ThreadLocal`就会留在内存里，造成内存泄漏。

`ThreadLocalMap` 考虑到了这种情况，在调用 `set()`、`get()`、`remove()` 时会处理掉。**但最好还是记得手动 `remove()`**。

## 线程池

池化技术：减少每次获取资源的消耗，提高对资源的利用率

线程池就是管理一系列线程的资源池。线程池中的线程和线程要执行的任务是相分离的，线程池对线程统一分配资源，当有任务要处理时，直接从线程池中获取线程来处理，处理完之后线程并不会立即被销毁，而是等待下一个任务。

### 创建线程池的方法

- 内置线程池：通过 `Executors` 类静态方法创建 `FixedThreadPool`、`singleThreadExecutor`、`CachedThreadPool`、`ScheduledThreadPool`（不推荐）
  - 不能灵活指定 `corePoolSize`、`maximumPoolSize`、`workQueue` 等参数
  - `FixedThreadPool/SingleThreadExecutor/ScheduledThreadPool` 使用的阻塞队列/延迟阻塞队列，实际上是无界的，任务量过大时可能堆积过多任务请求，造成OOM
  - `CachedThreadPool` 使用同步队列，`corePoolSize` 为 0、`maximumPoolSize` 无界，可创建无上限的线程数，造成OOM

**核心缺陷：没有使用有界队列、没有控制线程数**

- 通过 `ThreadPoolExecutor` 构造函数来创建线程池（推荐）

```
// 用给定的初始参数创建一个新的ThreadPoolExecutor
public ThreadPoolExecutor(
    int corePoolSize, //线程池的核心线程数量
    int maximumPoolSize, //线程池的最大线程数
    long keepAliveTime, //当线程数大于核心线程数时，多余的空闲线程存活的最长时间
    TimeUnit unit, //keepAliveTime参数的时间单位
    BlockingQueue<Runnable> workQueue, //任务队列，用来储存等待执行任务的队列
    ThreadFactory threadFactory, //线程工厂，用来创建线程，一般默认即可
    RejectedExecutionHandler handler //拒绝策略，当提交的任务过多而不能及时处理时，可定制策略来处理任务
) {...}
```

- 任务量  $\leq$  corePoolSize，等待任务量 = 0，直接在核心线程执行任务；核心线程随任务创建，即使空闲也不销毁。
- 等待任务量  $\leq$  workQueue容量，使用任务队列存储等待中的任务，线程池依然是 corePoolSize个线程。
- 等待任务量 > workQueue容量，线程池容量扩容至maximumPoolSize；多余线程随任务创建，keepAliveTime后销毁。



## 线程池的拒绝策略

`ThreadPoolExecutor` 在线程池达到最大、任务队列也达到最大容量时，会调用 `handler` 的拒绝策略

- `AbortPolicy`：抛出异常 `RejectedExecutionException`，拒绝新任务的处理
- `DiscardPolicy`：直接丢弃新任务
- `DiscardOldestPolicy`：丢弃最早的未处理的任务请求
- `CallerRunsPolicy`：尝试调用自己的线程（产生线程池的主线程）来执行任务
  - 可能会“阻塞”主线程的正常运行
  - 多余的任务保存在阻塞队列 `BlockingQueue` 中，可能会OOM

策略：增大 `BlockingQueue` 和堆内存、`maximumPoolsize`，以防止OOM

若 `CallerRunsPolicy` 导致服务器资源耗尽，需要考虑**任务持久化**，存起来、有余力再取出来执行：

- 设计任务表，将任务存到MySQL数据库（自定义拒绝策略，多余任务入库；重写阻塞队列使得先取出数据库中最早的任务）
- Redis 缓存任务
- 将任务提交到消息队列中

## 阻塞队列

名称	描述
ArrayBlockingQueue	一个用数组实现的有界阻塞队列，此队列按照先进先出(FIFO)的原则对元素进行排序。支持公平锁和非公平锁。
LinkedBlockingQueue	一个由链表结构组成的有界队列，此队列按照先进先出(FIFO)的原则对元素进行排序。此队列的默认长度为 Integer.MAX_VALUE，所以默认创建的该队列有容量危险。
PriorityBlockingQueue	一个支持线程优先级排序的无界队列，默认自然序进行排序，也可以自定义实现compareTo()方法来指定元素排序规则，不能保证同优先级元素的顺序。
DelayQueue	一个实现PriorityBlockingQueue实现延迟获取的无界队列，在创建元素时，可以指定多久才能从队列中获取当前元素。只有延时期满后才能从队列中获取元素。
SynchronousQueue	一个不存储元素的阻塞队列，每一个put操作必须等待take操作，否则不能添加元素。支持公平锁和非公平锁。SynchronousQueue的一个使用场景是在线程池里。Executors.newCachedThreadPool()就使用了 SynchronousQueue，这个线程池根据需要（新任务到来时）创建新的线程，如果有空闲线程则会重复使用，线程空闲了60秒后会被回收。
LinkedTransferQueue	一个由链表结构组成的无界阻塞队列，相当于其它队列，LinkedTransferQueue队列多了transfer和tryTransfer方法。
LinkedBlockingDeque	一个由链表结构组成的双向阻塞队列。队列头部和尾部都可以添加和移除元素，多线程并发时，可以将锁的竞争最多降到一半。

- `LinkedBlockingQueue` : `FixedThreadPool` 和 `singleThreadExecutor` 使用
- `SynchronousQueue` 同步队列： `CachedThreadPool` 使用，任务到来时必须复用或新建线程，线程过多可能OOM
- `DelayedQueue` 延迟队列： `ScheduledThreadPool` 使用，按照delay时间给任务排序出队；延迟队列满时自动50%扩容、永不超出上限，因此`ThreadPool`只能保持在`corePoolSize`，不会新建核心线程以外的线程

`ArrayBlockingQueue` 有界； `LinkedBlockingQueue/Deque` 可以指定有界、默认MAX\_VALUE(qi)；其余均无界

## 线程池中线程异常后，销毁新建 or 复用

- 用 `execute()` 提交的任务：未捕获异常导致线程终止，线程池创建新线程替代
- 用 `submit()` 提交的任务：异常会被封装在由 `submit()` 返回的 `Future` 对象、`Future#get()` 才抛出，线程继续复用

## 针对CPU密集型/I/O密集型的线程池优化

- **CPU密集型任务**：线程池大小建议与 CPU核心数相同，避免过多线程带来上下文切换的开销。核心线程数=最大线程数=CPU核数
- **I/O密集型任务**：线程池可以配置更多的线程，因为线程在等待 I/O 时不占用 CPU 资源。核心线程数>=核数\*2

# Future

用于进行异步调用、获取异步结果

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning); // 取消任务
    boolean isCancelled(); // 判断任务是否被取消
    boolean isDone(); // 判断任务是否已经执行完成
    V get() // 获取任务执行结果
    V get(long timeout, TimeUnit unit) // timeout内没有返回结果就抛出
    TimeOutException异常
}
```

```

<T> Future<T> submit(Callable<T> task);
Future<?> submit(Runnable task); // Runnable在内部也转为Callable

// submit()传入Runnable或Callable的task，会返回Future的实现类FutureTask，管理任务运行/
获取结果。
Future<String> future = executor.submit(() -> {
    Thread.sleep(1000); // 模拟耗时任务
    return "Hello from Future!";
});
String result = future.get(); // 阻塞直到任务完成并获取结果

```

- **只读**: Future 表示任务的结果，但不能主动设置结果。它只能通过任务的完成来产生结果。
- **get() 阻塞**: 如果结果尚未准备好，get() 方法会阻塞，直到任务完成并返回结果
- **非强制回调**: 不能自动通知任务何时完成，必须显式调用 get() 来检查任务是否完成，这意味着通常需要阻塞或轮询

## CompletableFuture

- `complete(T value) / completeExceptionally(Throwable ex)`: **主动设置成功的/异常的结果**
- `.thenApply(result -> {...})`: **自动回调、链式调用**

```

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
    return 10;
}).thenApply(result -> {
    return result * 2; // 对上一个任务的结果进行处理
}).thenApply(result -> {
    return result + 5; // 再次对结果处理
});

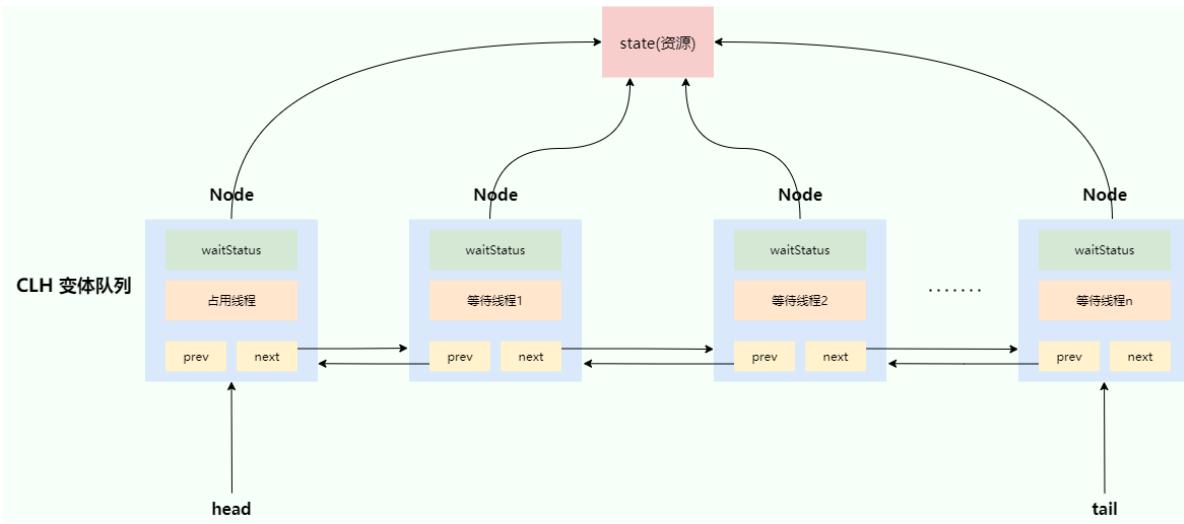
```

## AQS

抽象类 AbstractQueuedSynchronizer 用于构建锁和同步器。 ReentrantLock , Semaphore , SynchronousQueue 都是基于AQS

### AQS原理

- 若有被请求的资源处于空闲，则分配给请求的线程，将线程设为工作状态、资源设为锁定状态
- 若被请求的资源无空闲，则把线程阻塞加到CLH等待队列里，等待时机唤醒分配
- CLH队列锁：虚拟Node构成的双向队列



```

public abstract class AbstractQueuedSynchronizer extends
AbstractOwnableSynchronizer
    implements java.io.Serializable {
    // 共享变量，使用volatile修饰保证线程可见性
    private volatile int state;

    // 读写state值，final修饰不能被子类重写
    protected final int getState() {
        return state;
    }

    protected final void setState(int newState) {
        state = newState;
    }

    // CAS原子操作，state的更新都是用CAS完成的
    protected final boolean compareAndSetState(int expect, int update) {
        return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
    }
}

```

- `ReentrantLock` 初始设 `state=0` 表示未锁定；占有/重入锁 `state+1`，释放锁 `state-1`；  
`state!=0` 时其他线程不能占用锁
- `CountDownLatch` 任务分为N个子线程去执行，初始设置 `state=N`，执行完一个就CAS执行 `state-1`

## IO

- `InputStream / Reader`：所有的输入流的基类，前者是字节输入流，后者是字符输入流。
- `OutputStream / Writer`：所有输出流的基类，前者是字节输出流，后者是字符输出流。

# 字节流

## InputStream字节输入流

- `java.io.InputStream` 抽象类

- `read()` : 返回输入流中下一个字节的数据, 返回0-255, 未读取到则返回 -1
- `read(byte[] b)` : 读取一些字节到数组 `b` 中, 等同于 `read(b, 0, b.length)`
- `read(byte[] b, int off, int len)` : 偏移量+读取的最大字节数
- `skip(long n)` : 忽略输入流中的n个字节, 返回实际skip的数量
- `available()` : 返回输入流中可以读取的字节数。
- `close()` : 关闭输入流释放相关的系统资源。
- `readAllBytes()` : 读取输入流中的所有字节, 返回字节数组

- `FileInputStream`

- `try-with-resource` 直接使用:

```
try (InputStream fis = new FileInputStream("input.txt")) {
    while ((content = fis.read()) != -1) {
        System.out.print((char) content);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

- 通常配合 `BufferedInputStream` (字节缓冲输入流) 来使用:

```
// 新建一个 BufferedInputStream 对象
BufferedInputStream bufferedInputStream = new BufferedInputStream(new
FileInputStream("input.txt"));
// 读取文件的内容并复制到 String 对象中
String result = new String(bufferedInputStream.readAllBytes());
System.out.println(result);
```

- `DataInputStream`

读取指定类型数据, 不能单独使用

```
FileInputStream fileInputStream = new FileInputStream("input.txt");
// 必须将 fileInputStream 等其他输入流作为构造参数才能使用
DataInputStream dataInputStream = new DataInputStream(fileInputStream);
// 可以读取任意具体的类型数据
dataInputStream.readBoolean();
dataInputStream.readInt();
dataInputStream.readUTF();
```

- `ObjectInputStream`

从输入流中读取并转换成Java对象 (反序列化), `ObjectOutputStream` 用于序列化

```
ObjectInputStream input = new ObjectInputStream(new
FileInputStream("object.data"));
MyClass object = (MyClass) input.readObject();
input.close();
```

**用于序列化和反序列化的类必须实现 `Serializable` 接口，对象中如果有属性不想被序列化，使用 `transient` 修饰**

## OutputStream字节输出流

- `java.io.OutputStream` 抽象类

- `write(int b)`
- `write(byte[] b); write(byte[] b, int off, int len)`
- `flush()`：刷新此输出流，并强制写出所有缓冲的输出字节
- `close()`

- `FileOutputStream`

- `try-with-resource` 直接使用

```
try (FileOutputStream output = new FileOutputStream("output.txt")) {  
    byte[] array = "JavaGuide".getBytes();  
    output.write(array);  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

- 配合 `BufferedOutputStream` (字节缓冲输出流)

```
FileOutputStream fileOutputStream = new FileOutputStream("output.txt");  
BufferedOutputStream bos = new BufferedOutputStream(fileOutputStream)
```

- `DataOutputStream`

- `ObjectOutputStream` 将对象写入到输出流，序列化

## 字符流

区分字符流和字节流：操作字节流时不知道编码类型容易乱码；但字符流是JVM将字节转化而成较为费时。因此媒体文件用字节流，涉及到字符就用字符流。

字符流默认 `unicode`，也可以用构造方法传参自定义编码

## Reader字符输入流

- `java.io.Reader` 抽象类

- `read()`：从输入流读取一个字符。
- `read(char[] cbuf)`：从输入流读取一些字符存到字符数组 `cbuf`，等价于 `read(cbuf, 0, cbuf.length)`
- `read(char[] cbuf, int off, int len)`，增加了off和len
- `skip(long n)`
- `close()`

- `InputStreamReader` 和 `FileReader`

`InputStreamReader` 是字节流转换为字符流的桥梁；其子类 `FileReader` 是基于该基础上的封装，可以直接操作字符文件。

```
// 字节流转换为字符流的桥梁  
public class InputStreamReader extends Reader {}  
// 用于读取字符文件  
public class FileReader extends InputStreamReader {}
```

```
try (FileReader fileReader = new FileReader("input.txt");) {  
    while ((content = fileReader.read()) != -1) {  
        System.out.print((char) content);  
    }  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

## Write字符输出流

- `java.io.Writer` 抽象类
  - `write(int c)`
  - `write(char[] cbuf); write(char[] cbuf, int off, int len)`
  - `append(CharSequence csq)`: 将指定字符序列 `csq` 附加到该 `Writer` 对象，返回该 `Writer`
  - `append(char c)`: 将指定字符附加到该 `Writer` 对象
  - `flush()`: 刷新此输出流，并强制写出所有缓冲的输出字符。
  - `close()`
- `OutputStreamWriter` 和 `FileWriter`

```
// 字符流转换为字节流  
public class OutputStreamWriter extends Writer {}  
// 用于写入字符到文件  
public class FileWriter extends OutputStreamWriter {}
```

```
try (Writer output = new FileWriter("output.txt")) {  
    output.write("你好，我是Guide。");  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

## 字节缓冲流

IO非常消耗性能，缓冲流将数据加载至缓冲区，一次性读取/写入多个字节，从而避免频繁的 IO 操作，提高流的传输效率。

- 字节流和字节缓冲流的性能差别主要体现在 `write(int b)` 和 `read()` 这两个一次只读取一个字节的方法的时候。
- `read(byte[] b, int off, int len)` 和 `write(byte b[], int off, int len)` 差别不大

### 实现方法：

采用装饰器模式来增强 `InputStream` 和 `OutputStream` 子类对象的功能：将其作为构造方法的参数传入，来构造对应的缓冲流对象。

例如：通过 `BufferedInputStream` 字节缓冲输入流，来增强 `FileInputStream` 的功能：

```
BufferedInputStream bufferedInputStream = new BufferedInputStream(new FileInputStream("input.txt"));
```

原理：

`BufferedInputStream` 字节缓冲输入流：

- 从底层输入流（文件）读取数据到内存的过程不是一个字节一个字节读取，而是先一次性读到内部的缓冲数组，之后再根据所调用的 `read` 方法从缓冲区读到程序中。
- 默认大小 8192B，可通过 `BufferedInputStream(InputStream in, int size)` 指定缓冲区大小

`BufferedOutputStream` 缓冲输出流：同理先在内部缓冲数组中写入字节，之后当缓冲区满或 `flush()` 时才一并写到目的输出流（文件）去。

## 字符缓冲流

同理存在 `BufferedReader`（字符缓冲输入流）和 `BufferedWriter`（字符缓冲输出流）

## 打印流

`System.out` 实际上获取到了一个 `PrintStream` 对象，`System.out.print()` 实际调用了 `PrintStream#write()` 方法。

```
// PrintStream 字节打印流，是 OutputStream 子类
public class PrintStream extends FilterOutputStream implements Appendable,
Closeable {}

// PrintWriter 字符打印流，是 OutputStream 子类
public class PrintWriter extends Writer {}
```

## RandomAccessFile

可读可写、通过文件指针实现对内容的随机访问（跳转任意位置）

- 构造方法** `RandomAccessFile(String file, String mode)`
  - `file` 为文件路径
  - `mode` 操作模式
    - `r`：只读模式
    - `rw`：读写模式
    - `rws`：读写模式，且同步更新文件内容和文件元数据 (size date 等)
    - `rwd`：读写模式，且同步更新文件内容
- 常用方法**
  - `getFilePointer()`：返回当前文件指针的位置
  - `seek(long pos)`：将文件指针移动到指定的位置，可从该位置进行读写操作
  - `length()`：返回文件的长度

- write read...

## IO设计模式

### Decorator Pattern

- 装饰器增强了原有流的功能（如提供缓冲机制）
- 能够灵活地对任何字节流（众多子类）包装
- 符合开闭原则（Open-Closed对扩展开放、对修改关闭）

`FilterInputStream` 和 `FilterOutputStream` 是字节流IO的装饰器核心类。

`BufferedInputStream`、`DataInputStream` 都是 `FilterInputStream` 子类。

而对于 `FilterInputStream`，还可以继续用 `InflaterInputStream` 包装，由 `zipInputStream` 继承后实现压缩功能。**装饰器可以嵌套使用。**

### Adapter Pattern

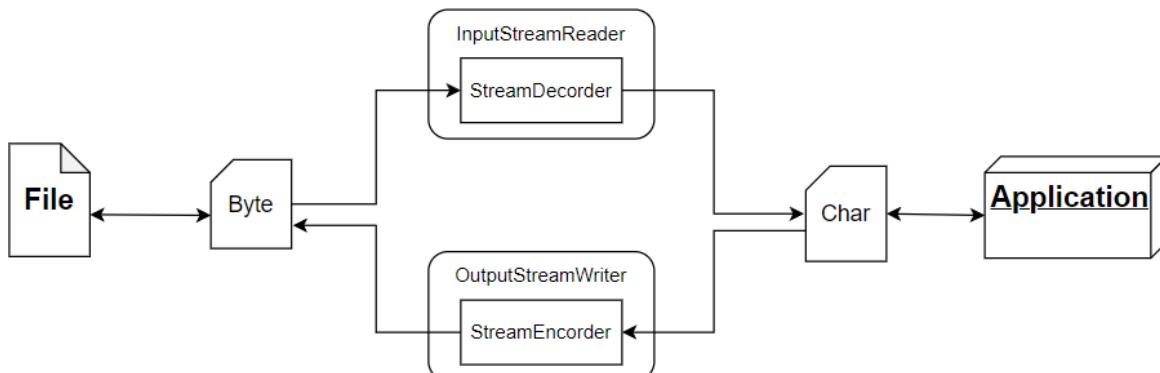
用于接口互不兼容的类的协调工作

被适配的类或对象成为Adaptee，作用于Adaptee的成为Adapter

- 类适配器使用继承实现
- 对象适配器使用组合实现

`InputStreamReader` 和 `OutputStreamWriter` 就是两个适配器Adapter，它们的适配者Adaptee就是 `InputStream` 和 `OutputStream` 的子类。

- `InputStreamReader` 继承 `Reader` 抽象类，使用 `StreamDecoder` 对字节进行解码，**实现字节流到字符流的转换**
- `OutputStreamWriter` 继承 `Writer` 抽象类，使用 `StreamEncoder` 对字符进行编码，**实现字符流到字节流的转换**



```

// InputStreamReader 是适配器, FileInputStream 是被适配的类
InputStreamReader isr = new InputStreamReader(new FileInputStream(fileName),
"UTF-8");
// BufferedReader 增强 InputStreamReader 的功能(装饰器模式)
BufferedReader bufferedReader = new BufferedReader(isr);
  
```

在FutureTask类中也使用了适配器，将 Runnable 适配成 callable

```
public FutureTask(Runnable runnable, V result) {  
    // 调用 Executors 类的 callable 方法  
    this.callable = Executors.callable(runnable, result);  
    this.state = NEW;  
}
```

## NIO

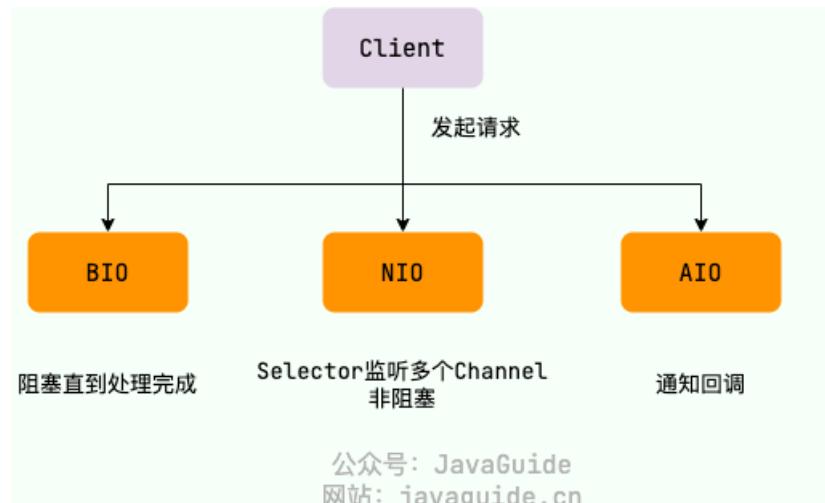
顾客来到餐厅点单：

- **BIO**: 每有一桌顾客落座，就要派一个服务员过去一直等着，直到那桌点好餐。
- **NIO**: 一个服务员在等待中同时关注着好几桌的客人，有哪桌喊他他就过去。
- **AIO**: 一个服务员并不一直等待，有人叫他他才过去，否则就先下楼。

早期Java的IO模型是BIO（阻塞IO，每次IO建立一个阻塞的线程等待IO操作完毕）。

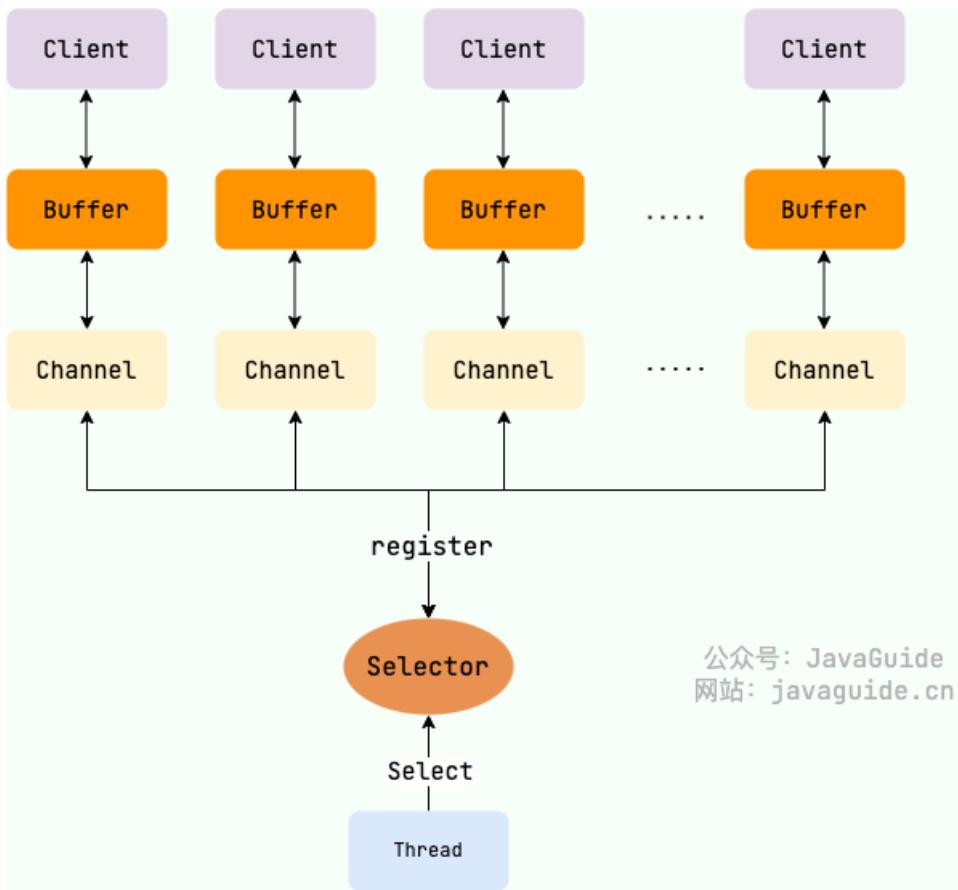
**NIO**（多路复用的非阻塞IO）是非阻塞、面向缓冲、基于通道的I/O，可以使用少量的线程来处理多个连接。

AIO（异步IO）使用异步回调，但用的不多。



## NIO实现原理

**NIO核心组件：**



- **Buffer缓冲区:** NIO 通过 Channel 来与外界数据进行连接，而读写数据都是通过缓冲区进行操作的。读操作的时候将 Channel 中的数据填充到 Buffer 中（再读进Java程序里），而写操作时将 Buffer 中的数据写入到 Channel 中（写到外部文件里）。

- Buffer使用静态方法进行实例化

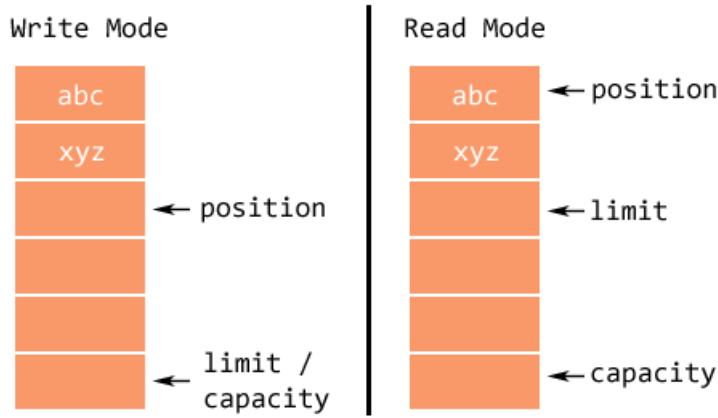
```
public static ByteBuffer allocate(int capacity); // 分配堆内存
public static ByteBuffer allocateDirect(int capacity); // 分配直接内存
```

- Buffer使用一个缓冲区数组同时完成读写操作，创建时默认写模式，通过 `clip()` 切换成读模式，`clear()` 切回写模式
- Buffer通过几个成员变量控制读写：

```
public abstract class Buffer {
    private int mark = -1; // 可选，将位置直接定位到该标记处
    private int position = 0; // 下一个可被读写的数据的位置(索引). 写模式切换到读模式clip时清零以从头读
    private int limit; // Buffer读写数据的边界. 写时为capacity, 读时为Buffer中实际数据大小
    private int capacity; // Buffer可存储最大容量, 创建后不可改变
}
```

上述成员变量满足:  $mark \leq position \leq limit \leq capacity$

读写切换前后的Buffer情况：



$0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity}$

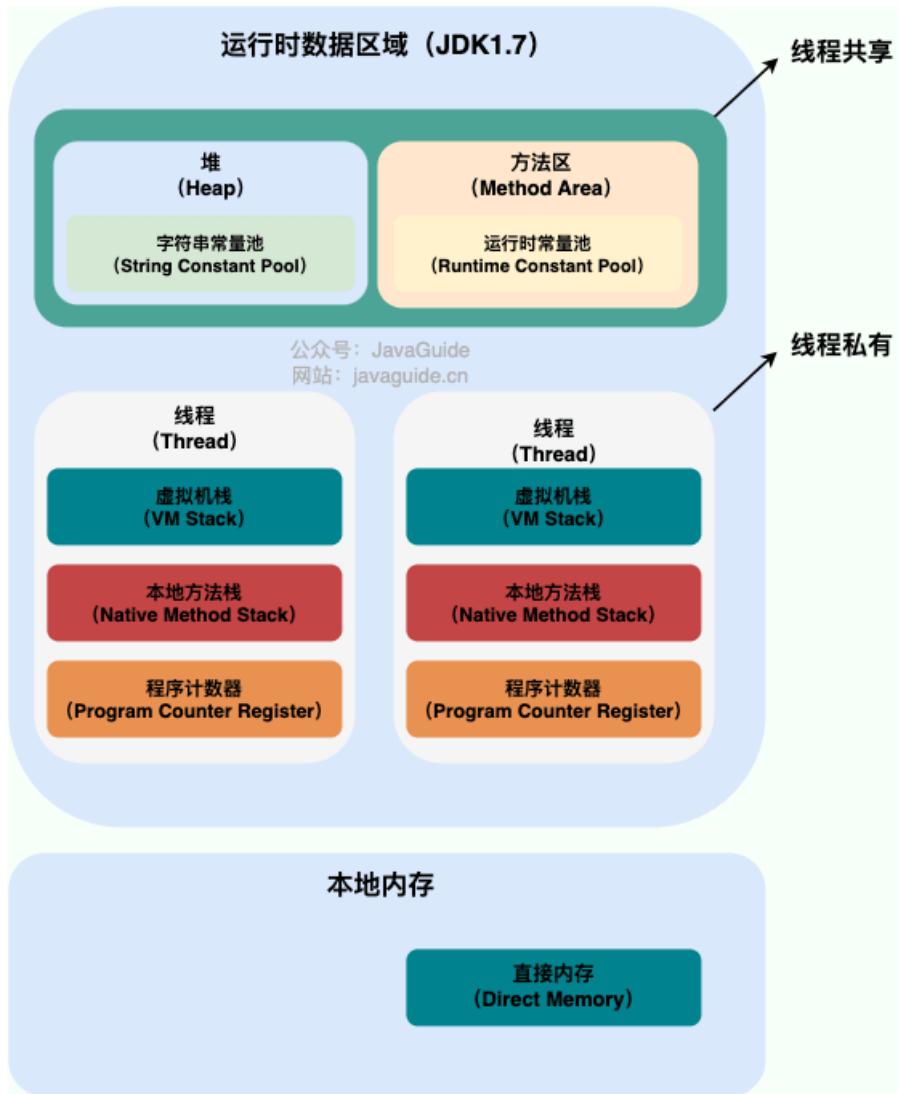
- **Channel通道:** Channel 是一个**双向的、可读可写的数据传输通道**, NIO 通过 Channel 来实现数据的输入输出。通道是一个抽象的概念, 它可以代表文件、套接字或者其他数据源之间的连接。
- **Selector选择器:** 允许一个线程处理多个 Channel, 基于事件驱动的 I/O 多路复用模型。所有的 Channel 都可以注册到 Selector 上, 由 Selector 来分配线程来处理事件。
  - 通过 Selector 注册通道的事件, Selector 会不断地轮询注册在其上的 Channel
  - 当事件发生时 (比如某Channel上有新的TCP连接接入、读写事件) 这个Channel就处于就绪状态, 被 Selector 轮询出来
  - Selector 会将相关的 Channel 加入到就绪集合中。通过 SelectionKey 可以获取就绪 Channel 的集合, 然后对这些就绪的 Channel 进行相应的 I/O 操作

## JVM

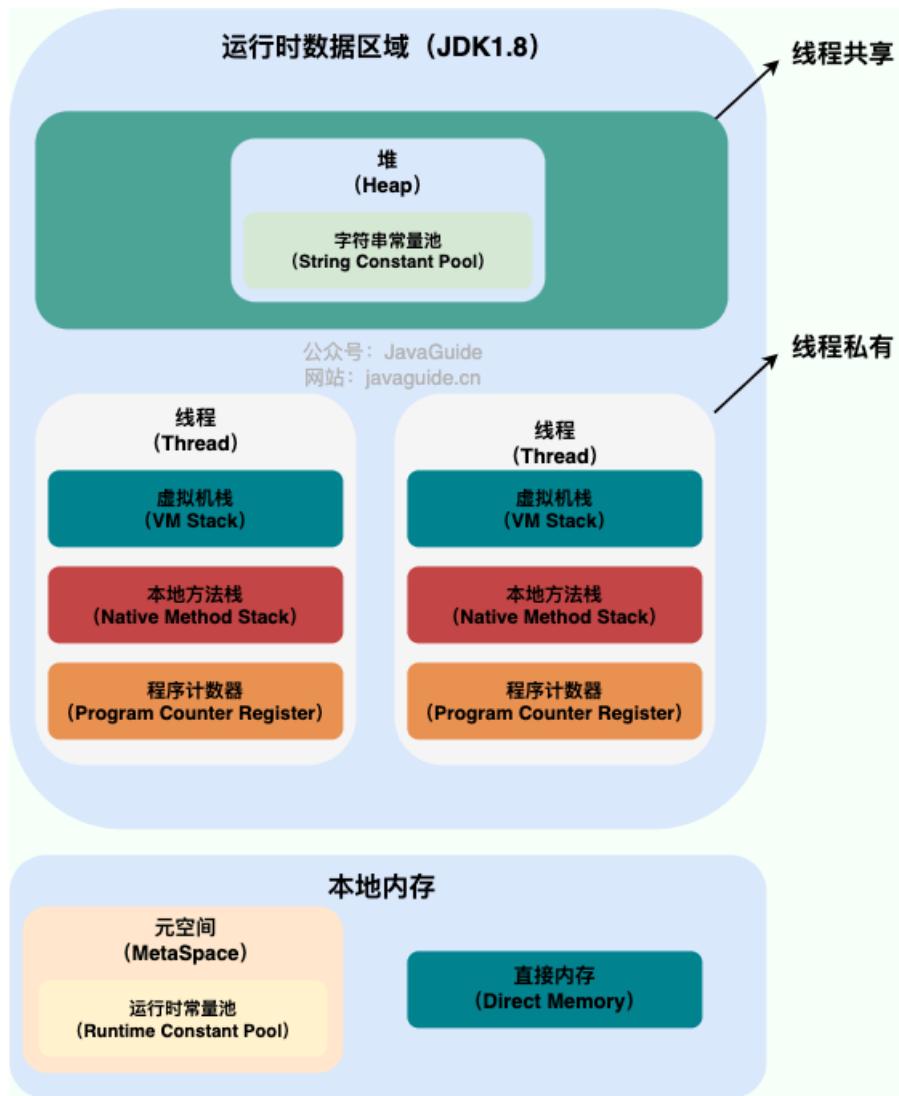
---

### 运行时内存区域

java7:



java8:



### 线程私有的：

随线程创建、随线程死亡

- 程序计数器
- 虚拟机栈
- 本地方法栈

### 线程共享的：

- 堆
- 方法区
- 直接内存 (非运行时数据区的一部分)

## 线程私有

### 程序计数器

- 程序计数器用于按序号读取指令，通过改变计数器**实现代码流程控制**：顺序执行、循环、分支、异常
- 多线程上下文切换时**记录当前线程运行位置**
- 唯一不会Out of Memory的区域

## JVM栈 (栈)

栈是JVM运行时内存的核心，除了native方法外，其他任何方法的调用都是使用栈。栈由栈帧组成，方法调用时对应的栈帧入栈，方法结束后栈帧出栈。



- 局部变量表：方法中声明的各种局部变量，基本数据类型、对象引用reference（对象存在堆里，栈中只有引用）
- 操作数栈：中间计算的临时值
- 动态链接：字节码中方法调用以**符号引用**形式出现，该符号存储在方法区的常量池中，当执行方法调用时，会到常量池中查找该符号引用对应的实际内存地址的直接引用，这个过程即动态链接。方法递归过多会导致栈帧过多，栈过深会StackOverflow异常（如果栈大小可以动态调整，则可能触发OOM异常）

## 本地方法栈

Java虚拟机栈为JVM执行Java方法（字节码），而本地方法栈则执行JVM用到的Native方法。HotSpot合二为一。

## 堆

JVM内存管理最大的一块，多线程共享，**唯一目的就是存放对象实例和数组**，几乎所有对象都存在这（随着JIT和逃逸分析的发展，仅存在于某方法中的对象可能直接放在栈上）

Java堆是垃圾回收的主要区域，也称为**GC堆**

## 字符串常量池

避免String对象重复创建影响性能。

HotSpot JVM中字符串常量池由 `StringTable` 实现，类似一个固定大小的哈希表（通过 `-XX:StringTableSize` 参数来设置大小），保存【字符串 - 字符串对象引用】的键值对映射。字符串对象的引用指向堆中的字符串对象，也就是说，**字符串常量池中存的是字符串对象的引用，实际的对象还在堆上。**

## 方法区

当JVM要使用一个类时，它需要读取并解析Class文件获取相关信息，再将信息存入到方法区。方法区会存储已被虚拟机加载的**类信息、字段信息、方法信息、常量、静态变量、即时编译器编译后的代码缓存等数据**。

方法区（抽象概念/规范）—— 永久代（具体实现）、元空间（具体实现）

- Java8之前，HotSpot用**PermGen永久代**实现方法区，存在**堆内存**中，受制于JVM内存，有固定上限

```
-XX:PermSize=N //方法区(PermGen)初始大小  
-XX:MaxPermSize=N //方法区(PermGen)最大大小,超过这个值将会抛出OutOfMemoryError,默认unlimited
```

- Java8开始，用**MetaSpace元空间**取代方法区，存在**本地内存**中，溢出风险小

## 常量池

运行时常量池存放编译期生成的各种**字面量 (Literal)** 和**符号引用 (Symbolic Reference)**

- 字面量：整数、浮点数和字符串字面量
- 符号引用：类符号引用、字段符号引用、方法符号引用、接口方法符号  
栈帧中的动态链接就会在这寻找方法符号引用

## 直接内存

**本地内存**中一块特殊的内存缓冲区，不在Java堆和方法区中。

JDK1.4的NIO引入了基于通道Channel与缓冲区Buffer的非阻塞IO方式，它使用Native库函数**给Buffer分配本地内存**。在Java堆上创建DirectByteBuffer对象作为Buffer的引用，通过DirectByteBuffer进行操作，避免Java堆和Native堆来回复制数据。

# 垃圾回收GC

Java垃圾回收（内存自动管理）主要管理堆中对象的分配与回收；有时也管理方法区的部分数据（类、常量池）

## 死亡对象判断方法

- **引用计数法**：给对象维护一个引用计数器，引用数为0则表示死亡；但无法解决循环引用问题
- **可达性分析（引用链）**：从一些称为**GC Roots**的对象出发，沿引用链图搜索可达节点；不可达为可回收对象（但不一定立刻回收）
  - **GC Roots**：栈帧中的局部变量对象，Native栈中引用的对象，方法区类静态属性引用对象、常量引用的对象，同步锁持有的对象，JNI(Java Native Interface)引用的对象

## 引用类型

### 强引用StrongReference

- 最普遍的引用类型，正常情况下**创建对象即为强引用**
- JVM不会回收强引用，即使Out of Memory

```
object obj = new Object(); // 强引用，GC不会回收该对象
```

### 软引用SoftReference

- 通过 `java.lang.ref.SoftReference` 类创建
- 软引用的关联对象可以一直存活，直到内存不足时才回收
- 常用于实现**内存敏感的缓存机制**，比如缓存对象不常用但创建代价大

```
SoftReference<Object> softRef = new SoftReference<>(new Object());  
// 软引用，内存紧张时可能被 GC 回收
```

### 弱引用WeakReference

- 通过 `java.lang.ref.WeakReference` 类创建
- 比软引用更弱，一旦GC发现则直接回收
- 常用于设计**弱引用哈希表** `weakHashMap`，key没有强引用则自动回收

### 虚引用PhantomReference

- 通过 `java.lang.ref.PhantomReference` 类创建，无法通过虚引用访问对象，唯一用途是追踪该对象的回收
- 一旦GC直接回收，回收时加入 `ReferenceQueue` 队列
- 通常结合 `ReferenceQueue` 使用，跟踪对象的回收状态，实现一些需要**在对象回收后进行清理**的场景

```
ReferenceQueue<Object> refQueue = new ReferenceQueue<>();  
PhantomReference<Object> phantomRef = new PhantomReference<>(new Object(),  
refQueue);  
// 虚引用，只能通过 ReferenceQueue 检测对象是否被回收
```

## GC算法

- 标记-清除 **Mark-and-Sweep**

标记出所有存活的对象，标记完成后统一回收未被标记的对象。最基础的GC算法，其他都以此为实现基础。

问题：效率不高；GC后产生大量不连续的**内存碎片**

- 复制 **Copying**

提前将内存分成相等两半，每次使用其中一半；每次GC将存活的对象复制到另一半上，则清理后原来占用的一半就是完整空白的

问题：可用内存缩小为一半；不适合老年代（需要复制的存活对象较多）

- 标记-整理 **Mark-and-Compact**

标记后不是直接清除，而是让所有存活的对象向一端移动，然后清理掉端边界以外的内存（分别合并了已用内存和剩余内存）

问题：效率不高

- 分代回收

根据对象的存活周期将对象分成新生代和老年代，分别划归到不同的内存区域中

- **新生代**：每次GC都有大量对象死去，存活对象少、有老年代进行空间担保，因此使用**复制算法**
- **老年代**：对象存活几率高、GC频率少，但没有额外空间进行担保，因此用标记清除或**标记整理**

## GC算法-分代回收

1. **新生代**内存(Young Generation): `Eden` (新生) 、 `s0` 和 `s1` (Survivor)

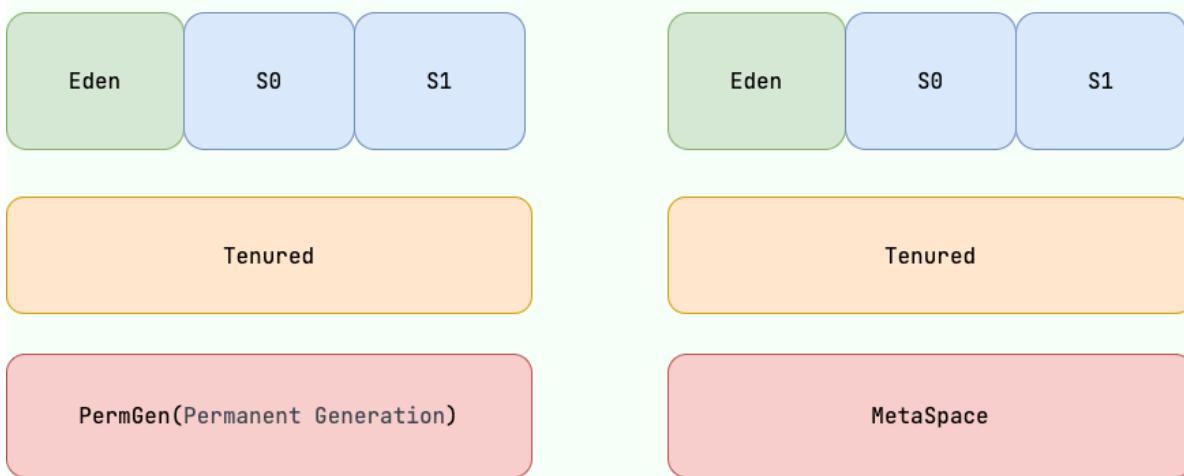
在GC中存放新创建的对象，主要用于**短生命周期**的对象

2. **老生代**(Old Generation): `Tenured`

在GC中存放**长期存活**的对象，从新生代晋升到老年代的对象会驻留在**这里**

3. **永久代**(Permanent Generation): `PermGen`

PermGen在JDK1.8后被MetaSpace取代，放在本地内存，不参与Java GC



## HotSpot GC类型

- Partial GC部分回收:
  - Minor GC / Young GC 新生代回收: 只对整个新生代 Eden, Survival(s0, s1) 垃圾收集, 开销小
  - Major GC / Old GC 老年代回收: 对整个老年代 Tenured 回收。需要注意的是MajorGC在有的语境中也用于指代FullGC
  - Mixed GC 混合回收: 对整个新生代和部分老年代回收
- Full GC 整堆回收: 收集整个Java堆和方法区

## 分配与回收流程

### 1. 对象优先在Eden分配

- 当Eden没有足够空间进行分配时, JVM将发起一次**Minor GC**, 之后能放进Eden还是放Eden
- 大对象(大量连续空间, 字符串、数组)直接进老年代, 减少新生代的回收频率

### 2. 长期存活的对象逐渐增加Age、进入老年代

- 对象在Eden经过一次MinorGC后被**复制进Survivor**, 年龄Age++变为1
- 在Survivor中每经过一次MinorGC则Age++, 达到年龄阈值(默认15)进入老年代
- **空间分配担保**: 在MinorGC之前, 确保老年代本身还有容纳新生代所有对象的剩余空间
  - 若`-xx:HandlePromotionFailure`设置允许担保失败, 则尝试一次有风险的MinorGC; 否则进行FullGC

### 3. 老年代内存不足, 进行Major GC 或 Full GC

可能导致较长的Stop-The-World, 暂停应用程序所有线程

### 4. 方法区回收

- 常量池回收: 常量不被引用时, 等待Full GC回收
- 类卸载:
  - 没有该类的实例
  - 该类的ClassLoader不被引用
  - 该类不被其他静态或常量引用

满足这三条, 则卸载该类、等待Full GC回收

## 垃圾收集器Collector

只要是GC就会**STW** (Stop-The-World) 停止所有用户线程。ParNew和Parallel是通过**GC时多线程**来加快GC、缩短STW；而CMS和G1能通过GC线程和用户线程**并发**，减少STW的影响。

Java8默认Parallel GC; Java9以上默认G1。

- **Serial收集器**: 单线程GC

Serial (新生代) 使用复制算法, Serial Old (老年代) 使用标记整理

- **ParNew收集器**: 多线程版本的Serial

- **Parallel收集器**: 多线程收集器, 关注吞吐量 (CPU利用率)

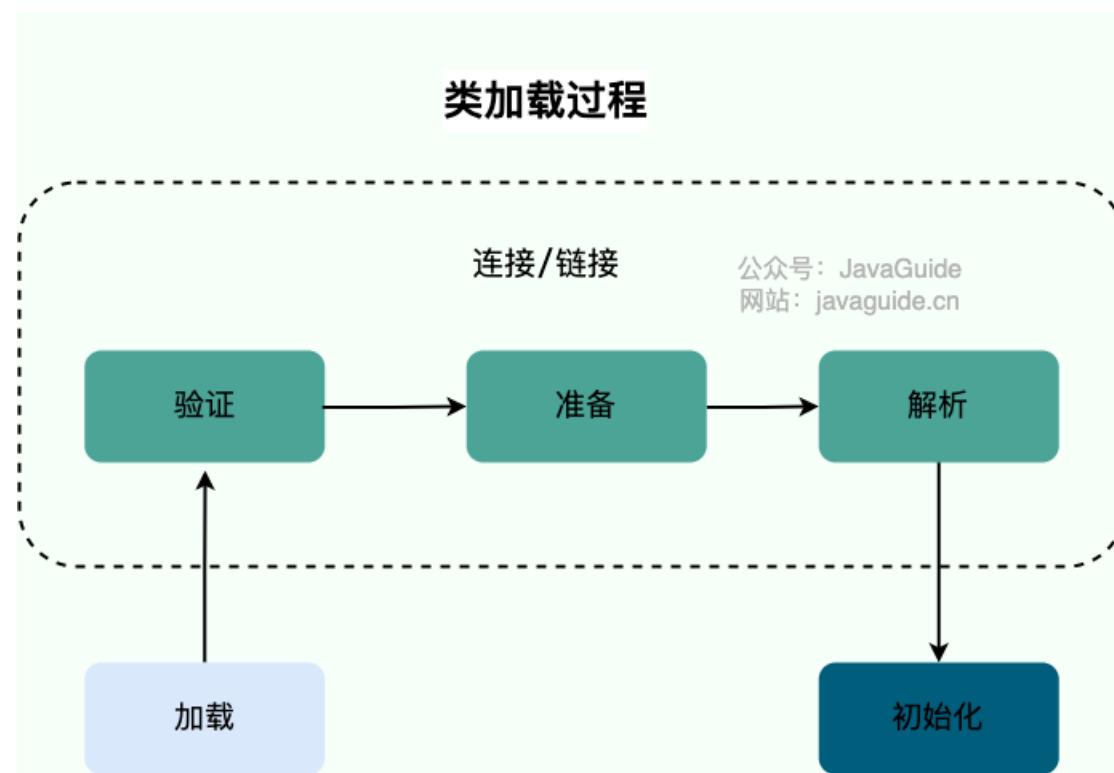
Parallel Scavenge (新生代) 使用复制, Parallel Old (老年代) 使用标记整理

- **CMS (Concurrent Mark-and-Sweep)** : 并发收集器, GC线程和用户线程并发执行, 关注用户体验 (最短STW)

- **G1收集器**: 并发, 面向多CPU, 高吞吐量、低STW

## 类加载器ClassLoader

类的加载过程。ClassLoader作用于“加载”步骤中。



## ClassLoader作用

- **加载类的字节码**: 将Java类的字节码 `.class` 文件, 加载到到JVM内存中, 生成一个代表该类的 `Class` 对象
- **动态加载类**: JVM按需动态加载类, 已加载的类存放在ClassLoader中, 只有没加载过的才会再用ClassLoader加载进来
- **支持自定义类加载器**: 通过扩展继承 `ClassLoader` 抽象类, 开发者可以对特定类的进行自定义加载

- 双亲委派模型：类加载器遵循一种层次化机制，使得核心类优先用顶层类加载器，保证系统稳定安全

## ClassLoader种类

- JVM内置

1. `BootstrapClassLoader` 启动类加载器

- 最顶层的加载类，是JVM的一部分，由C++实现，没有父级
- 主要用来加载JDK内部核心类库 %JAVA\_HOME%/lib，以及被 `-xbootclasspath` 参数指定的路径下的所有类
- `ClassLoader# getParent()` 中表示为 `null`，因为C++实现的在Java中没有对应

2. `ExtensionClassLoader` 扩展类加载器

- JVM外实现，继承自 `ClassLoader` 抽象类
- 主要负责加载 %JRE\_HOME%/lib/ext 目录下的 jar 包和类，以及被 `java.ext.dirs` 系统变量所指定的路径下的所有类

3. `AppClassLoader` 应用程序类加载器

- JVM外实现，继承自 `ClassLoader` 抽象类
- 面向我们用户的加载器，负责加载当前应用 classpath 下的所有 jar 包和类

- 自定义

通过继承 `ClassLoader` 抽象类，实现自定义的类加载器

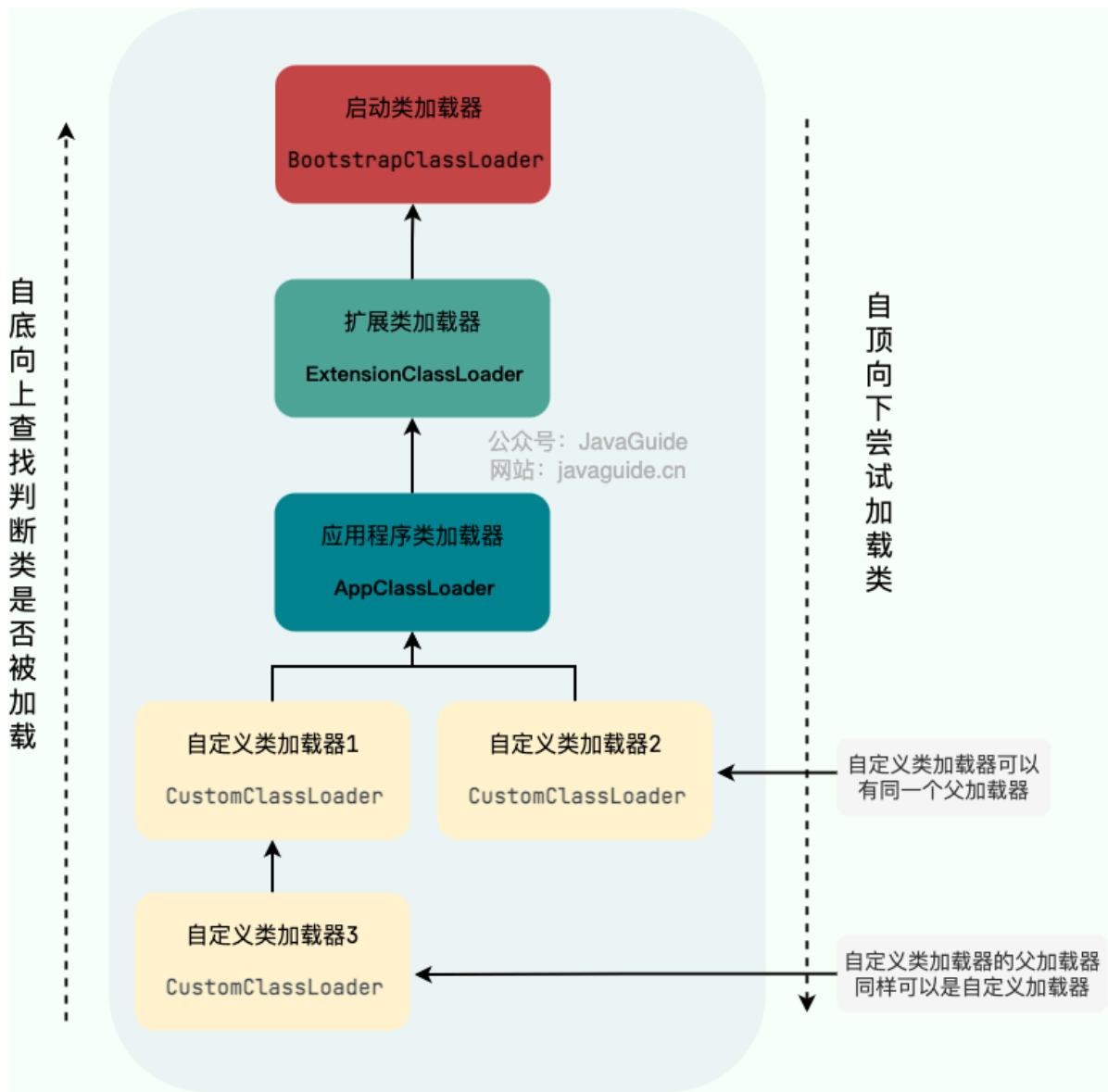
### `ClassLoader# getParent()`

每个类都可以通过 `class# getClassLoader()`，获取到它的类加载器

每个类加载器都可以通过 `ClassLoader# getParent()`，获取它的父类加载器

- 自己编写的 Java 类，它的类加载器是 `AppClassLoader`
- `AppClassLoader` 的父类加载器是 `ExtensionClassLoader`
- `ExtClassLoader` 的父类加载器是 `BootstrapClassLoader`，因此输出结果为 `null`

## 双亲委派模型



## 流程

- 判断类是否已加载**: 自底向上找目标类是否已被加载过（子类->父类一层层找），若被加载则直接返回
- 尝试委派父类加载**: 类加载器首先不会自己去尝试加载目标类，而是委派给父类加载器去完成（调用父加载器 `loadClass()` 来加载类）。因此所有的加载请求最终都会传送到顶层的 `BootstrapClassLoader`。
- 父类无法加载则自己加载**: 只有当父加载器反馈无法完成这个加载请求（它的父类和它自己的范围中没有目标类）时，子加载器才会尝试自己去加载（调用自己的 `findClass()` 方法来加载类）
- 自顶向下都无法加载这个类，那么它会抛出异常 `ClassNotFoundException`

## 好处

- **保证系统稳定**

核心类库都会被先委派给更顶层的类加载器，核心API不会被篡改

- **避免类重复加载**

**JVM判断类是否相同，结合类名+类加载器判断。**同一个类名，被父类加载器和子类加载器分别加载后也变成了两个类。若不进行双亲委派，则可能在判断类是否已加载时出错，导致类分别被父子重复加载了。

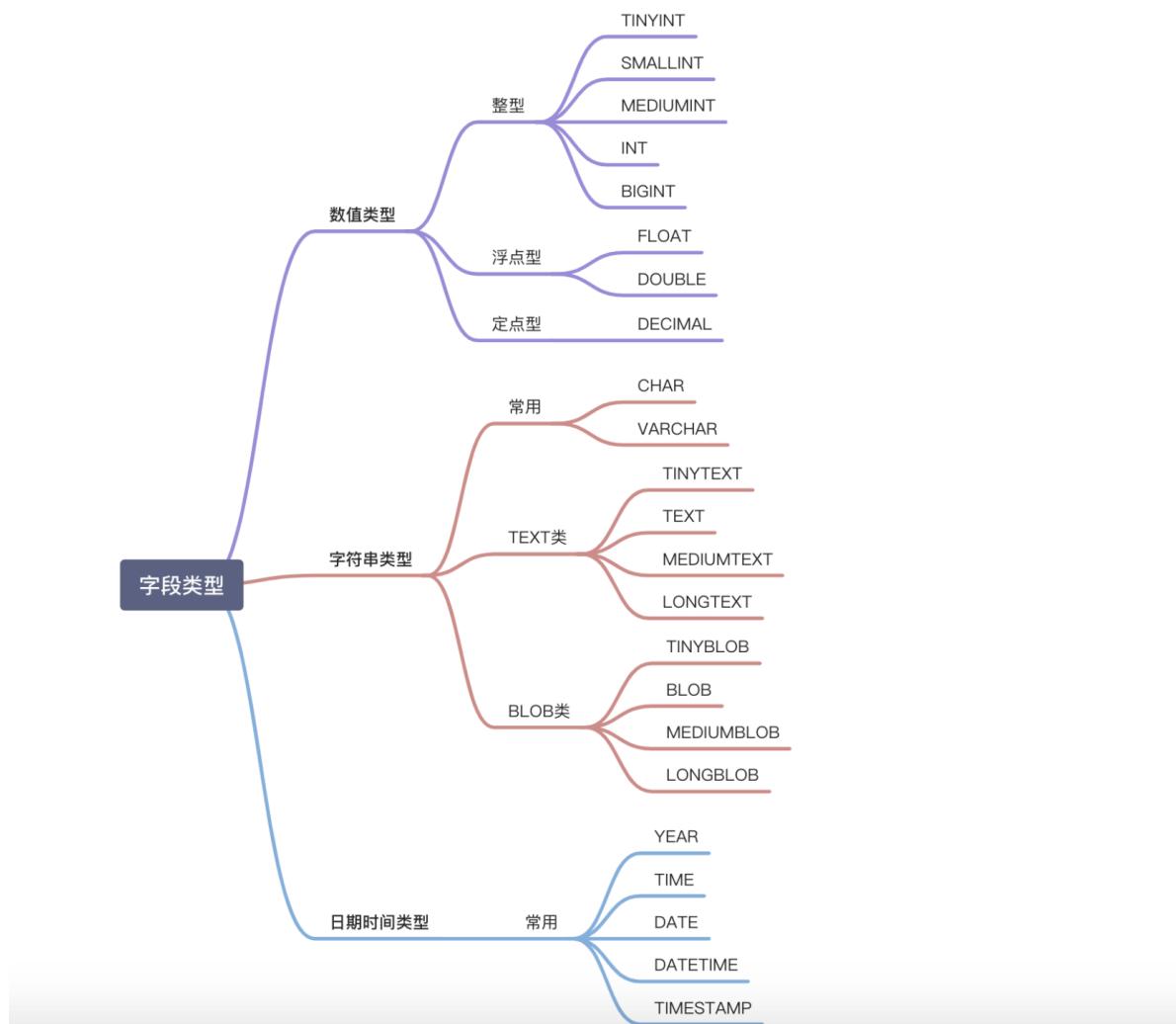
# RDBMS - MySQL

关系型数据库，MySQL为例

## 数据库范式

- **第一范式1NF**: 属性不可再分
- **第二范式2NF**: 消除了非主属性对于主键的部分函数依赖 (非主属性必须对主键是完全函数依赖)  
主键 {学号, 身份证号} -> 姓名, 但是 学号->姓名 就已足够, 姓名 对于主键是部分函数依赖, 要把主键拆开成俩表
- **第三范式3NF**: 消除了非主属性对于主键的传递函数依赖 (非主属性必须直接依赖于主键)  
学号->系号, 系号->系主任, 那么 系主任 对 学号 就是传递函数依赖

## MySQL字段



- **CHAR和VARCHAR**: 定长字符串和变长字符串

- CHAR 在存储时会在右边填充空格以达到指定的长度（检索时会去掉空格，因此存入的字符串最好别以空格结尾）
- VARCHAR不会填充，而是额外花一两个字节保存长度信息，能够避免上述问题
- 存入超过长度的字符串时，**严格模式下会拒绝存入并报错，非严格模式会截断存入并警告**
- **DECIMAL和FLOAT/DDOUBLE**: 定点数和浮点数。Decimal存储精确小数值，对应  
`java.math.BigDecimal`
- **TEXT/BLOG**: 大文本数据；二进制数据（媒体文件）
  - **不推荐用**，无默认值、索引效率低、IO开销高
  - 换VARCHAR，或外部存储服务代替（COS、ES）
- **DATETIME和TIMESTAMP**: SQL提供的时间类型，此外还有一种做法是直接存**数值型时间戳**

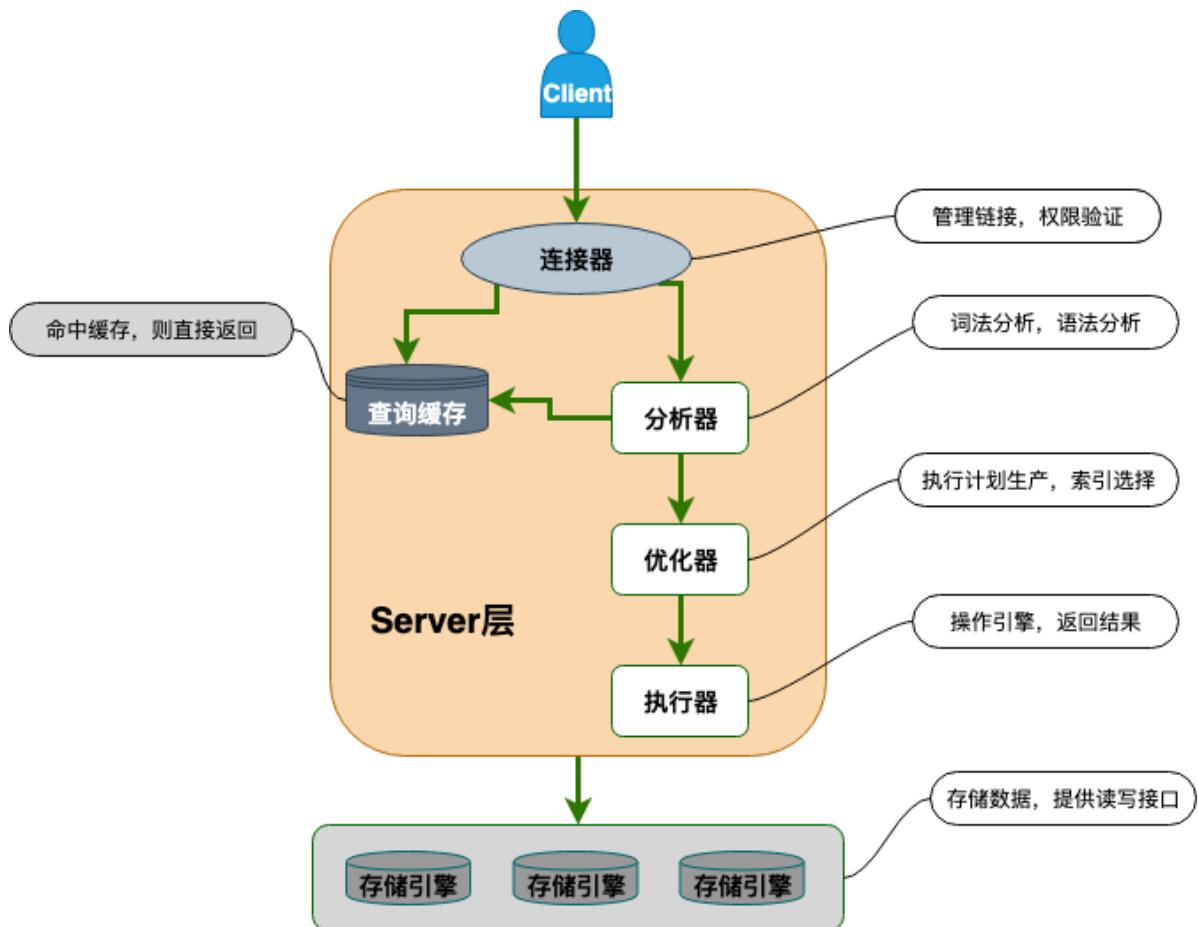
类型	存储空间	格式	范围	时区信息
DATETIME	5-8B	YYYY-MM-DD hh:mm:ss[.fraction]	1000-01-01 00:00:00[.000000] ~ 9999-12-31 23:59:59[.999999]	无
TIMESTAMP	4-7B	YYYY-MM-DD hh:mm:ss[.fraction]	1970-01-01 00:00:01[.000000] ~ 2038-01-19 03:14:07[.999999]	有
<b>数值型时间戳(INT)</b>	4B	纯数字如 1578707612	1970-01-01 00:00:01之后	无

通常选择数值型时间戳，或者TIMESTAMP

- **NULL**: 空值。**不建议用NULL作默认值！**
  - NULL在SQL中是占用存储空间的，而 '' 空字符串本身不占空间（VARCHAR会根据长度占空间）
  - NULL只能用 `IS NULL`, `IS NOT NULL` 判断，'' 空串可以使用比较运算符
  - `SELECT NULL=NULL` 为 `false`，但是 `DISTINCT, GROUP BY, ORDER BY` 时认为相等
  - `COUNT(列名)` 和聚合函数会忽略NULL，但是 `COUNT(*)` 包含NULL
- **不存在BOOLEAN**，实际上用 `TINYINT(1)` 表示布尔值

## SQL语句在MySQL的执行过程

### MySQL运行逻辑架构



- **Server层**

所有跨存储引擎的功能(Procedure, Trigger, view, function)、通用的日志模块 binlog。

查询缓存在MySQL8.0被移除，因为不实用。

- **存储引擎**

默认InnoDB，只有InnoDB支持事务 Transaction (基于内部的 redo log)；MySQL5.5之前是 MyISAM，不支持行级锁（并发性能低）、不支持事务、不支持崩溃恢复。

引擎层是插件式的，是基于表的，每个表可以用不同的存储引擎。

## SQL语句执行

需要执行的SQL语句分为查询和更新（增删改），有不同的执行流程

- **查询语句**

```
select * from tb_student A where A.age='18' and A.name=' 张三 ';
```

1. 先检查是否有权限，无权限则返回错误；若在MySQL8.0之前还会检查查询缓存
2. 分析器进行词法分析、语法分析，判断语法正确性、提取关键词
3. 优化器确定执行方案（比如先查age还是name？）
4. 执行器执行上述方案，调用存储引擎API查询数据

- **更新语句**

```
update tb_student A set A.age='19' where A.name=' 张三 ';
```

1. 先查到该条数据，走一遍查询流程、但是不走缓存，更新语句会使缓存失效
2. 调用存储引擎API更改数据，InnoDB引擎进行更改、同时开始双日志同步、二阶段提交：

- InnoDB更改数据后redo log进入prepare预提交状态，告诉执行器执行完成
  - 执行器记录binlog，告诉InnoDB
  - 提交redo log为commit提交状态
3. 更新完成

## SQL语法执行顺序

1. 加载 **from**关键词后面跟的表，计算笛卡尔积，生成虚拟表vt1。这也是sql执行的第一步：表示要从数据库中执行哪些表。
2. 筛选关联表中满足**on**表达式的数据，保留主表数据，并生成虚拟表vt2。join表示要关联的表，on代表连接条件。
3. 如果使用的是**外连接**，执行on的时候，会将主表中不符合on条件的数据也加载进来，作为外部行。
4. 如果from子句中涉及多张表，则重复第一步到第三步，直至所有的表都加载完毕，更新vt3。
5. 执行**where**表达式，筛选出符合条件的数据生成vt4。
6. 执行 **group by** 子句进行分组。分组会把子句组合成唯一值并且每个唯一值只包含一行，生成vt5。一旦执行group by，后面的所有步骤只能操作vt5中的列（group by的子句包含的列）和聚合函数。

下一步开始才可以使用select中的别名，因为group by之后才能确定操作的列。
7. 执行**聚合函数**，例如sum、avg等，生成vt6。
8. 执行**having**表达式，筛选vt6中的数据。having是唯一一个可以在分组后执行的条件筛选表达式，生成vt7。
9. 执行**SELECT**，从vt7中筛选列，生成vt8。
10. 执行**distinct**，对vt8去重，生成vt9。其实执行过group by后就没必要再去执行distinct，因为分组后，每组只会有一条数据，并且每条数据都不相同。
11. 按照**order\_by\_condition**对vt9进行排序，此处亦可以使用别名。这个过程比较耗费资源。
12. 执行 **limit** 语句，取出指定条数的结果集返回给客户端。

## MySQL索引

在设计数据库索引时，要优先考虑如何最大限度地减少磁盘IO次数

### 索引划分

- 按照数据结构划分
- 按照物理存储方式划分：聚集和非聚集
- 按照应用维度划分：
  - **主键索引**：加速查询 + 列值唯一（不能有NULL）+ 表中只有一个
  - **二级索引/辅助索引/非主键索引**：存储的是主键的值，查找需要回表。普通索引、唯一索引、前缀索引都属于二级索引
  - **普通索引**：仅加速查询
  - **唯一索引**：加速查询 + 列值唯一（可以有NULL）
  - **覆盖索引**：一个索引包含（覆盖）所有需要查询的字段的值
  - **联合索引**：多列值组成一个索引，专门用于组合搜索，其效率大于索引合并

- **前缀索引**: 只适用于字符串类型，对文本的前几个字符创建索引，相比普通索引建立的数据更小，因为只取前几个字符。
- **全文索引**: 对文本的内容进行分词，进行搜索。目前只有 `CHAR, VARCHAR, TEXT` 列上可以创建全文索引。一般不会使用，效率较低，通常使用搜索引擎如ElasticSearch代替

## 常见索引数据结构

- Hash表：查询需要计算hashcode，不支持顺序和范围查询
- BST二叉查找树：不平衡时最差可能退化为O(N)
- AVL树：一种自平衡BST，平衡性较高但插入删除时维护代价大
- 红黑树：一种自平衡BST，插入删除O(1)，平衡性相对较弱、在磁盘中搜索性能一般（一个节点就一个键值对，树高比B树高、存储密度低）；但在内存中性能优异，因此TreeSet TreeMap HashMap都用了
- B树 自平衡多路查找树：也叫B-树，**键值对分布在树的所有节点中，每个节点可以存放多个键值对**。查找时从根节点开始，通过对关键字(key)二分，确定查找路径依次向下查找
- B+树：B树变体，**所有键值对存储在叶子节点**，非叶子节点仅作索引；**叶子节点之间形成链表**（方便顺序和范围查找）

InnoDB和MyISAM的索引结构都是B+树。

## 聚集索引和非聚集索引

- **聚集索引Clustered Index**
  - 物理存储顺序和按照索引顺序一致
  - 每个表只能有一个聚集索引（物理顺序是定的），该列称为聚集键
  - 索引本身就是数据，索引的data（叶子节点）直接存储了表中的数据行
- 更新成本较高，可能要重新排列数据
- **非聚集索引Non-Clustered Index**
  - 索引和数据分离，索引的data存的是数据的地址；可以建立多个非聚集索引
- 查找效率低

MyISAM使用非聚集索引；InnoDB使用聚集索引、使用主键作为聚集键，其他**辅助索引**的data存的都是主键（辅助索引会先得到主键，再走一遍主索引，称为**回表**）

## 覆盖索引

索引的字段（key或data）覆盖了所有需要查询的字段，**无需回表**二次查询

常见于根据字段查询主键；存在a字段索引时查询a

# 联合索引

也称**组合索引、复合索引**, 使用多个字段创建索引

```
# 对cus_order表建立联合索引(score, name)
ALTER TABLE `cus_order` ADD INDEX id_score_name(score, name);
```

**最左匹配**: 使用联合索引时, 按照索引字段顺序从左往右匹配WHERE条件

**联合索引原理**: 联合索引建立的B+树, 是按照**字段从左往右顺序进行排序的**, 因此靠右的字段是全局无序、局部有序(当左边一个字段确定时, 当前字段缩小到的范围就是有序排列的, 就可以继续使用索引), 因此是最左匹配、逐渐深入来进行查找

以联合索引(a,b,c)为例

- `where a = 1`, 完全使用索引
- `where a = 1 and b = 2`, 完全使用索引
- `where b = 2 and a = 1`, 完全使用索引 (优化器优化, 交换条件字段顺序)
- `where a = 1 and c = 3`, 仅使用索引定位到 `a=1`, 之后沿着记录所在链表遍历查找
- `where b = 2`, 未命中索引

范围查询

- `<、>`: 停止匹配

联合索引(a,b), 查找 `where a > 1 and b = 1`

联合索引仅定位到 `a>1` 即停止, 因为之后 `b` 不是有序排列的, 因此从 `b` 开始沿链表遍历查找

- `<=、>=、between`: 部分继续匹配

`where a >= 1 and b = 1`

分成 `a=1` 和 `a>1` 两步:

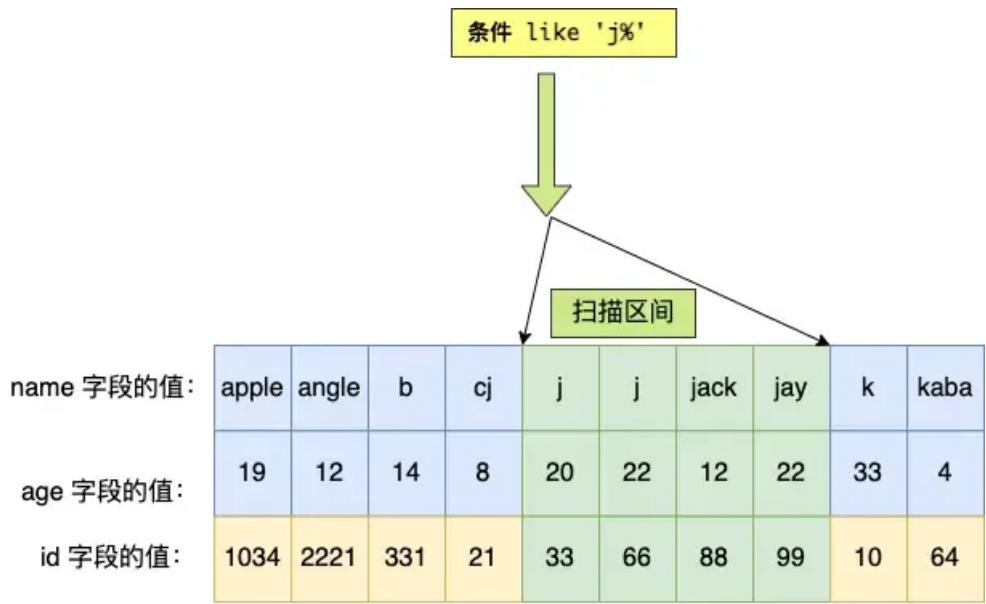
- 对于 `a=1` 的情况, 在这个子区间内 `b` 是有序的, 可以继续向后匹配 `b`
- 对于 `a>1`, 匹配到此停止

`where a between 2 and 8 and b = 2`

相当于 `a=2`、`a=8`、`2<a<8` 三步, 同理。

- `like`前缀查询: 部分继续匹配

`where name like 'j%' and age = 22`



联合索引 index\_name\_age 示意图  
(先按照 name 排序, name 相同的情况下, 再按照 age 排序)

联合索引匹配到 name 以 j 开头的扫描区间，这个区间内 age 是无序的，因此只能遍历扫描；但是在 name=j 的子区间内，age 是有序的，所以在此时依然会用联合索引取到 name='j' and age=22 的定位，从这里开始扫描。

## 索引下推

MySQL 5.6 引入，减少回表，使得联合索引在查询时得到最大程度的利用

例子：主键 id，联合索引 (a,b)，查询 where a = 1 and b > 2

- 没有索引下推：先用二级索引回表查到完整数据再进行剩下筛选。在联合索引树中先查到 a=1，将符合该条件的记录全都进行回表查询，然后再过滤 b>2 得到结果
- 索引下推：在联合索引树中查找同时满足 a=1 and b>2 的记录，再一次性回表得到结果

## 正确建立索引

- 适合建立索引的字段：
  - 频繁被查询、作为 WHERE 条件
  - 常需要 ORDER BY, GROUP BY, DISTINCT：索引已经是有序的
  - 用于表连接 JOIN

不适合的：

- 频繁更新：维护索引成本不低
- NULL：NULL 在查询时难以被数据库优化；可以用 0, false 替代
- 避免重复和冗余：尽量给上述符合条件的字段建立联合索引，而非单列索引。单表别超过 5 个，索引过多增加 MySQL 优化器生成执行计划的负担
- 索引顺序：区分度最高、长度小、使用频繁的，放在最左侧

## 索引失效场景

- 组合索引不遵从最左匹配
- 在索引列上使用函数、计算、类型转换: `where length(s) = 6`, 索引保存的是索引字段的原始值
- like通配符在左: `where s like '%li'`
- 隐式转换:  
WHERE条件字段为字符串、要匹配数值时,会对字符串进行隐式转换,可能进行截取: `'100'`、`'0100'`、`'100a'`都可以转换为数值`100`,因此无法确定其在索引中的位置,索引失效
- 走索引不如全表扫描快:
  - or有一边非索引
  - in取值范围较大: `where b in (1,2,3,4,5)`
  - order by: 虽然索引是排好序的,但是要回表

## MySQL执行计划EXPLAIN

支持SELECT、DELETE、INSERT、REPLACE、UPDATE。常用于分析SELECT:

```
EXPLAIN SELECT ... FROM ... WHERE ...
```

信息	Result	1	剖析	状态							
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	student	(Null)	ref	联合索引	联合索引	62	const	1	100 (Null)	

@稀土掘金技术社区

- **id**

`SELECT`序列标识符,用于标识每个`SELECT`语句的执行顺序。

`id`越大优先级越高,相同则从上往下执行

- **select**

查询类型

- **SIMPLE**: 简单查询,不包含`UNION`或者子查询。
- **PRIMARY**: 查询中如果包含子查询或其他部分,外层的`SELECT`将被标记为`PRIMARY`。
- **SUBQUERY**: 子查询中的第一个`SELECT`。

```
SELECT * FROM tb_student WHERE age = (SELECT MAX(age) FROM tb_student);
```

外层和内层分别为`PRIMARY`和`SUBQUERY`

- **UNION**: 在`UNION`语句中,`UNION`之后出现的`SELECT`。
- **DERIVED**: 在`FROM`中出现的子查询将被标记为`DERIVED`。
- **UNION RESULT**: `UNION`查询的结果。

- **table**

查询用到的表的表名，以及过程中的临时表：

- <unionM,N> : 使用了 id 为 M 和 N 的表的 UNION 结果
- <derivedN> : 使用了 id 为 N 的表所产生的的派生表结果，可能产生自 FROM 中的子查询
- <subqueryN> : 使用了 id 为 N 的表所产生的的子查询结果

- **type**

查询执行的类型，执行的策略从优到劣排序：

system > const > eq\_ref > ref > fulltext > ref\_or\_null > index\_merge > unique\_subquery > index\_subquery > range > index > ALL

- **system**: 表中只有一行记录，是 *const* 在一种特例
- **const**: 查询条件可以精确地定位到唯一一条记录，用于主键或唯一索引
- **eq\_ref**: 当连表查询时，前一张表的行在当前这张表中只有一行与之对应。是除了 system 与 const 之外最好的 join 方式，用于使用主键或唯一索引作为连表条件。
- **ref**: 普通索引查询，查询结果可能找到多个符合条件的行。
- **index\_merge**: 查询条件使用了多个索引，开启Index Merge优化，此时执行计划中的key列出了所用的索引
- **range**: 对索引列进行范围查询，执行计划中的key列出了所用的索引
- **index**: 查询遍历了整棵索引树，与 ALL 类似，只不过扫描的是索引，而索引一般在内存中，速度更快
- **ALL**: 全表扫描

- **possible\_keys**

所有可能用到的索引。若为NULL则表示没有可用索引，有可能出现了索引失效

- **key**

经过优化器评估后，实际所用的索引。若为NULL则表示没用索引，可能索引失效

- **key\_len**

实际使用索引时，使用的索引长度(Byte)；在联合索引中可能是多个字段的长度和

```
#联合索引(age, name), age为int, name为varchar(10)
EXPLAIN SELECT * FROM tb_student WHERE age = 18 AND name = '张三';
#key_len = 14, 因为int 4B + varchar(10) 10B = 14B
```

- **rows**

预计要读取的行数

- **filtered**

按查询条件过滤后，留存的记录数的百分比

- **Extra**

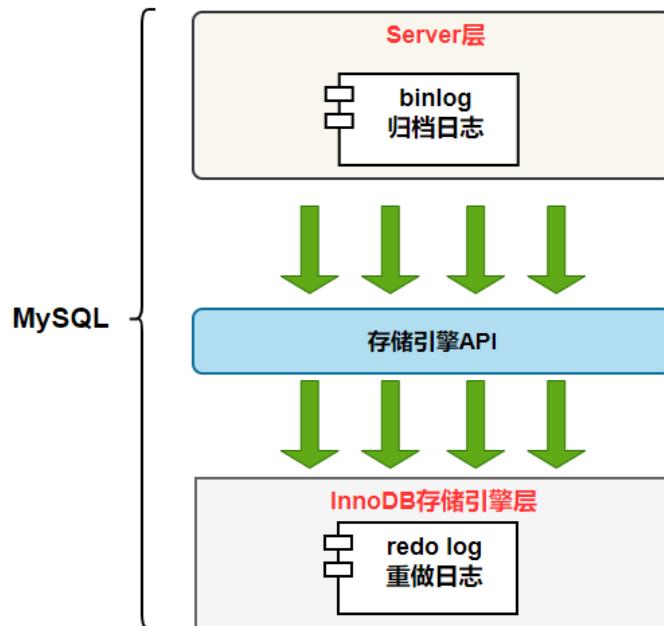
额外信息

- **Using filesort**: 在排序时使用了外部的索引排序，没有用到表内索引进行排序。可能有性能问题
- **Using temporary**: 需要创建临时表来存储查询的结果，常见于ORDER BY和GROUP BY。可能有性能问题
- **Using index**: 表明查询使用了覆盖索引，不用回表，查询效率非常高
- **Using index condition**: 表示查询优化器选择使用了索引条件下推这个特性
- **Using where**: 表明查询使用了 WHERE 子句进行条件过滤。一般在没有使用到索引的时候会出现

- Using join buffer (Block Nested Loop): 连表查询的方式，表示当被驱动表的没有使用索引的时候，MySQL 会先将驱动表读出来放到 join buffer 中，再遍历被驱动表与驱动表进行查询

## MySQL日志

binlog、redo log、undo log



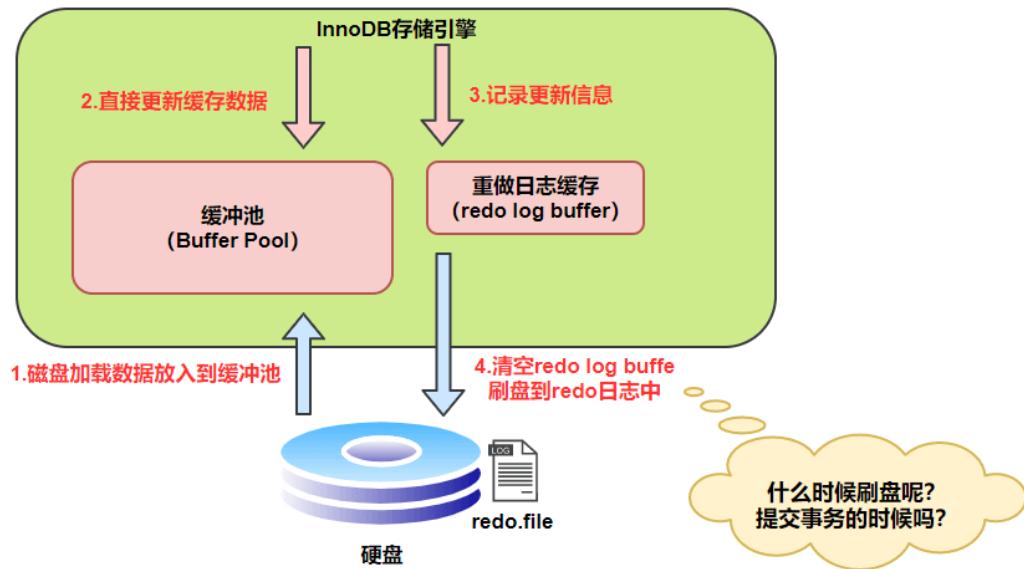
## redo log

InnoDB独有，物理日志（记录数据页修改），用于**崩溃恢复**。

**redo log意味着事务与磁盘同步**，有redo log记录就会保证数据已经持久化到磁盘，即使崩溃后也会重做

### log buffer缓存

- 数据 - Buffer Pool: MySQL数据以页为单位，一次读一页出来，放在缓冲池Buffer Pool中。查询和更新都先查Buffer Pool没有再磁盘IO加载
- redo log - redo log buffer**: redo log在内存中也存在一个buffer，每次有更新记录就先放到buffer里，等待时机进行**redo log刷盘**（先写入mysql外部的文件系统缓存page cache中，再刷入磁盘中的redo log文件里）



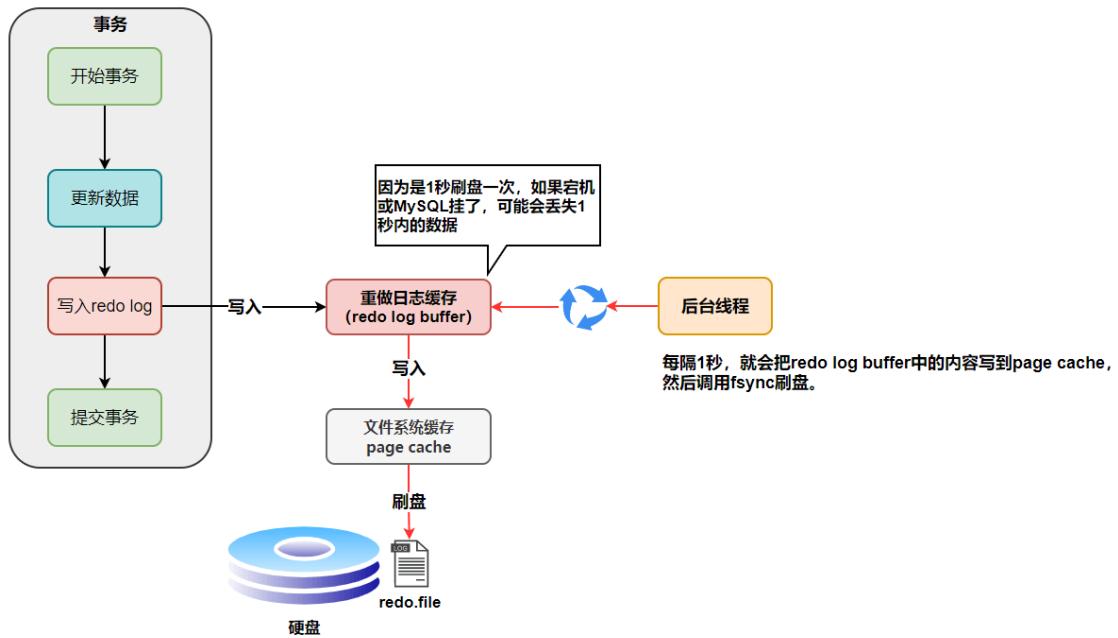
## redo log刷盘

为什么不直接将数据页刷盘？因为一页数据16kb，一次事务可能只修改挤b，刷盘效率低，因此用 redo log记录修改。数据页的刷盘有另外的策略

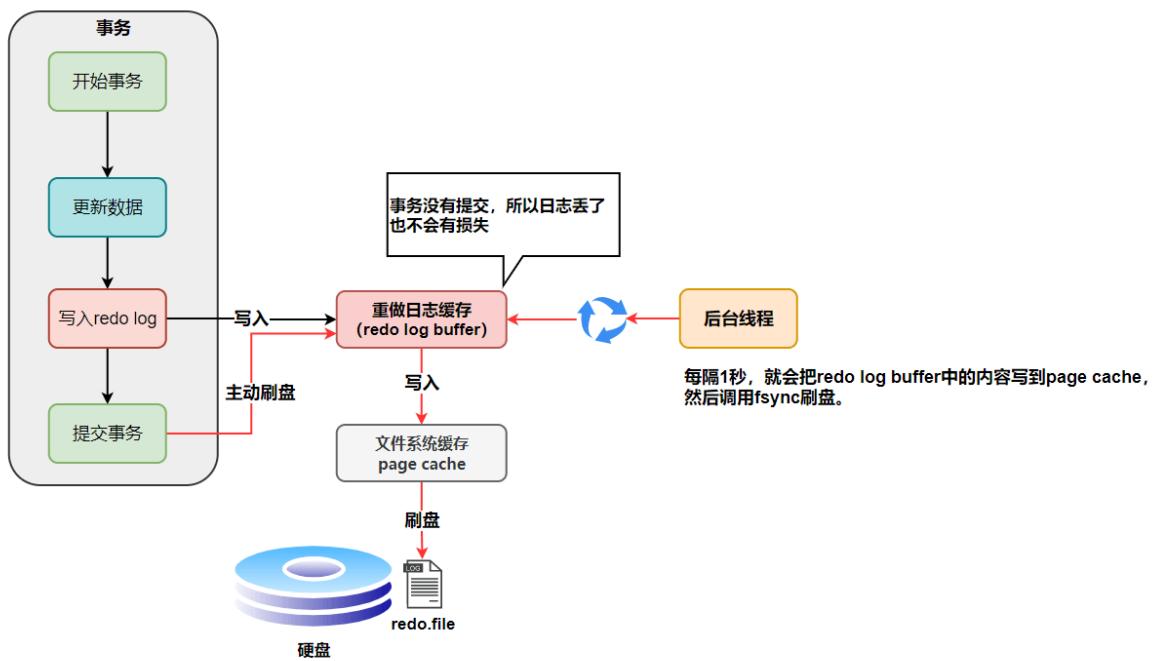
- **事务提交**: 事务提交后从log buffer刷盘redo log, 由 `innodb_flush_log_at_trx_commit` 参数设置刷盘策略 (默认 1) :
  - 0 : 事务提交不会刷盘, 交由其他方式 (比如后台刷新线程) 刷盘, 性能最高、最不安全
  - 1 : 默认。每次事务提交都刷盘, 性能最低、最安全
  - 2 : 每次事务提交只是把log buffer里的redo log文件写入page cache文件系统缓存
- **后台刷新线程**: InnoDB后台有个线程, 每隔1秒就将脏数据 (已修改但未写入磁盘的数据) 和相关的redo log buffer一起刷盘
- **log buffer空间不足**
- **Checkpoint检查点**: InnoDB定期设定Checkpoint, 将脏数据和相关的redo log一起刷盘
- **正常关闭服务器**: MySQL正常关闭时刷盘

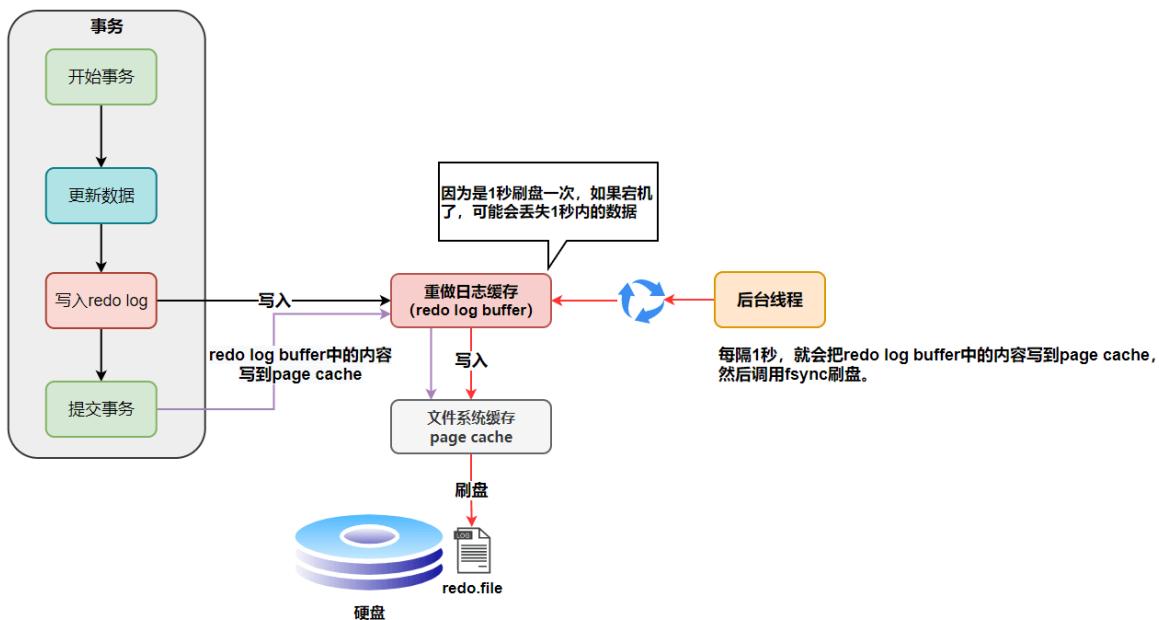
`innodb_flush_log_at_trx_commit` 参数控制的三种刷盘策略示意图：

innodb\_flush\_log\_at\_trx\_commit = 0



innodb\_flush\_log\_at\_trx\_commit = 1



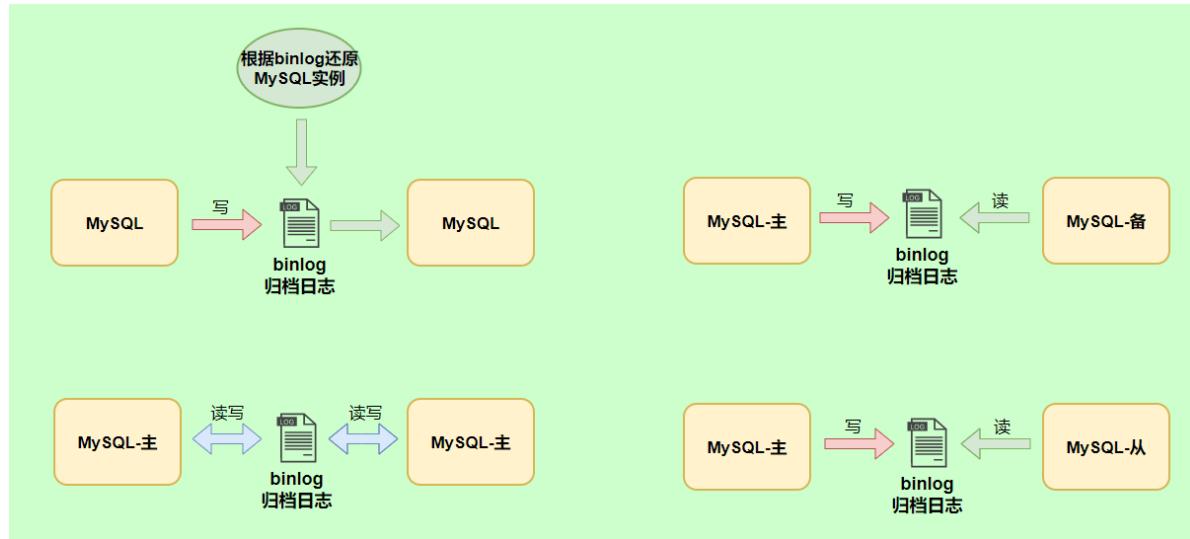


这种情况下，事务提交后就会把redo log写入mysql外部的page cache里，因此mysql挂了不受影响，但假如系统宕机了就会损失1s的数据

## binlog

MySQL server层的通用日志，逻辑日志（记录SQL语句），不管用什么存储引擎都会在数据更新时产生binlog。

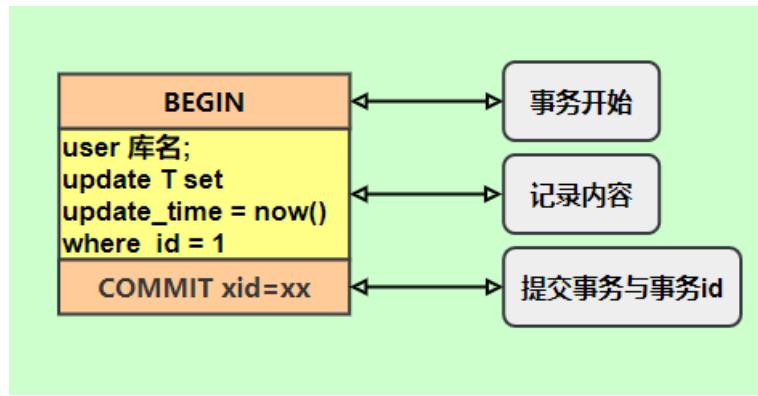
**用于数据备份、主备、主主、主从，实现MySQL集群架构的数据一致性**



### binlog记录格式

有三种，通过 `binlog_format` 指定：

- `statement`：记录SQL语句原文



用binlog恢复数据时会执行SQL语句，直接执行 now() 就会获取当前系统时间、造成数据不一致

- `row`：记录填充了具体数据的SQL语句

恢复数据时通过mysqlbinlog工具解析出来再执行，能够解决数据不一致；但是这种格式占用较多存储空间、更消耗性能

- `mixed`：折中，只有造成数据不一致的SQL语句被换成 `row`，其他保留 `statement` 原样

## binlog写入

- **binlog缓存**：因为事务binlog必须要一次性写入，mysql会给每个线程分配一个binlog cache。事务执行过程中，持续把日志写到binlog cache中。事务提交时，再把binlog cache写入磁盘（这个过程同样也是先写进文件系统缓存page cache中，再刷入磁盘）
- 通过 `binlog_cache_size` 参数控制单个线程binlog cache大小，超过该容量就要暂存到磁盘
- 通过 `sync_binlog` 控制写入磁盘的策略（默认1）：每次事务提交都会将日志从binlog cache里写到page cache中，而累积 `sync_binlog` 个事务后再从page cache写入磁盘。
  - 如果是0，则只写到page cache，系统自行判断什么时候写入磁盘
  - 默认是1，则每次事务提交都写盘
  - 如果是n，则累积n个事务再写盘

除了默认为1的情况，其他都有可能因为系统宕机而丢失数据。

## 两阶段提交

事务执行到底，开始将日志提交，使用二阶段提交：

**redo log prepare ---> binlog写入 ---> redo log commit**

- 若不设置二阶段提交：
  - 崩溃时交了redo log未交binlog：redo log将数据恢复到磁盘，但是binlog没有，导致备份和主从复制丢失该数据，数据丢失
  - 崩溃时交了binlog未交redo log：有事务的逻辑记录，但是物理层面没有，数据不一致
- 设置了二阶段提交：
  - redo log已commit，则直接提交
  - redo log只是prepare，若binlog已写入则提交redo log，否则回滚事务

只有redo log专门设计了prepare状态（已写入但未提交），binlog一旦写入就相当于提交，因此只能是redo log来做两阶段。

## undo log

InnoDB独有，逻辑日志（记录SQL语句），用于**事务回滚Rollback和多版本并发控制MVCC**

根据实际执行的SQL语句记录对应的undo语句：

1. **Insert undo log**: 记录新插入的数据、对应的DELETE语句。若事务提交则直接清理掉
2. **Update undo log**: 记录数据更新前后的值、对应的UPDATE语句。若事务提交，则会加入history list保留一段时间

### undo log与MVCC

在MVCC中undo log作用是保存并提供旧版本的数据

多个事务同时执行时，会根据该事务的时间戳，提供正确版本的数据，**确保每个事务在整个执行过程中都能够读到一致的数据（即事务刚开始时的数据快照）**

# MySQL事务

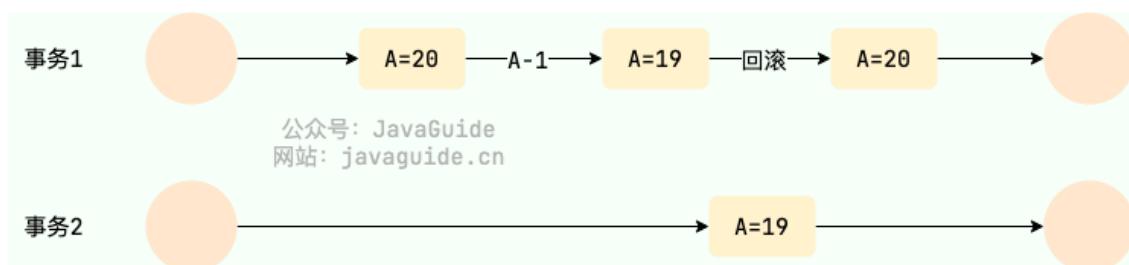
## ACID特性

1. **Atomicity原子性**: 最小执行单位，不能分割，要么都执行完，要么不执行。由undo log回滚来保证。
2. **Consistency一致性**: 事务执行前后，数据库状态一致（数据变动正确）。由AID共同确保。
3. **Isolation隔离性**: 多个事务并发执行时，事务的操作相互之间隔离，在未提交之前对其他事务不可见。由MVCC或锁机制实现。
4. **Durability持久性**: 事务一旦提交，结果永久保存在数据库中、不会因意外丢失。由redo log实现。

## 事务并发问题

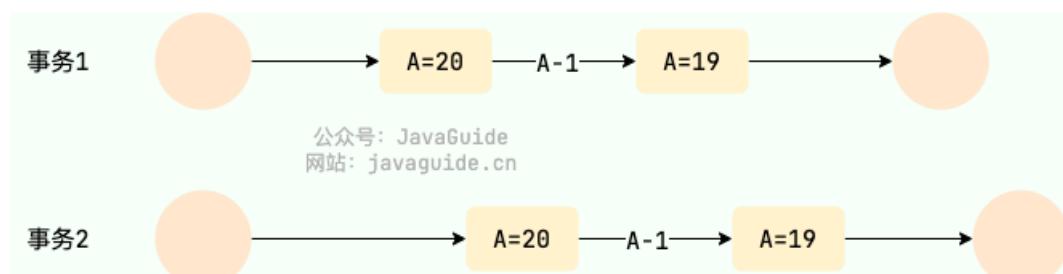
### • 脏读dirty read

事务1修改了数据（但未提交），被事务2读到，而事务1再回滚，那么事务2读到了脏数据



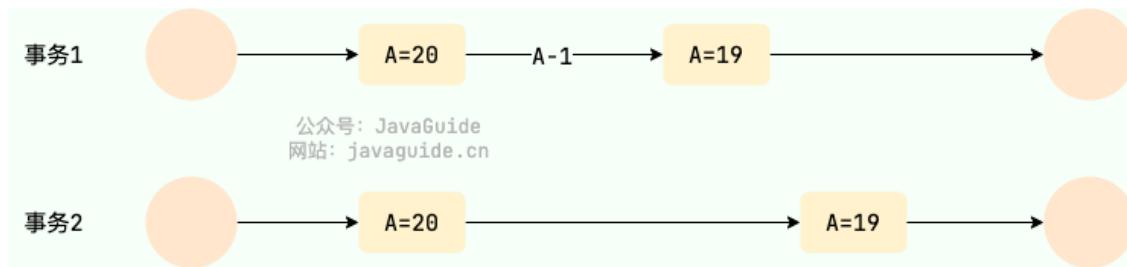
### • 丢失修改lost modify

事务1和事务2同时访问并修改数据，事务1先修改、事务2后修改，那么事务1的修改丢失



- 不可重复读unrepeatable read

事务2两次读取同一个数据，这中间事务1修改了该数据，那么事务2两次读的数据可能发生了变化



- 幻读phantom read

事务2两次读取某个范围的数据，这中间事务1插入或删除了数据，那么事务2第二次读取时会发现多了少了数据（好像幻觉）

幻读是不可重复读的一种特殊情况，幻读重点在于记录条数的变动，不可重复读重点在于记录的更改。两者应对方法不一样，update和delete直接加Record Lock即可，但是insert时没有记录无法加锁，只能加Gap Lock。

## 并发控制方式

### 1. 锁

相当于悲观模式的控制。MySQL主要通过读写锁（S锁-共享锁-读锁，X锁-排他锁-写锁）实现并发控制。

此外还有行级锁和表级锁，InnoDB能够支持行级锁。

### 2. MVCC多版本并发控制

乐观模式。

对一份数据会存储多个版本，通过事务的可见性来保证事务能看到自己应该看到的版本。通常会有一个全局的版本分配器来为每一行数据设置版本号，版本号是唯一的。

MVCC在MySQL中实现所依赖的手段主要是：隐藏字段、read view、undo log。

- undo log : undo log 用于记录某行数据的多个版本的数据。
- read view 和隐藏字段 : 用来判断当前版本数据的可见性。

## 事务隔离级别

- Read Uncommitted读取未提交

最低级别，能够读取并发事务未提交的数据变更，导致脏读幻读不可重复读

- Read Committed读取已提交

能读取并发事务已提交的数据变更，导致幻读不可重复读

通过MVCC实现

- Repeatable Read可重复读

事务中多次读取相同数据保持一致，即其他事务对所读数据的修改是不可见的；但是仍有可能幻读——其他事务插入了符合本事务查询条件的新数据，而可重复读只给已有数据上锁。

通过MVCC实现；InnoDB默认级别，InnoDB使用了MVCC+锁

- Serializable串行化：最高级别，事务串行执行，完全避免并发问题。代价过高

通过锁实现

需要注意的是，丢失修改（提交覆盖）本质上是业务逻辑层面的竞争问题，不是事务需要保证的。串行化或者乐观锁可以保证。

解决并发问题	Read Uncommitted	Read Committed	Repeatable Read	Serializable
脏读	✗	✓	✓	✓
不可重复读	✗	✗	✓	✓
幻读	✗	✗	✗	✓

InnoDB的可重复读隔离级别实际上解决了幻读问题！

- **一致性非锁定读**（不涉及修改、但要一致性）：快照读
- **锁定读/当前读**（修改数据）：**Next-Key Lock**（行级锁Record Lock锁住已存在数据行 + 间隙锁Gap Lock间隙锁阻止插入新行）

## MVCC多版本并发控制

通过在每个数据行上维护多个版本来实现的。当一个事务要对数据库中的数据进行修改时，MVCC会先为该事务创建一个数据快照，而不是直接修改实际的数据行。

### 1. 读操作 (SELECT)

- 事务在进行读取某数据行时，会选择其不晚于事务开始时间的最新版本，确保事务开始时其已经存在；
- 事务读取的是数据的快照，因此其他并发事务的对此数据的修改不会影响到当前事务的读取

### 2. 写操作 (UPDATE, INSERT, DELETE)

- 事务会为修改的数据行创建一个新的版本，并将新版本数据写入数据库
- 原始版本依旧保留，使得其他事务不受当前修改的影响

### 3. 事务提交和回滚

- 事务提交，它所作的修改成为数据库中的最新版本，对其他事务可见
- 事务回滚，它所作的修改被撤销，对其他事务不可见

## 一致性非锁定读和锁定读

- **一致性非锁定读**：读时不需要给数据加锁，加了X锁的数据也能直接读，读到的可能是旧版本数据
  - 通常使用版本号或时间戳，控制数据的可见性（一致性）
  - InnoDB中的MVCC使用**快照读**实现
  - 在 Repeatable Read 和 Read Committed 隔离级别下，执行普通SELECT时，InnoDB使用一致性非锁定读（但是RC时会通过Read View更新读的快照）
- **锁定读（当前读）**：读时需要给数据加锁，读取到的是最新的数据
  - `select ... lock in share mode`，加S锁
  - `select ... for update, insert, update, delete`，加X锁

InnoDB在Repeatable Read情况下执行当前读时，先加锁、然后读取最新的数据，因此**仍有可能发生不可重复读、幻读！**

## InnoDB实现MVCC

- 数据行的隐藏字段
  - `trx_id`: 最后一次插入或更新该行的事务ID。DELETE在数据库内部也是更新（逻辑删除）
  - `roll_ptr`: 回滚指针，指向该行的undo log
- 基于事务的一致性视图Read View: 在一个事务开始时创建，记录事务开始时系统中其他活跃事务（未提交事务）的情况，用于此事务之后查询数据时的版本可见性，保证一致性
  - `m_ids`: 活跃事务列表，记录创建Read View时系统中所有未提交事务的ID列表
  - `min_trx_id`: `m_ids` 中最小（最早开始）的事务ID，**小于 `min_trx_id` 的数据版本均可见**
  - `max_trx_id`: 最大事务ID+1，即下一个将分配的ID，**大于等于 `max_trx_id` 的数据版本均不可见**
  - `creator_trx_id`: 当前Read View事务ID

当事务读取数据行时，检查该数据行的 `trx_id` 与Read View比对：

1. 若小于 `min_trx_id`，则可见（比所有活跃事务都早，必然已提交）
2. 若大于 `max_trx_id`，则不可见（还没开始的事务，必然不可见）
3. 若介于二者之间：
  - `trx_id` 在 `m_ids` 中，不可见（说明是在当前事务运行过程中提交的）
  - 不在 `m_ids` 中，可见（说明它虽然比当前的一些活跃事务晚开始，但在当前事务开始时已经提交了）

总结：不在 `m_ids` 说明要么已完成要么未发生，已完成的那批就是可见的，其他不可见。

- 基于数据行的链式undo log

undo log的作用有两个，一个是回滚，另一个是MVCC读取旧版本记录。undo log通过数据行隐藏字段 `roll_ptr` 形成链表，链表头节点是最新版本的记录，链尾是最早记录

## RC和RR隔离等级下MVCC不同

在Read Committed和Repeatable Read隔离等级下，InnoDB都使用MVCC实现非锁定一致读，但是生成Read View的时机不同：

- RC: 在每次SELECT读取时都会生成一个新Read View（更新了 `m_ids` 列表，因此能读已提交）
- RR: 事务开始后只在第一个SELECT生成一个Read View，之后保持一致（静态，一定程度防止不可重复读和幻读）

## MVCC的局限性

MVCC提供了快照读机制，解决了大部分的可重复读和幻读问题，但是涉及到当前读时仍有问题：

事务A先快照读，事务B添加了一条该范围内的数据，事务A对该数据进行修改，事务A再次快照读则能读到该条数据

**幻读。**这是因为修改操作属于当前读，对最新数据生效，**事务A更新了该数据的版本号、能够被事务A快照读了**

事务A先快照读，事务B添加/删除了一条该范围内的数据，事务A再当前读 `select for update` 则条数不对

**幻读。**因为当前读是读最新的数据

事务A先快照读，事务B修改了数据，事务A再当前读 `select for update` 则读到修改数据  
**不可重复读。** 同样因为当前读是读最新的数据。这也说明，**MVCC并没有完全解决不可重复读！**

## 临键锁Next-key Lock

针对MVCC未能解决的问题，InnoDB使用Next-key Lock配合解决

- **Next-key Lock锁构成**

*Next-key Lock* 临键锁 = Record Lock 行级锁 + Gap Lock 间隙锁

- **Record Lock** 行级锁：锁住已存在的数据，目的是解决 不可重复读 以及 **删除数据造成的幻读**
- **Gap Lock** 间隙锁：基于索引顺序，锁住了查询覆盖范围，目的是解决 **添加数据造成的幻读**

- **Next-key Lock加锁时机**

当执行 当前读 时，InnoDB会对涉及的数据加上Next-key Lock，直到事务提交。

这同样存在问题，因为 **Next-key Lock在执行当前读时才加上**，在这之前其他事务的操作仍可能影响。

**因此最好在事务一开始，就使用select for update对其上锁！**

- **Next-key Lock其他的局限性**

只能防止select for update范围内的数据增删改，无法阻止范围外的操作

## MySQL锁

### 行级锁和表级锁

MyISAM仅支持表级锁；InnoDB都支持，默认行级锁

- 表级锁

- **全表扫描、DDL语句时加表级锁**
- 加锁开销小、并发效率低

`select for update / select in share mode` 在索引失效时给每一行加行锁，最终表现为表锁

- 行级锁

- InnoDB中**针对索引字段加的锁**
- **当前读操作时加行锁**
- 加锁开销大、并发效率高、会死锁

**注意：**InnoDB中行级锁需要作用在索引字段。当执行 `UPDATE`、`DELETE` 语句时，如果 `WHERE` 条件中字段没有命中唯一索引或者索引失效的话，就会导致扫描全表、尝试逐行加行级锁！

### Next-key Lock加锁范围

- **Next-key Lock只会对查找到的行加锁，扫描过程不会加锁**

- 执行主键（唯一索引）的等值查询时，Next-key Lock**优化**：

- 主键（唯一索引）的等值查询，数据存在时，会加**行级锁**
- 主键（唯一索引）的等值查询，数据不存在时，会给查询的值所在区间加**间隙锁**

非索引字段查询时，InnoDB会执行全表扫描、给每行尝试加行锁；**不会加间隙锁（因为没有索引，无法确定顺序和范围）**

## 意向锁Intension Lock

表锁与行锁不兼容，要加表锁需要先判断是否已有行锁，但是一行行扫描开销太大  
意向锁是表级的，用于在**表级别**和**行级别**之间建立一种协调关系，**快速判断能否加表锁**。

**当前事务的当前读操作之前，需要加上意向锁，方便后续其他事务加表锁时的快速判断：**

- **意向共享锁IS**：事务有意向对表中的某些记录加S锁，加锁前必须先取得该表的IS锁
- **意向排他锁IX**：事务有意向对表中的某些记录加X锁，加锁前必须先取得该表的IX锁

**意向锁的兼容性：**

- 意向锁之间相互兼容
- 意向锁和行级锁兼容
- 涉及到实际的表级锁时，只有**表级S锁和IS锁兼容**

## MySQL性能优化

### 1. 慢SQL定位与分析

- 使用**MySQL慢查询日志**，定位执行过慢的SQL语句
- 使用**EXPLAIN**命令，分析查询计划，特别是索引使用情况 `ref / index / all`

### 2. 数据库执行优化（内部优化）

#### ◦ 索引优化：

- 优先选择WHERE条件、ORDER BY GROUP BY DISTINCT、JOIN ON的字段，建立联合索引
- 区分度高、长度小、常用的索引放在左侧，遵从最左匹配原则
- 优先考虑覆盖索引，避免回表

#### ◦ 表结构优化：

- 避免TEXT/BLOG存文件，用COS
- 表字段用最小的数据类型：
  1. IP地址可以用内置函数转化为整形存取
  2. 非负型数据考虑无符号UNSIGNED，多出一倍存储空间
  3. 小范围整形（年龄、状态）用TINYINT，仅占1字节8位
- 合理反范式

正常要满足第三范式，但可能会拆分出多个表、增加关联查询、降低查询效率，因此可以保留一点冗余信息

#### ◦ SQL语句优化：

- 避免 `SELECT *`，消耗更多CPU和IO，无法使用覆盖索引（非常好索引）
- 避免索引失效：不最左匹配、索引列用函数、like通配符、隐式转换
- 用连接查询代替子查询，因为子查询会生成临时表、该表无法使用索引
- 合理使用分页

### 3. 数据库架构优化（外部优化）

- **读写分离**：将读操作和写操作分离到不同的数据库实例，提升数据库的并发处理能力。

**分库分表**: 将数据分散到多个数据库实例或数据表中，降低单表数据量，提升查询效率。但要权衡其带来的复杂性和维护成本，谨慎使用。

**数据冷热分离**: 根据数据的访问频率和业务重要性，将数据分为冷数据和热数据，冷数据一般存储在低成本、低性能的介质中，热数据高性能存储介质中。

**缓存机制**: 使用 Redis 等缓存中间件，将热点数据缓存到内存中，减轻数据库压力

## 分库分表

### 水平分库分表

水平的分库和分表其实挺像的

- **水平分库**: 根据某些字段（如用户 ID、订单 ID、时间等）对数据进行切分，保证每个库存储的数据量大致相同。
  - 例如，将订单按订单 ID 的范围分到多个库中，ID 从 1-1000000 的订单存储在数据库 A，1000001-2000000 的订单存储在数据库 B。
- **水平分表**: 将表按某种规则拆分成多个表，每个表存储部分数据。例如，将单个订单表拆分为多个订单表，按订单 ID 范围分表。
  - **按 ID 范围分表**: 例如订单 ID 为 1-1000 在表 A，1001-2000 在表 B。
  - **按哈希分表**: 将订单 ID 取模后分配到不同的表（如 `order_id % 10`）。
  - **按时间分表**: 根据时间段划分，如将 2023 年的订单放到一个表，2024 年的订单放到另一个表。

### 垂直分库分表

- **垂直分库**: 根据业务模块的不同将数据划分到不同的数据库中。例如，将用户数据存储在一个数据库中，订单数据存储在另一个数据库中。
- **垂直分表**: 将表按字段划分，减少单个表的字段数量，使每个表的列数更少，便于提高查询效率。例如，将一个订单表拆成两个表，一个存储订单的基本信息，另一个存储订单的详细信息。

## 不推荐用外键和级联

意思是不应该在数据库层面添加**外键约束和级联更新**，但是外键作为一个字段经常是有必要存在的  
外键与级联更新适用于单机低并发，不适合分布式、高并发集群；级联更新是强阻塞，存在数据库更新风暴的风险；外键影响数据库的插入速度。

外键应该放在应用层手动解决。

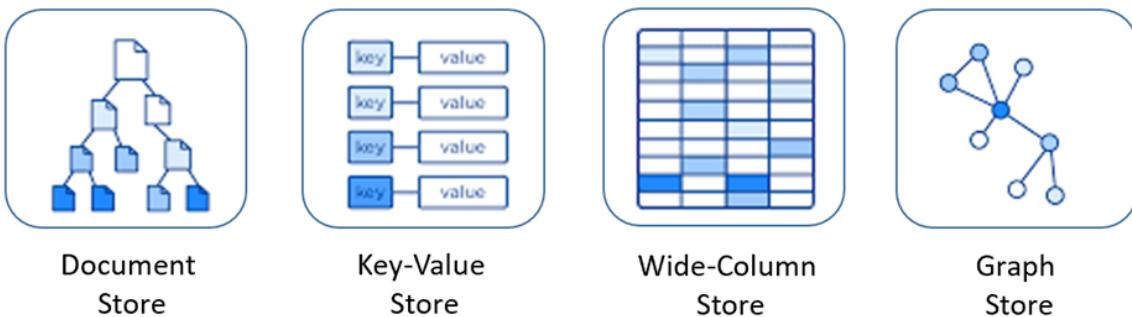
## drop truncate delete

- `drop` : `drop table 表名` , 直接将整个表都卸载掉
- `truncate` : `truncate table 表名` , 清空表中的数据，再插入数据的时候自增长id又从 1 开始
- `delete` : `delete from 表名 where 列名=值` , 删除某一行的数据，如果不加 `where` 子句和 `truncate table 表名` 作用类似

### DDL和DML

- `truncate` 和 `drop` 是**DDL(数据定义语言)**语句，操作**立即生效**，原数据不放到 rollback segment**不能回滚，不触发 trigger**
- `delete` 是**DML(数据库操作语言)**语句，这个操作会放到 rollback segment 中，**事务提交之后才生效**，提交后可以回滚

## NoSQL - Redis



NoSQL数据库主要可以分为下面四种类型：

- **键值Key-Value**: 简单灵活，可以完全控制 value 字段中存储的内容，没有任何限制。Redis 和 DynanoDB
- **文档Document**: 数据被存储在类似于JSON对象的文档中，清晰直观。每个文档包含成对的字段和值，这些值通常可以是字符串、数字、布尔值、数组或对象等，并且它们的结构通常与开发者在代码中的一致。MongoDB
- **图Graph**: 应用于内部高度连接的数据集。典型场景社交网络、推荐引擎、欺诈检测和知识图谱。Neo4j 和 Giraph
- **宽列Wide-Column**: 适合存储大数据、高性能读写。Cassandra 和 HBase

## Redis

**REmote DIctionary Server**，一个C语言开发的NoSQL数据库。

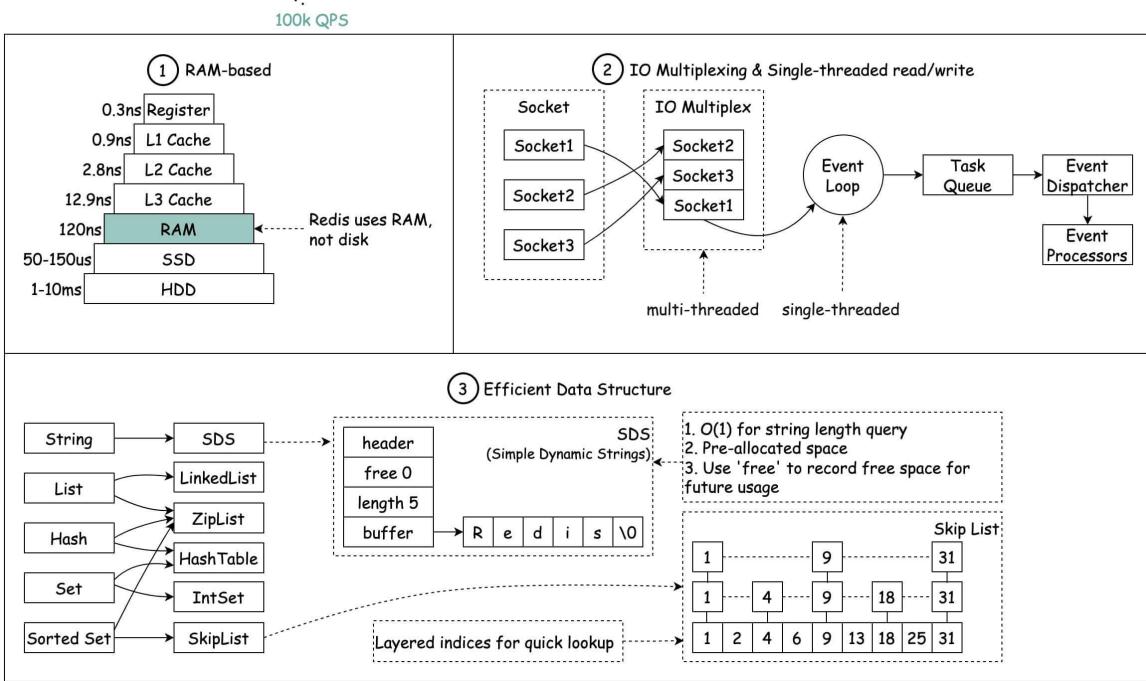
- 数据保存在内存中，读写速度非常快，常用作分布式缓存
- Redis以键值对形式存储数据，key为String类型，value可以是 String/Hash/List/Set/Bitmap/Geo...
- Redis还支持事务、持久化、Lua 脚本、发布订阅模型、多种集群方案（Redis Sentinel、Redis Cluster）

为什么不用作主数据库？——内存成本高、Redis提供的持久化有数据丢失风险

## Redis为什么速度快

Redis单条查询延迟是1毫秒内，QPS是10w+；MySQL单条查询在10毫秒，QPS是1k+

## Why is Redis so fast?



1. Redis基于内存，内存的访问速度比磁盘快很多

寄存器Register, 缓存L1/L2/L3 Cache都是CPU内部的；RAM是内存/主存；SSD固态和HDD机械都属于磁盘DISK

Random Access Memory; Solid-State Disk; Hard Disk Drive

2. Redis基于单线程事件循环和IO多路复用，形成了一套高效的事件处理模型

3. Redis内置了多种优化过后的数据类型/结构实现，性能非常高

## Redis和Memcached

除了Redis，其他的分布式缓存方案：

- Memcached：早期常用的分布式缓存，没有Redis强大
- Dragonfly：完全兼容Memcached和redis、最快速，但生态和资料不如Redis

区别	Redis	Memcached
数据类型	较为丰富	只支持字符串（复杂的要序列化）
持久化	可以将内存数据存进磁盘，可灾难恢复	只能把数据存在内存中
线程模型	单线程、IO多路复用	多线程、非阻塞IO复用网络模型
特性支持	订阅发布、Lua脚本、事务、原生集群	
过期数据删除	惰性删除、定期删除	只有惰性删除

## Redis数据类型

## 基本数据类型

String	List	Hash	Set	Zset
SDS	LinkedList/ZipList/QuickList	Dict+ZipList	Dict+Intset	ZipList+SkipList

特殊数据类型：Bitmap、HyperLogLog、Geospatial

## String

最简单常用，可以储存任何二进制数据（整形、字符串、编码、序列化对象）。没有用C的原生字符串，Redis构建了简单动态字符串 **SDS**（Simple Dynamic String）：1. 用len属性记录长度(不用遍历计算) 2. 一开始两倍扩容 - 后面慢扩容 3. 惰性缩容 4. 可存储 \0 二进制数据。

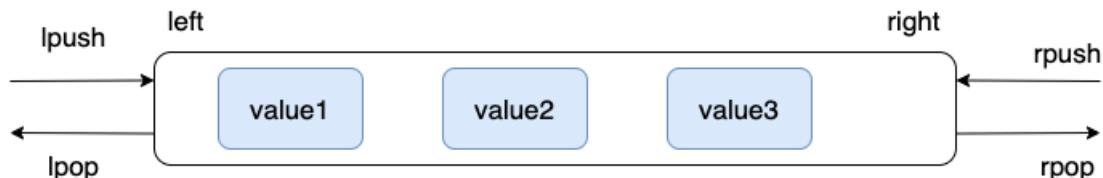
- SET key value
- **SETNX** key value: 只有当key不存在才设置值
- GET key
- MSET key1 value1 key2 value2 ...
- MGET key1 key2
- STRLEN key: 获取value长度
- INCR key / DECR key: 将key储存的数值+1 / -1
- EXISTS key
- DEL key (通用)
- EXPIRE key seconds (通用): 设置key的过期时间

场景：

- 缓存Session、Token、图片地址、序列化对象（比Hash节省内存），用SET GET
- 限流一定时间的用户请求、记录网页一定时间内的访问数，用SET GET INCR DECR
- 简易分布式锁（例如防止重复下单），用SETNX
- 全局唯一ID生成：1. UUID工具包 2. Redis自增（可结合雪花思想）3. 雪花算法（时间戳+序号自增）4. 数据库自增

## List

Redis使用一个双向链表来实现List，支持反向查找遍历



- RPUSH key value1 (value2...): 列表尾部（右侧）添加元素
- LPUSH key value1 (value2...): 列表头部添加元素
- LPOP key: 移除并获取列表头元素（最左）
- RPOP key
- LLEN key: 获取列表元素数量
- LRANGE key startIndex endIndex: 获得列表指定范围内的元素[start, end]

场景：

- 信息流展示，如文章动态，用LPUSH LRANGE
- 简易消息队列，但是缺陷过多，建议用Redis5.0的Stream

## Hash

Redis中的Hash是一个String类型的键值对field-value集合（平面表，不能再嵌套）。

**原理：**底层使用两个dict哈希表（桶+链表），`ht[0]`正常使用，`ht[1]`在扩容时使用。当负载>1扩容一倍，从`ht[0]`往`ht[1]`搬迁；但不是一次性迁移避免阻塞，而是每次增删改查渐进式迁移；迁移过程中仍可读写（先查表1再差表2）。

- HSET key field value: 给指定hash表中的指定field字段设置值
- HSETNX key field value: 表中field不存在时才设置
- HMSET key field1 value1 f2 v2...
- HMGET key f1 f2
- HGETALL key: 获取指定hash表中所有键值对
- HEXISTS key field
- HDEL key f1 (f2...): 删除字段
- HLEN key: 获得表中字段数量
- HINCRBY key field increment: 指定字段做运算，正数加负数减

场景：存储对象数据

## Set

无序非重复集合，相比于List提供了判断元素是否在某个Set内的接口；实现了交并集、差集操作

- SADD key member1 member2...: 向指定Set添加元素
- SMEMBERS key: 获取指定Set所有元素
- SCARD key: 获得元素数量
- SISMEMBER key member: 是否在set中
- SINTER key1 key2...: 交集
- SINTERSTORE destination key1 key2...: 交集结果存储在destination中
- SUNION key1 key2...: 并集（也有相应的STORE）
- SDIFF key1 key2...: 差集（也有STORE）
- SPOP key count: 随机移除n个元素
- SRANDMEMBER key count: 随机获取n个元素

场景：

- 访问量/点赞数，用SCARD
- 共同好友、好友推荐、订阅号推荐等，用SINTER SUNION SDIFF (STORE)
- 随机获取如抽奖点名，用SPOP SRANDMEMBER

## Sorted Set

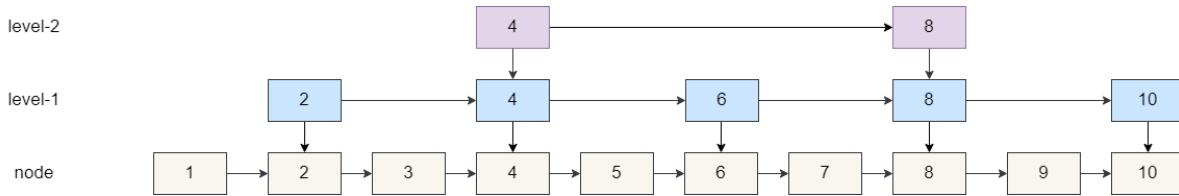
有序集合，在Set基础上添加了权重参数score，使得集合有序排列、能够获取范围

- ZADD key score1 member1 (score2 member2...)
- ZREM key member1 (member2...): 移除元素
- ZCARD key: 获取数量
- ZSCORE key member: 活鹅指定元素的score
- ZINTER / ZUNION / ZDIFF / (...STORE)
- ZRANGE / ZREVRANGE key startscore endscore: 获取指定score范围的元素，从低到高/高到低
- ZRANK / ZREVRANK key member: 获取指定元素的排名，按score低到高/高到低排序

场景：需要排序或优先级的场景，如排行榜、优先任务队列，用ZRANGE、ZRANK

## SortedSet底层：skiplist跳表

SortedSet在一定阈值下（元素<64B, 个数<128）使用ziplist（连续数组），超过则使用跳表skiplist。跳表在链表的基础上建立了多级索引，改进了链表O(n)时间复杂度，插入、删除、查询都为O(log n)。跳表还能够进行快速地范围查找。



理想情况，每一级索引个数都是下一级的一半。Redis运行在内存，一般来说元素个数不超过 $2^{16} = 65536$ ，因此索引高度不建议超过16。跳表按照随机概率建立高一级索引（1级索引0.5, 2级索引0.25，以此类推），是概率平衡、而非强制平衡。

- **AVL平衡树vs跳表：**AVL时间复杂度和跳表一样，但严格地维持左右子树的绝对平衡，而跳表是概率平衡；AVL每次插入删除都可能要旋转，比跳表耗时
- **红黑树vs跳表：**红黑树允许稍微不平衡的情况，查询比AVL慢、插入删除比AVL快，但是相较于跳表依然实现更复杂、范围查找不如跳表
- **B+树vs跳表：**B+树的意义是尽可能减少IO，实现依然复杂

## Bitmap位图

基于String（底层就是二进制数据）通过offset进行位操作，用于表示多个连续元素的状态集、节省存储空间

offset 0	offset 1	offset 2	offset 3	offset 4
1	0	0	1	1

- SETBIT key offset value: 设置指定offset的值，value为0或1
- GETBIT key offset
- BITCOUNT key start end: 获取位范围内1的个数
- BITOP operation destkey key1 (key2...): 对一个或多个Bitmap进行位运算，operation可选AND/OR/XOR/NOT

场景：状态信息集，如签到、行为统计等

## HyperLogLog基数统计

一种基数计数概率算法，能够去重统计大量元素，Redis实现并进行了优化、节省空间（当计数较少时使用稀疏矩阵、较多时使用稠密矩阵，占用12k）

- PFADD key element1 (element2...)
- PFCOUNT key1 (key2...): 获取若干个HyperLogLog的唯一计数
- PFMERGE destkey key1 key2...: 合并若干个HyperLogLog算出唯一计数

场景：数据量巨大的计数场景，如热门网站访问统计、热门帖子UV统计

## Geospatial地理空间

简称GEO，用于存储地理位置经纬度信息，基于Sorted Set实现，可以计算距离、获取附近元素

- GEOADD key longitude1 latitude1 member1(long2 lat2 m2...): 添加元素对应的经纬度GEO
- GEOPOS key member: 获取指定元素经纬度
- GEODIST key member1 member2 unit: 计算距离, unit可选m/km/mi/ft
- GEORADIUS key long lat radius unit: 按地点半径查找, 末尾可加ASC/DESC, COUNT等参数
- GEORADIUSBYMEMBER key member radius unit: 按元素位置半径查找
- ZREM key member: 移除元素。GEO底层Sorted Set可以直接用相关命令

## Redis持久化

Redis不同于Memcached的重要方面

3种持久化方式：

1. 快照 snapshotting, RDB
2. 只追加文件 append-only file, AOF
3. RDB与AOF混合持久化 (Redis4.0)

**三者不互斥，可以同时打开多个持久化！**

### RDB

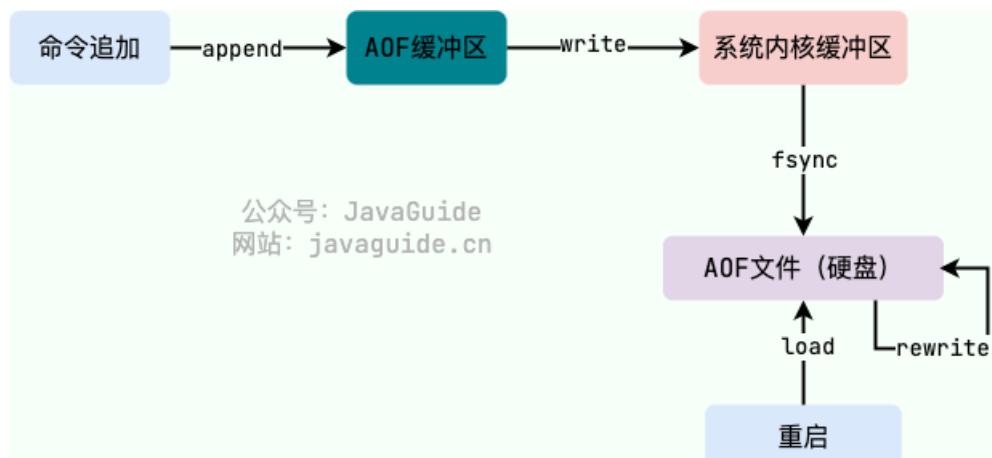
快照持久化，创建内存数据在某个时间点上的副本。提供了两个命令进行RDB持久化：

- save: 同步保存，阻塞Redis主线程 (Redis单线程，主线程就是进程)
- bgsave: fork出一个子进程进行保存，不影响主进程。Redis默认选项，可以在 `redis.conf` 配置文件中进行自动化设置 (`save 300 10` 表示每300s若有10个key发生变化则bgsave)

### AOF

类似binlog。比RDB更实时、更安全，Redis6.0默认开启。

执行流程：每执行一条数据更改指令，将会把该指令写入AOF缓冲区，然后再写入系统内核的AOF文件中（文件系统缓存page cache），最后再根据fsync策略来决定何时同步到磁盘。硬盘中的AOF需要定期rewrite重写以压缩空间；Redis重启后通过硬盘的AOF文件进行数据恢复到内存。



3种AOF持久化策略 (fsync策略) :

- appendfsync always: 每次write完了都立刻fsync刷盘, fsync完了线程才返回, 安全但性能低
- appendfsync everysec: 每秒钟同步一次, 线程write完了直接返回、由子线程刷盘, 兼顾
- appendfsync no: 只write不fsync, 系统决定何时刷盘 (linux默认30s一次)

**AOF在命令执行完才记录日志, 为了不阻塞当前命令执行**

**AOF重写**: 实际上通过读取数据库中的键值对进行, 和现有AOF无关。使用子进程执行重写、避免影响主进程。重写期间会维护一个**AOF重写缓冲区**, 记录服务器正在执行的命令, 重写完成后再追加到新AOF文件, 实现一致性

**Redis重启-AOF校验**: Redis重启时, 会对AOF进行完整性检查 (校验和checksum)

## RDB + AOF: 混合持久化

RDB优点: 压缩的二进制数据, 占用空间小; 恢复简单 (直接解析还原文件数据)

RDB缺点: 安全性实时性不足, 有丢失风险; 不易于理解和人工分析

AOF优点: 更安全、实时; 支持秒级丢失、同步操作轻量 (仅需追加命令)

AOF缺点: 文件体积大需要重写; 性能开销大 (经常刷盘); 恢复复杂 (慢需要一条条执行命令)

### 混合持久化

结合RDB + AOF: AOF重写时, 直接把RDB写到AOF文件开头。好处是快速加载、降低丢失风险。默认关闭。

# Redis线程模型与IO

- 对于读写命令来说, Redis 一直是**单线程模型** (主事件循环)
- Redis4.0 引入异步命令来优化大键值对操作
- Redis6.0 正式引入了多线程来处理网络请求, 提高网络 IO 读写性能

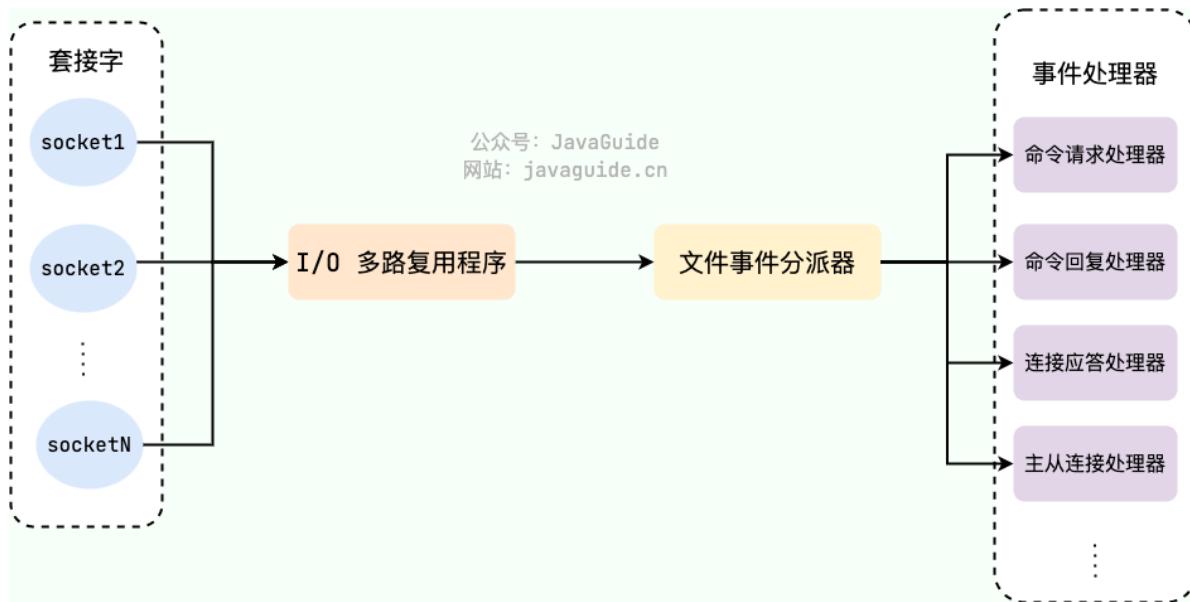
### 单线程

**Reactor模式**: 一种用于处理并发 I/O 操作的设计模式, 由事件多路复用器Selector、事件分发器Reactor、事件处理器Handler组成, 工作流程为注册事件-监听-分发-处理

Redis基于 Reactor模式设计开发了一套高效的事件处理模型: 通过**IO 多路复用程序**来监听客户端的多个socket, 主线程通过事件循环依次处理这些事件。

好处:

- Redis 的性能瓶颈不在 CPU, 主要在内存和网络。事件处理是单线程处理, 而IO是多路复用的。
- 不需要多线程来监听多个客户端连接, 减少开销。
- 多线程会存在死锁、线程上下文切换等问题, 甚至会影响性能; 单线程编程容易并且更容易维护、不用加锁。



## 多线程(Redis6.0)

Redis 的瓶颈主要受限于内存和网络，多线程主要是为了提高网络 IO 读写性能，因此读写仍然是单线程的。修改 redis 配置文件 `redis.conf` 进行配置。

### 后台线程

- 释放 AOF / RDB 等过程中产生的临时文件资源
- 调用 `fsync` 函数将系统内核缓冲区还未同步数据强制刷盘（AOF 文件）
- 释放已删除的大对象

## 内存管理：过期时间

内存珍贵，一般存入数据要设置过期时间（由过期字典维护），通过 `expire key 60` 设置60s过期、`persist` 移除过期

### 过期key删除策略

- 惰性：读写时才检查过期，CPU友好、内存压力大
- 定期：周期性抽查，CPU不友好
- 延迟队列：放入延迟队列、到期删除，维护麻烦
- 定时：分别维护定时器，CPU压力最大

Redis采用定期+惰性删除，平衡CPU和内存压力

## Redis性能优化

### 慢查询

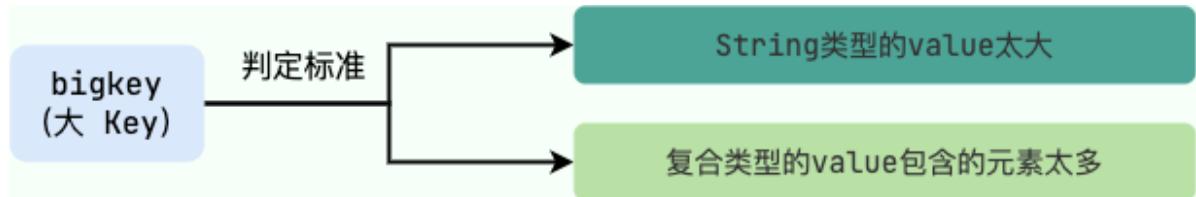
可以用自带 `SLOWLOG GET` 命令查询慢日志。

尽量不使用 O(n) 以上复杂度过高的命令（如 `keys`, `hgetall`），对于数据的聚合操作放在客户端做

## key集中过期

1. 给 key 设置随机过期时间。这是一个常见的实践
2. 开启 lazy-free 延迟释放：Redis 4.0 引入，采用异步方式延迟释放 key，将该操作交给单独的子线程处理，避免阻塞主线程

## bigkey



操作大key可能造成阻塞，体现在三方面：工作线程阻塞、网络阻塞、客户端响应超时阻塞

可以用自带 `--bigkeys` 来查找，或借用第三方工具分析

**删除bigkey/清空数据库**：一下子多出大量空闲空间需要整理（空闲内存块链表），阻塞主线程。建议分批删除、异步删除

## hotkey

访问量暴增的热点数据是hotkey，可能超出redis处理能力，是性能瓶颈。

可以用自带的 `--hotkeys` 来查找或借助其他工具，或根据业务提前预估和监控。

解决办法：读写分离，主节点写、多个子节点读；Redis集群；二级缓存，存一份hotkey到jvm本地

## 持久化阻塞

### 1. save创建RDB快照

默认配置用 `bgsave`，fork一个子进程生成RDB，但是如果手动执行了save命令则会阻塞主线程

### 2. 重写时fork阻塞

fork操作发生在RDB和AOF重写时，redis主线程调用fork操作产生共享内存的子进程，由子进程完成持久化文件的重写工作。如果fork操作本身耗时过长，必然后导致主线程的阻塞。要避免产生过大的redis实例

### 3. AOF刷盘阻塞

AOF持久化方式（`fsync` 策略）设置为always时严重降低性能；当磁盘压力过大时也会阻塞AOF日志的同步操作

## Swap内存交换

redis快是因为它在内存里，如果发生了swap、部分数据被换出内存进入磁盘，则造成致命影响

(系统命令)根据redis进程号查询内存交换信息来确认是否发生了swap。需要留足够内存来预防swap

## 网络阻塞

使用批量操作，进行少次多量的网络IO，原生批量操作 `MSET`、`MGET`、或使用 pipeline

# Redis生产问题

## 缓存穿透

无效请求，同时穿透了缓存和数据库（key完全不存在，数据库返回空数据），导致数据库宕机。常见于黑客攻击

- **缓存空对象**，即无效请求的【key-null】键值对。消耗内存，可以加过期时间TTL缓解。
- **布隆过滤器**，快速查找key是否可能存在，不合理的key直接视为无效请求。内存占用小，但复杂且小概率误判。  
布隆过滤器实际上是一个位图数组bitmap，根据数据库数据的hash算出，若key不存在于布隆那必是假key，但key通过了布隆则不一定是真key。
- 对异常IP进行限流

## 缓存击穿（热点key问题）

突然高并发的请求穿透了缓存、落到了数据库（存在但未缓存的key）。常见于新出现的热点key，比如秒杀；且缓存重建的相关业务非常复杂、耗时长，导致缓存一定时间失效。

- **提前预热**：将热点数据提前加入缓存、设置合理过期时间。一般结合逻辑过期使用。
- **互斥锁**：缓存失效后，确保同时只有一个请求访问数据库、然后更新缓存，其他阻塞。可以使用`setnx`实现简易互斥锁。
- **永不过期/逻辑过期**：不设置过期时间，或者只是添加一个`expired`字段，过了热点时间手动删除，可结合提前预热。  
当发现时间已久、需要更新时，当前线程会建一个新线程去异步更新缓存（更新过程加锁），本身先返回旧数据。

## 缓存雪崩

缓存同时大面积失效（或缓存宕机），导致大量请求直接落到数据库

- 对于Redis不可用：Redis集群；多级缓存（本地+Redis）；容错处理降级限流（快速失败、牺牲服务）
- 对于缓存大面积失效：设置随机过期时间TTL

# 缓存策略与数据库一致性

## 缓存策略

- **cache aside旁路缓存**：调用者以数据库为主，数据库更新时主动更新缓存。
- **read/write through**：将数据库和缓存整合为一个服务，调用者不关心具体。复杂高成本。**阿里  
canne**
- **write behind cache缓存写回**：调用者以缓存为主，后台线程异步将缓存写入数据库。一致性低。

# Cache Aside Pattern 旁路缓存的一致性分析

四种方案：更新数据库-更新缓存；更新缓存-更新数据库；更新数据库-删除缓存；删除缓存-更新数据库。

两个问题：并发问题，并发写+写、并发读+写（读操作也可能写缓存）；两步操作，第一步成功第二步失败。

## 1. 并发问题

- 【更新数据库+更新缓存】：更新+更新的两种方案，并发写+写的情况下有一致性问题

```
线程A更新数据库 x=1
线程B更新数据库 x=2 (final)
线程B更新缓存 x=2
线程A更新缓存 x=1 (final)
```

此外有些时候只需要写数据库，不会再查询，更新缓存是无效写操作。

- 【删除缓存-更新数据库】：并发读+写的情况下有一致性问题，需要考虑到读操作也可能写缓存

```
线程A删除缓存x=1
线程B读x,缓存未命中,从数据库读到旧值写入缓存 x=1 (final)
线程A更新数据库 x=2 (final)
```

- 【更新数据库-删除缓存】：并发读+写也有问题，但需要读数据库-写缓存的时间，比更新数据库-删缓存的时间长，一般来说不会出现这种情况。

因此当选择**更新数据库-删除缓存 (Cache Aside Pattern)**，可以基本解决并发一致性问题。

此外还有一种方案：**延迟双删**，即在一定延迟后再次删除缓存，解决了并发不一致问题。

## 2. 两步操作，可能第一步成功第二步失败。此处的一致性问题无法直接解决，使用重试机制：

- 同步重试：失败了直接重试，大概率还是失败，不行
- 异步重试：引入消息队列

现在有一种新方案：**订阅数据库变更日志binlog，自动更新缓存**。这也是read/write through缓存策略，现在**阿里cannel**实现了这种方式。

# Redis应用场景

## 1. 缓存

- 分布式锁**：通常基于Redisson或setnx实现，能够控制**多个JVM进程**（部署在不同的服务器）对共享资源的访问

JDK提供的ReentrantLock类、synchronized关键字等，能够控制一个JVM内的多个线程对本地共享资源的访问，这是**本地锁**

- 分布式Session**：利用String或者Hash数据类型保存Session数据，所有的服务器都可以访问

- 限流**：一般是通过Redis + Lua脚本的方式来实现限流

# Redis分布式锁

JDK提供的`ReentrantLock`类、`synchronized`关键字等，能够控制一个JVM内的多个线程对本地共享资源的访问，这是**本地锁**

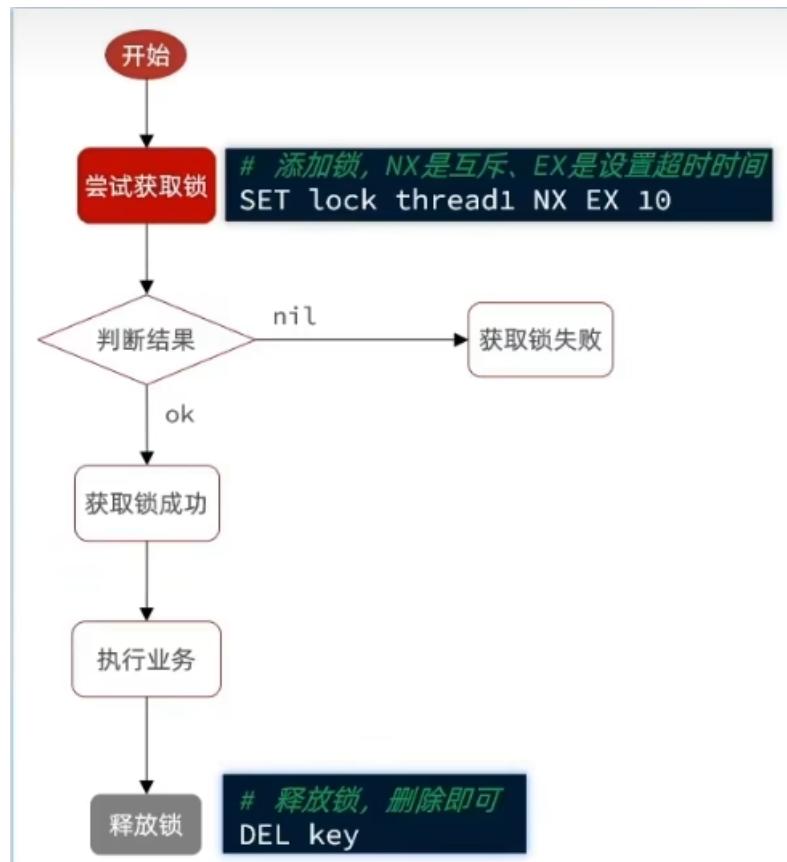
**分布式锁：**能够控制**多个JVM进程**（部署在不同的服务器）对共享资源的访问

- 多进程可见
- 互斥：锁同时只能被一个线程持有——`setnx`
- 高可用（安全）：锁不能被某一线程因为bug而永久持有，要保证最终能释放、不影响其他线程访问——`expire`
- 可重入：一个节点获取锁后，还可以再次获取锁
- 高性能

## 简易分布式锁 setnx

加锁：`set lockkey valuethread1 NX EX 10`，一条命令加锁、设置过期，保证原子性。

释放锁：`del lockkey`，放在finally块中保证释放，高可用。



- **误删问题：**

1. 线程1获取锁，业务阻塞、锁超时释放；
2. 线程2获取锁、执行业务中；
3. 而线程1在finally块中会再次释放锁，此时释放的就是已经被线程2获取的锁了。**线程1误删导致线程2不再线程安全。**

**防止误删：**锁中存储代表该线程的【线程标识】，释放锁时校验是否为当前线程，不是当前线程说明锁已经被别的线程占用。光用线程Id是不够的，集群下可能多线程Id相同；可以考虑用uuid代表一个jvm服务，分布式锁存储 `UUID+threadId`。

- **原子性问题 - 误删：**

1. 线程1获取锁、业务执行完、判断锁线程标识、准备释放锁（判断+释放非原子操作）；
2. jvm阻塞（比如full GC），释放锁的操作被阻塞，然后锁超时；
3. 线程2获取锁、执行业务，线程1再释放锁、导致误删。**线程1的判断+释放非原子操作，导致了误删。**

解决一致性问题：

- redis事务(不推荐)：实际在提交时批处理，原子性✓隔离性一致✗，不能失败回滚和数据加锁。需结合乐观锁。
- **lua脚本**：`redis.call(命令, key, [参数1,...])`；在redis内使用 EVAL 执行脚本

```
class SimpleRedisLock {
    private static final KEY_PREFIX = "lock:";
    private static final ID_PREFIX = UUID.randomUUID().toString() + "-";
    private String name;
    private StringRedisTemplate stringRedisTemplate;
    // fullArgs构造函数
    public SimpleRedisLock(...) {...}
    // 尝试获取锁
    public boolean tryLock(long timeoutSec) {
        // 锁存储[线程标识]，用于在释放锁时确保释放的是正确线程的锁，防误删问题
        // 光用线程Id不够，多线程下线程Id冲突；考虑uuid标识当前jvm，再进行拼接
        String threadId = ID_PREFIX + Thread.currentThread().getId();
        // 原子操作，key，value，time，timeunit
        boolean success = stringRedisTemplate.opsForValue()
            .setIfAbsent(KEY_PREFIX + name, threadId, timeoutSec,
TimeUnit.SECONDS);
        // 返回，防止null
        return Boolean.TRUE.equals(success);
    }
    public void unlock() {
        // 获取当前线程的[线程标识]
        String threadId = ID_PREFIX + Thread.currentThread().getId();
        // 获得锁中当前的[线程标识]
        String curLockId = stringRedisTemplate.opsForValue().get(KEY_PREFIX +
name);
        // 锁是当前线程占用的
        if (threadId.equals(curLockId)) {
            stringRedisTemplate.opsForValue().delete(KEY_PREFIX + name);
        }
    }
}
```

改为使用lua脚本解决误删问题一致性问题：

```
class SimpleRedisLock {
    private static final RedisScript<Long> UNLOCK_SCRIPT; // Redis执行脚本返回值是
Long
    static { // 初始化时就IO导入lua文件
        UNLOCK_SCRIPT = new DefaultRedisScript<>();
        UNLOCK_SCRIPT.setLocatoin(new ClassPathResource("unlock.lua"));
    }
    ...
    void unlock() {
        // redis执行脚本命令，execute(脚本, key参数集合, 其他参数...)
    }
}
```

```

        stringRedisTemplate.execute(
            UNLOCK_SCRIPT,
            Collections.singletonList(KEY_PREFIX + name),
            ID_PREFIX + Thread.currentThread().getId();
        )
    }
}

```

## setnx分布式锁的问题

- 不可重入
- 不可重试（获取不到直接返回了）
- 超时释放（timeout与业务无关，长业务有超时风险）
- 主从一致性：redis集群主从同步有延迟，主节点加锁set key立刻宕机，从节点未能同步锁，锁丢失，其他节点都可以抢到锁。

## Redisson

基于Redis和Java的分布式工具集合，in-memory data grid

[中文文档](#) | [官方英文文档](#)

配置：

```

@Configuration
public class RedissonConfig{
    @Bean
    public RedissonClient redissonClient(){
        Config config = new Config();
        // 单节点redis模式，设置ip地址和密码

        config.useSingleServer(). setAddress("redis://192.168...."). setPassword("123456");
        // 创建reddissonclient
        return Redisson.create(config);
    }
}

```

使用：在需要使用的类中IoC注入 `redissonClient`，即可使用它提供的各种锁

## 可重入的Redis分布式锁

### 解决不可重入问题

`setnx lockname thread` 不可重入的局限性在于只能通过key是否存在来判断是否加锁，而不能判断是谁加的锁。

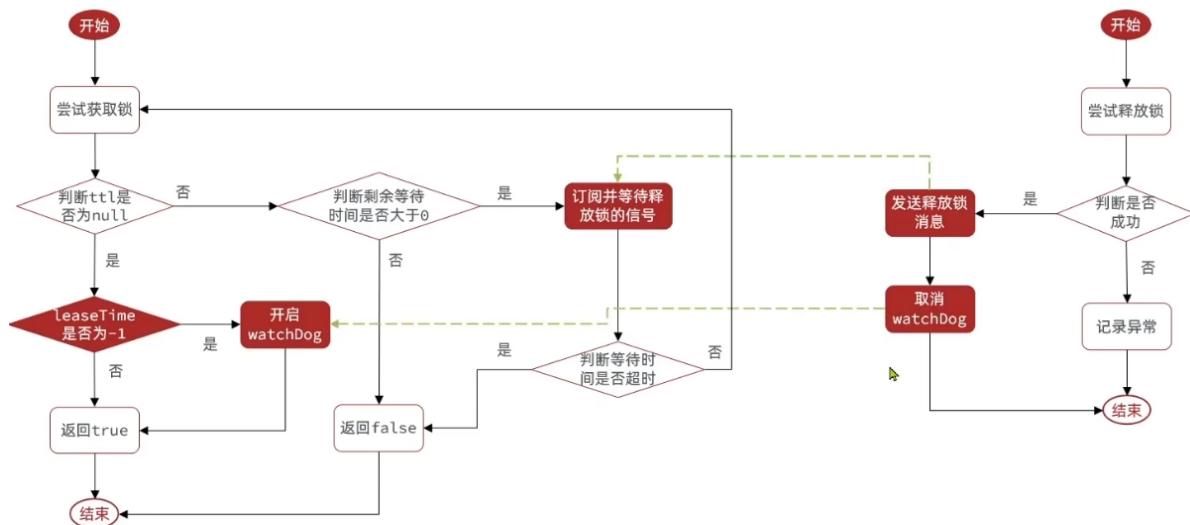
回顾：Java `ReentrantLock` 原理：**加锁时判断线程ID是否为自己，重入时计数器+1，退出时-1，为0则释放锁。**

redisson使用**hash**数据结构 `lockname - {thread: count}`。但是hash没有setnx这样的原子性操作，需要先判断再设置，于是用lua脚本保证原子性。

## redisson分布式锁原理 - 优化不可重试和超时释放问题

- 不可重试：通过参数 `waittime` 控制等待时间。  
redis释放锁时会发布事件；线程B获取锁失败后会订阅事件，超时时间内监听到该事件说明锁被释放，就可以尝试竞争锁；超时时间内通过循环和信号量去竞争锁。通过消息订阅和信号量避免高CPU占用的忙等。
- 超时释放：参数 `leasetime = -1` 开启watchdog机制，为获得锁的该线程设置定时任务（存在 `concurrentMap` 里），不断更新有效期，在长业务场景下就能防止超时释放

## Redisson分布式锁原理



**ttl**是尝试获取锁时返回的锁剩余有效期：若null表示该锁已被释放、当前线程可以去竞争锁；否则表示还剩多久锁才释放。

## MultiLock - 优化主从一致性

主从一致性：主节点加锁后宕机且未同步，导致锁丢失。

设置多个独立(主)节点，给所有(主)节点加上锁才算成功。这样即使有1个节点宕机锁丢失，其他节点仍拿着锁，不会被趁虚而入。

**红锁RedLock**：不需要所有都拿锁，超过一半拿到锁即视为成功。

# Maven

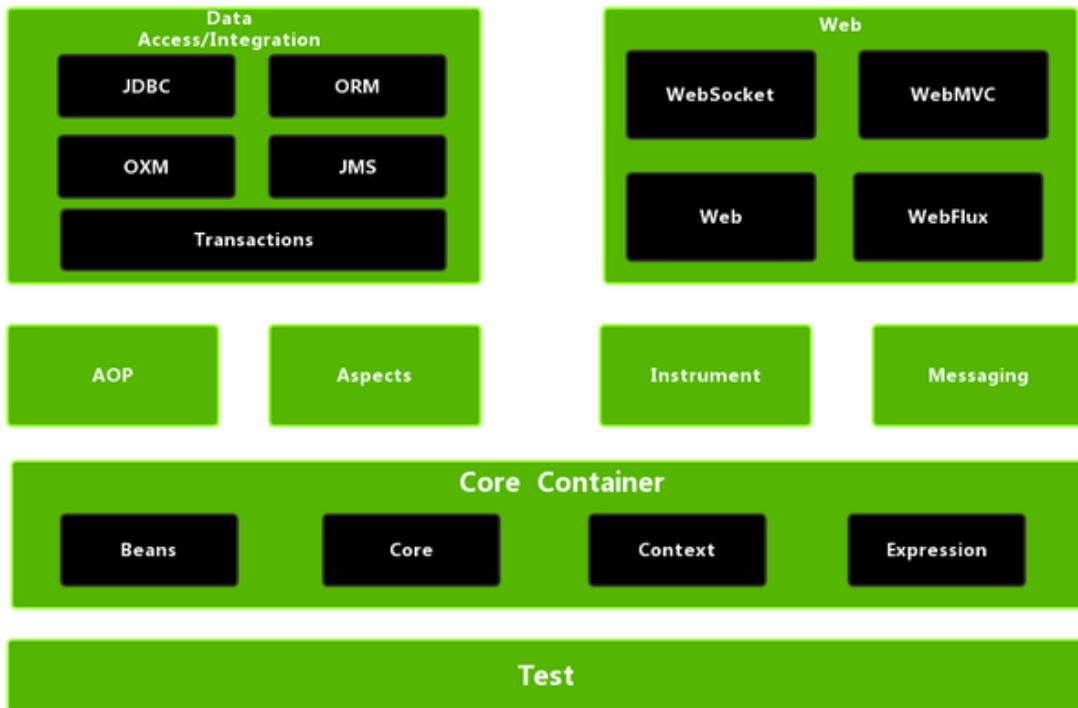
## 核心概念总结

# Spring

## Spring基础

Spring是一款Java开发框架，**Spring提供的核心特性主要是IoC和AOP**，在此基础上设计了许多辅助开发的模块。

# Spring模块



## 1. Core Container 核心模块

核心模块，其他所有功能依赖的基础模块，主要提供 IoC 依赖注入功能的支持

- spring Core: Spring 框架基本的核心工具类。
- spring Beans: **Bean** 的创建、配置和管理等功能
- spring Context: 提供对国际化、事件传播、资源加载等功能的支持。
- spring Expression: 提供对表达式语言 (Spring Expression Language) SpEL 的支持

## 2. AOP 切面模块

- spring AOP: 提供了**AOP面向切面**的编程实现。
- spring Aspects: 该模块为与 **AspectJ 集成**提供支持。
- spring Instrument: 提供了为 JVM 添加代理 (agent) 的功能。具体来讲，它为 Tomcat 提供了一个织入代理

## 3. Data Access/Integration 数据模块

- spring JDBC: 提供了对**数据库访问**的抽象 JDBC。不同的数据库都有自己独立的 API 用于操作数据库，而 Java 程序只需要和 JDBC API 交互，这样就屏蔽了数据库的影响。
- spring Transactions: 提供对**数据库事务**的支持。
- spring ORM: 提供对 Hibernate、JPA、iBatis 等 ORM 框架的支持。**Object-Relational Mapping****对象关系映射**, **Java对象-数据库表**
- spring OXM: 提供如 JAXB、Castor、XMLBeans 等 OXM 框架支持。**Object-XML Mapping**, **Java对象-XML**
- spring JMS: **Java消息服务**。自 Spring Framework 4.1 以后，它还提供了对 spring-messaging 模块的继承

## 4. Web模块

- spring Web: 对 Web 功能的实现提供一些最基础的支持。
- spring WebMVC: 提供对 **Spring MVC** 的实现。
- spring WebSocket: 提供了对 **WebSocket** 的支持，WebSocket 可以让客户端和服务端进行双向通信。

- spring Webflux：提供对 WebFlux 的支持。WebFlux 是 Spring Framework 5.0 中引入的新响应式框架。与 Spring MVC 不同，它不需要 Servlet API，是完全异步。

## 5. Messaging模块

spring Messaging 是从 Spring4.0 开始新加入的一个模块，主要职责是为 Spring 框架集成一些基础的报文传送应用。

## 6. Spring Test 测试模块

面向 TDD 测试驱动开发，提供对JUnit（单元测试）、Mockito（用来Mock对象）、PowerMock（解决Mockito缺点比如无法模拟 final/static/private方法）等常用测试框架的支持

# Spring / SpringMVC / SpringBoot

- Spring

**Spring 提供的核心功能主要是 IoC 和 AOP。** Spring包含了多个功能模块，其中最重要的是 Spring-Core（主要提供 IoC 依赖注入功能的支持），Spring 中的其他模块（比如 Spring MVC）的功能实现基本都需要依赖于该模块

- SpringMVC

Spring中的一个重要模块，能够快速构建 MVC 架构的 Web 程序（Model Controller View，数据、业务逻辑、显示三者分离）

- SpringBoot

开启某些 Spring 特性时，需要用 XML 或 Java 进行显式配置，较为麻烦；SpringBoot旨在简化配置，开箱即用

# Spring IoC

## IoC/DI简介

**Inversion of Control 控制反转：**将原本在程序中手动创建对象的控制权交给第三方比如 IoC 容器

- **控制**：指的是对象创建（实例化、调用、管理）的权力
- **反转**：控制权由代码本身交给外部环境（IoC 容器）来装配和管理

传统方法使用new实例化对象，依赖关系需要手动管理（比如有一个接口的具体实现类改为另一个）；使用IoC思想，则简化资源管理方式，实例化由IoC容器负责。

**Dependency Injection 依赖注入**是IoC思想的最佳实现方式。

传统方式：硬编码，在内部进行外部依赖的实例化

```
public class Service {  
    private Repository repository;  
    public Service() {  
        this.repository = new Repository(); // 强耦合  
    }  
}
```

DI方式：

## 1. 构造器注入

```
public class Service {  
    private Repository repository;  
    public Service(Repository repository) { // 构造器注入  
        this.repository = repository;  
    }  
}
```

适用于强依赖（必须依赖），但是可能造成构造器臃肿

## 2. Setter注入

```
public void setRepository(Repository repository) { // Setter 注入  
    this.repository = repository;  
}
```

适用于可选依赖，灵活可变，但是可能造成依赖未设置问题

## 3. 接口注入，不常用

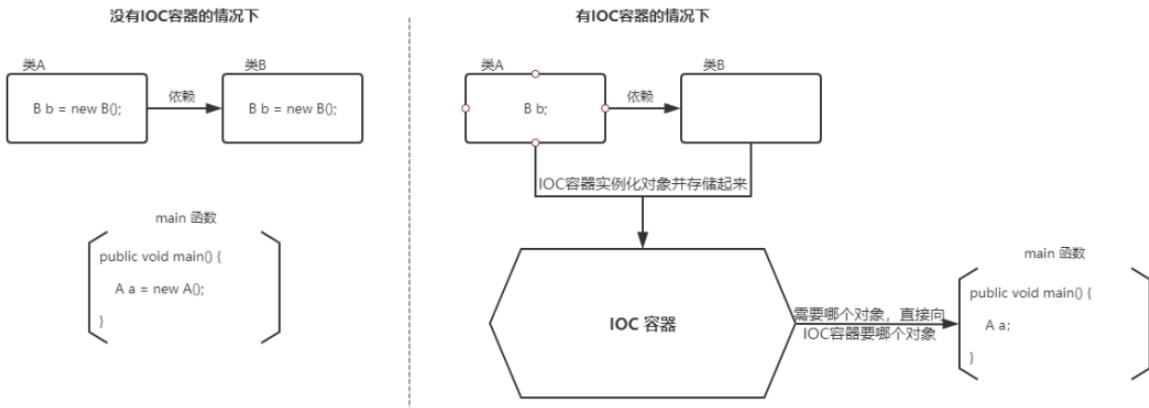
```
public interface Injectable {  
    void injectDependency(Repository repository);  
}  
  
public class Service implements Injectable {  
    private Repository repository;  
    @Override  
    public void injectDependency(Repository repository) {  
        this.repository = repository;  
    }  
}
```

# Spring IoC容器

Spring容器（IoC容器）是Spring用来实现IoC的载体，实际上就是个key-value Map，Map中存放的是各种对象。

Spring容器核心组件：

- BeanFactory：最基本的IoC容器，延迟加载。
- ApplicationContext：功能更强大的容器，支持国际化、事件传播、AOP等。



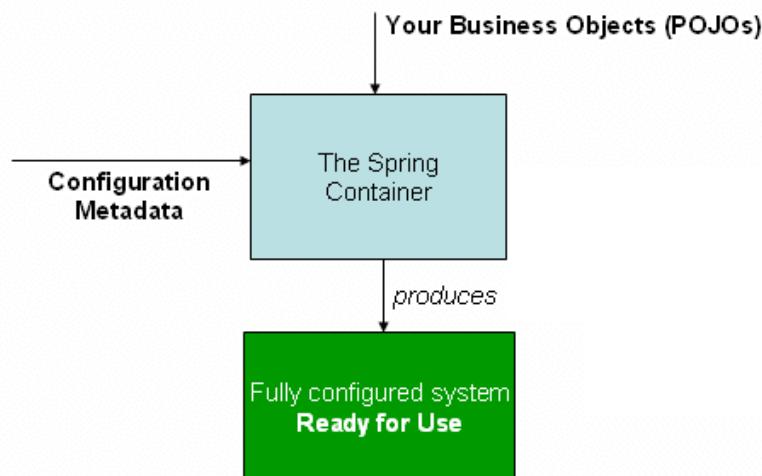
例如：在实际项目中一个 Service 类可能依赖了很多其他的类，假如我们需要实例化这个 Service，你可能要每次都要搞清这个 Service 所有底层类的构造函数，如果利用 IoC 的话，你只需要配置好，然后在需要的地方引用就行了，这大大增加了项目的可维护性且降低了开发难度。

## Bean

**Bean** 代指的就是那些被 IoC 容器所管理的对象。

使用IoC的能力时，需要先把一个对象声明为Bean，然后在需要使用的地方进行注入。

声明Bean，需要配置metadata提供给IoC Bean容器：可以通过XML文件、注解、Java配置类进行



## 声明一个类为Bean的注解

- `@Component`：通用的注解，可标注任意类为 `spring` 组件。如果一个 Bean 不知道属于哪个层，可以使用 `@Component` 注解标注
- `@Repository`：对应持久层即 Dao 层，主要用于数据库相关操作。
- `@Service`：对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao 层。
- `@Controller`：对应 Spring MVC 控制层，主要用于接受用户请求并调用 `service` 层返回数据给前端页面。

`@Bean` 只能加在方法上

**@Component和@Bean：**

- `@Component` 注解作用于类，而 `@Bean` 注解作用于方法。

- `@Component` 通过类自动扫描来自动侦测并装配到 Spring 容器中（可以使用 `@ComponentScan` 定义要扫描的路径，从中找出标识了需要装配的类，自动装配到 Spring 的 bean 容器中）。
- `@Bean` 需要在该方法中产生这个bean、显式定义返回值。`@Bean` 在容器中注册了这是某个类的实例，之后当需要用时提供给我
- `@Bean` 注解比 `@Component` 更灵活、更精确、更自定义。有时只能通过 `@Bean` 注解来注册 bean，比如当我们引用第三方库中的类需要装配到 Spring 容器时、或是对 Bean 初始化有复杂特殊要求。

```
// 这个例子无法使用component达到效果
@Bean
public OneService getService(status) {
    case (status) {
        when 1:
            return new serviceImpl1();
        when 2:
            return new serviceImpl2();
        when 3:
            return new serviceImpl3();
    }
}
```

## 注入Bean的注解

Spring 内置的 `@Autowired`；JDK 提供的 `@Resource` 和 `@Inject` 都可以用于注入 Bean。

`@Autowired` 可在构造函数、方法、字段和参数上使用。`@Resource` 主要在字段和方法上注入，不支持在构造函数或参数上使用。

### `@Autowired`

- 默认注入方式为 `byType`，根据类型匹配 Bean；
- 当接口有多个实现类则会注入方式为 `byName`，根据变量名匹配；  
建议通过 `@Qualifier` 注解来显式指定名称

```
// 例子：SmsService接口有俩实现类SmsServiceImpl1, SmsServiceImpl2

// 报错，byName 和 byType 都无法匹配到 bean
@Autowired
private SmsService smsService;

// 正确注入 SmsServiceImpl1 对象对应的 bean
@Autowired
private SmsService smsServiceImpl1;

// 正确注入 SmsServiceImpl1 对象对应的 bean
// smsServiceImpl1 就是我们上面所说的名称
@Autowired
@Qualifier(value = "smsServiceImpl1")
private SmsService smsService;
```

### `@Resource`

- 默认注入方式 `byName`，名称匹配不到则 `byType`
- 可以指定 `name` 和 `type` 两个属性，注入方式会根据指定属性而选择（不建议两个属性都指定）

```
// 报错, byName 和 byType 都无法匹配到 bean
@Resource
private SmsService smsService;

// 正确注入 SmsServiceImpl 对象对应的 bean
@Resource
private SmsService smsServiceImpl;

// 正确注入 SmsServiceImpl 对象对应的 bean (推荐)
@Resource(name = "smsServiceImpl")
private SmsService smsService;
```

## Bean注入的场景 (在代码中的注入方式)

需要注意的是，Bean注入是Spring IoC容器实现的**自动装配**（被注入对象在实例化时就会自动将依赖的Bean注入），而不需要像之前诠释**DI方式**场景手动在外部实例化依赖

1. 构造函数注入
2. Setter注入
3. Field字段注入：直接在类的字段上使用注解（如 `@Autowired` 或 `@Resource`）来注入依赖项

```
@Service
public class UserService {
    // field-based DI
    @Autowired
    private UserRepository userRepository;

    // constructor-based DI
    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    // setter-based DI
    @Autowired
    public void setUserRepository(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

推荐使用构造函数注入：保证依赖使用非空、有助于创建不可变对象(线程安全)、方便测试

构造函数注入适合处理**必需的依赖项**，而 **Setter注入** 则更适合**可选的依赖项**。

## Bean作用域

- 通用
  - **singleton**单例Bean: IoC 容器中只有唯一的 bean 实例。默认为单例Bean (单例设计模式的应用)
  - **prototype**原型Bean: 每次获取, 如 `getBean()`, 都会创建一个新的 bean 实例
- 仅Web应用可用
  - request 请求Bean: 每一次 HTTP 请求都会产生一个新的 bean, 仅在当前 HTTP request 内有效
  - session 会话Bean: 每一次来自新 session 的 HTTP 请求都会产生一个新的 bean, 仅在当前 HTTP session 内有效
  - application/global-session 应用Bean: 每个 Web 应用在启动时创建一个 Bean, 在应用关闭前有效
  - websocket 套接字Bean: 每一次 WebSocket 会话产生一个新的 bean

通过 `@Scope` 注解配置作用域:

```
@Bean  
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)  
public Person personPrototype() {  
    return new Person();  
}
```

## Bean线程安全

prototype作用域下, Bean在每次获取时都会创建一个新的, 不存在线程安全问题。

singleton作用域下, 若是**无状态Bean** (没有可变成员变量) 的, 则也线程安全, 实际上大部分Bean (如Service、DAO) 都是无状态的; **有状态Bean** (包含可变的成员变量) 则需要考虑线程安全问题。

解决办法:

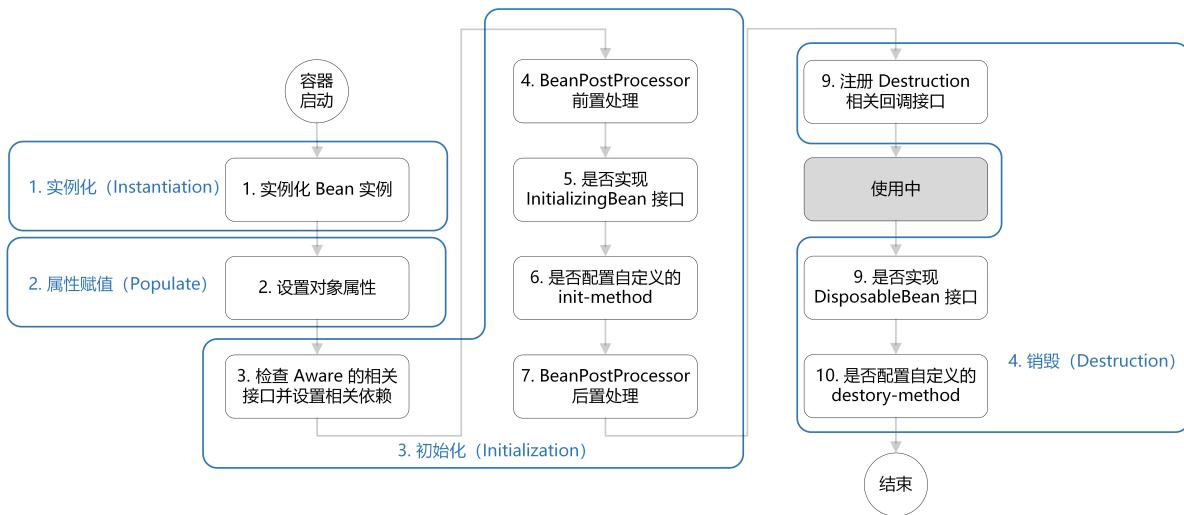
1. `ThreadLocal` 确保线程独立

```
public class UserThreadLocal {  
    private UserThreadLocal() {}  
    private static final ThreadLocal<SysUser> LOCAL =  
        ThreadLocal.withInitial(() -> null);  
    public static void put(SysUser sysUser) {  
        LOCAL.set(sysUser);  
    }  
    public static SysUser get() {  
        return LOCAL.get();  
    }  
    public static void remove() {  
        LOCAL.remove();  
    }  
}
```

2. 同步机制: `synchronized` 或 `reentrantLock`

## Bean生命周期

实例化 → 属性赋值 → 初始化 → 销毁



- 1. 实例化Instantiation**: Spring容器使用 Java反射API 创建Bean实例
- 2. 属性赋值Populate**: 容器根据配置文件 (XML、注解或 Java 配置类) 对 Bean 的属性进行依赖注入 (构造器、Setter、字段)
- 3. 初始化Initialization**
  - **检查Aware注入相关依赖**: 用于在bean中获取相应的Spring容器资源
    - 若 bean 实现了 BeanNameAware 接口，注入当前 Bean 对应的名字
    - 若 bean 实现了 BeanClassLoaderAware 接口，注入加载当前Bean的 classLoader
    - 若 bean 实现了 BeanFactoryAware 接口，注入当前 BeanFactory 容器对象的引用
    - .....
  - **前置处理**: 初始化前，调用 BeanPostProcessor 自定义的前置函数
  - **实际初始化 (伴生回调: 接口实现或配置文件)**：
    - 若实现了 InitializingBean 接口，执行 afterPropertiesSet() 方法
    - 若配置中指定了初始化方法 (`init-method` 或 `@PostConstruct` 注解)，则调用该方法
  - **后置处理**: 初始化后，调用 BeanPostProcessor 自定义的后置函数
- 4. 销毁Destruction**: 初始化完成后先记录下销毁方法，使用完毕后再调用
  - **实际销毁 (伴生回调: 接口实现或配置文件)**：
    - 若实现了 DisposableBean 接口，执行 `destroy()` 方法
    - 若配置中指定了销毁方法 (`destroy-method` 或 `@PreDestroy` 注解)，则调用该方法

### 四大扩展点：

1. Aware: 用于在bean中获取相应的Spring容器资源
2. PostProcessor: 用于在初始化前后修改bean或监控
3. 初始化: 当前bean初始化完成后的伴生回调，用于补充初始化操作
4. 销毁: 当前bean即将销毁前的伴生回调，用于补充销毁操作

### PostProcessor vs 初始化/销毁回调

- **PostProcessor**: 容器级别的扩展，用于拦截所有Bean的初始化前后处理，由接口实现、作为一个单独的Bean注册到容器中

- **初始化/销毁回调**: 仅作用于当前单个Bean内部的扩展点，通常由注解驱动（挨个实现Bean内部接口导致耦合度高）

# Spring AOP

## AOP简介

### Aspect-Oriented Programming 面向切面编程

OOP 不能很好地处理一些分散在多个类或对象中的公共行为（如日志记录、事务管理、权限控制、接口限流、接口幂等等），这些行为通常被称为 **横切关注点 (cross-cutting concerns)**。AOP与OOP互补，AOP实现了**横切逻辑和业务逻辑的分离**。

- AOP: 将**横切关注点**从核心业务逻辑中分离出来，通过动态代理、字节码操作等技术，实现横切代码的复用和解耦
- OOP: 将业务逻辑按照**对象**的属性和行为进行封装，通过类、对象、继承、多态等概念，实现代码的模块化和层次化

AOP术语	意义
连接点 joinpoint	被通知的对象中的所有方法都可以作为连接点，执行方法时的某个特定时刻（调用后、抛出异常时）。切面通过连接点判断具体是什么业务方法
切点 pointcut	被切面拦截或增强的连接点
通知 advice	拦截到连接点后要进行的操作、进行功能增强的代码，也就是切点的实现
切面 aspect	切面=切点+通知。切面是对横切关注点封装、成为类，一个切面一个类。
织入 weaving	将通知应用到切点对应的连接点上。有编译时织入 (AspectJ) 和运行时织入 (SpringAOP)

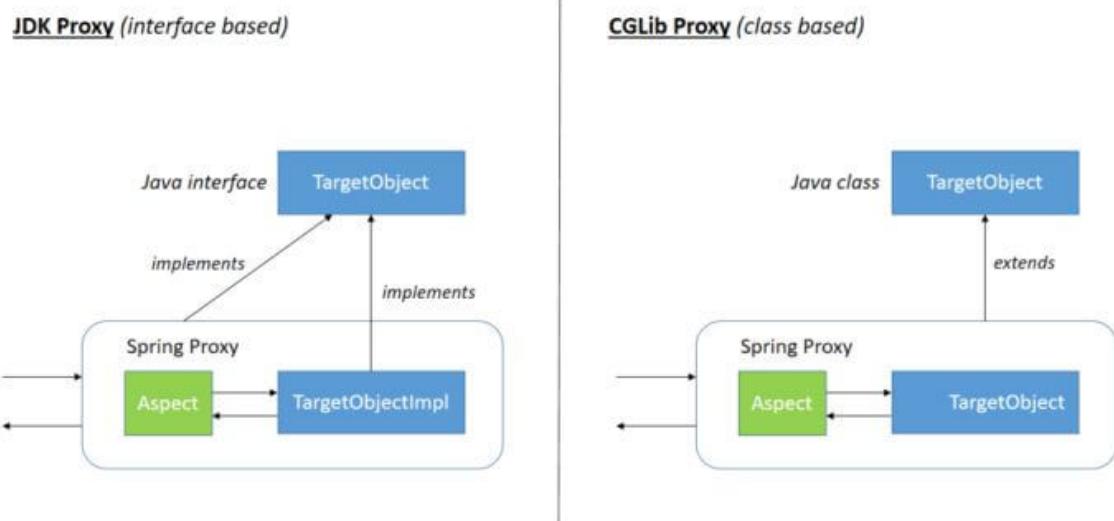
AOP实现方式：

1. **SpringAOP**: 基于**Proxy**, 运行时增强
2. **AspectJ**: 基于**字节码操作**, 编译时增强 (已集成到SpringAOP中, 更复杂但功能更强大)

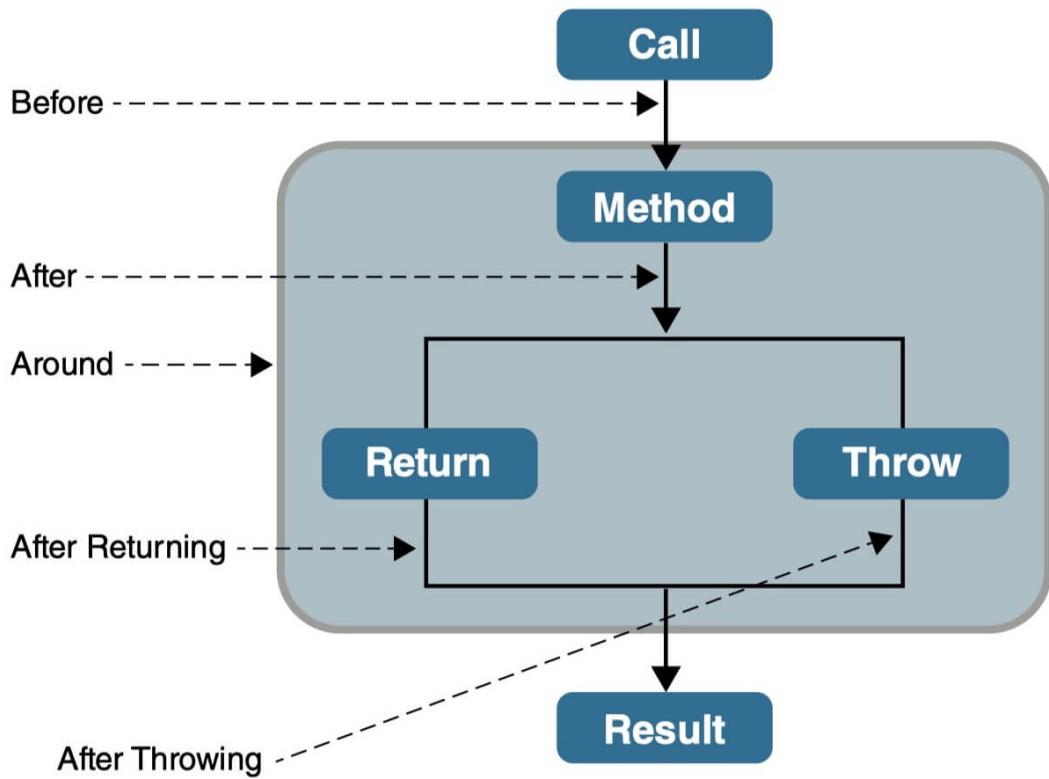
## SpringAOP 原理

- 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
- 对于没有实现接口的对象：无法使用Proxy，使用 **CGLib** 生成一个被代理对象的**子类**来作为代理

## Spring AOP Process



## Advice通知类型



- Before / After: 前置/后置通知，在方法调用前后执行
- AfterReturning / AfterThrowing: 返回结果值后 / 抛出异常后，二者互斥
- Around: 环绕通知，功能最强，可以获得目标对象和执行的方法

## 实战

1. 自定义注解 `public @interface xxxAnnotation`, 注解到目标方法上
2. 定义切面对象 `xxxAspect` 类, 使用 `@Aspect @Component` 注解标注为切面
3. 使用 `@PointCut` 注解注入切点 (即上面的自定义注解)
4. 使用 `@Around` 等通知, 处理切点、实现AOP

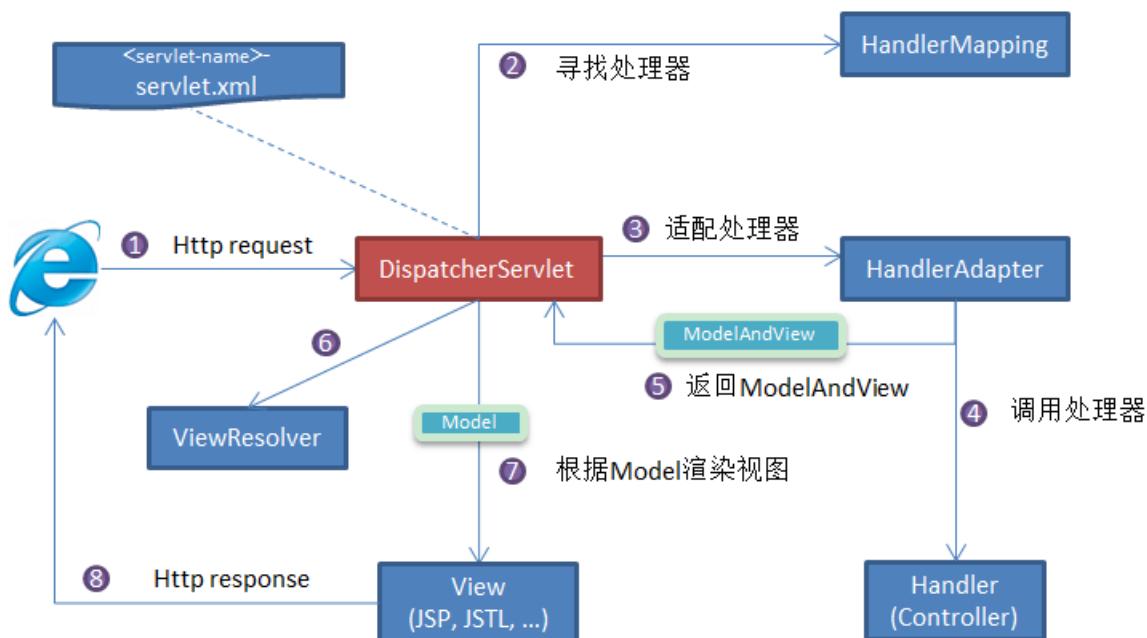
```

// 自定义注解LogAnnotation,希望通过切面实现"调用方法打印日志"
@Aspect
@Component
public class LogAspect {
    @Pointcut("@annotation(com.whq.annotation.LogAnnotation)") // 切点全限名注入
    public void logPointCut() {}

    @Around("logPointCut()")
    public Object around(ProceedingJoinPoint point) throws Exception {
        System.out.println("调用了" + point.getSignature().getName() + "方法"); // 日志输出
        return point.proceed(); // 执行原方法
    }
}

```

## SpringMVC



### 工作流程（前后端不分离，JSP）：

1. 客户端（浏览器）发送请求，**DispatcherServlet** 中央处理器拦截请求。
2. **DispatcherServlet** 调用 **HandlerMapping 映射器**。**HandlerMapping** 根据 URL 去匹配 **Handler 请求处理器**（也就 **Controller 控制器**），并会将请求涉及到的拦截器和 **Handler** 一起封装。
3. **DispatcherServlet** 根据匹配到的目标 **Handler**，调用对应 **HandlerAdapter 适配器**，去执行该 **Handler**。
4. **Handler 请求处理器** 完成对用户请求的处理后，会返回一个 **ModelAndView 对象**给 **DispatcherServlet**，包含数据和逻辑视图信息。
5. **DispatcherServlet** 调用 **ViewResolver 视图处理器**，根据 **ModelAndView** 中的逻辑视图，查找真正的 **view**。
6. **DispatcherServlet** 进行视图渲染，把视图返回给客户端。

**前后端分离架构下**，**handler** 不返回 **ModelAndView**、直接返回 JSON 数据，而 **ViewResolver** 也不会被调用。开启方法：

- 使用 `@RestController` 注解代替传统的 `@Controller`，这样处理器方法默认都会返回JSON格式数据，而不是试图解析视图
- 若仍使用 `@Controller`，可以结合 `@ResponseBody` 注解来返回 JSON

## 过滤器和拦截器

- **过滤器Filter**: **Servlet 层面的拦截**，适用于整个Web、拦截所有请求（包括静态资源），**在 DispatcherServlet之前执行**。
- **拦截器Interceptor**: **SpringMVC 层面的拦截**，仅拦截 Controller 逻辑层。

# Spring设计模式

- **工厂模式**: 使用工厂模式通过 `BeanFactory`、`ApplicationContext` 创建 bean 对象
- **代理模式**: SpringAOP使用代理模式实现
- **单例模式**: bean默认单例
- **模板方法模式**: `JdbcTemplate`等以Template结尾的类（通常是对数据库的操作）
- **观察者模式**: Spring事件驱动模型
- **适配器模式**: SpringAOP的增强或通知(Advice)使用到了适配器模式、springMVC 中也是用到了适配器模式适配 `Controller`

# Spring循环依赖

## 三级缓存解决循环依赖

两个Bean互相引用对方时形成了循环依赖，同时考虑到AOP动态代理，Spring使用**三级缓存**来正确创建它们：

```
// 一级缓存
/** Cache of singleton objects: bean name to bean instance. */
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);

// 二级缓存
/** Cache of early singleton objects: bean name to bean instance. */
private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);

// 三级缓存
/** Cache of singleton factories: bean name to ObjectFactory. */
private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);
```

1. 一级缓存 `singletonObjects`：单例池，存放创建完成已就绪的bean（实例化 - 属性注入 - 初始化完成）。但是原型bean不在这里。
2. 二级缓存 `earlySingletonObjects`：存放仅实例化的bean（未属性注入），是三级缓存创建出来的对象，防止AOP代理时每次都从工厂里产生新的代理对象
3. 三级缓存 `singletonFactories`：存放 `ObjectFactory`，`ObjectFactory#getObject()` 方法会生成原始early bean或者AOP作用下的代理对象。只对单例Bean生效。

### 循环依赖时创建bean流程：

允许循环依赖时，bean在实例化时就**提前暴露**出去，将对应 ObjectFactory 加入三级缓存，该工厂可以动态生成早期bean或代理对象。

1. Spring尝试创建A (加入三级缓存)
2. A依赖了B， Spring尝试创建B (加入三级缓存)
3. B又依赖了A， Spring先到一级、二级缓存没找到A，于是在三级缓存调用 ObjectFactory#getObject() 方法去获取 A 的 **前期暴露的对像**early bean，将其加入二级缓存，并移除三级缓存
4. B将二级缓存中的A注入，循环依赖解决
5. 完成初始化后分别加入一级缓存，移除二、三级缓存

**为什么用工厂而不直接放bean早期对象？**因为AOP，有些bean会被增强，工厂可以延迟创建、动态决定是否创建增强对象。直接放bean就是一个固定的引用关系、可能导致代理丢失。

**只用两级缓存够吗？**不考虑AOP，一级（已就绪）+三级（工厂用于获取早期暴露对象）就够了，因为此时工厂暴露的早期对象是唯一且一致的。但是AOP动态代理时，当有多个对早期对象的引用，每次从工厂拿到的对象都可能生成是1个新的代理。三级缓存中的第二级避免了一个bean工厂生成多个代理对象。

**三集缓存能解决所有bean的循环依赖吗？**不能，三级缓存只能解决**单例bean**的循环依赖，而**原型bean**的循环依赖无解

## @Lazy解决循环依赖

@Lazy 注解用来标识类是否需要懒加载/延迟加载，可以作用在类上、方法上、构造器上、方法参数上、成员变量中。

- 若Bean没有@Lazy，则在Spring IoC容器启动时就会被创建
- 若Bean被@Lazy标注，则在第一次请求才进行创建

**@Lazy标注依赖属性，解决循环依赖：**

```
@Component
class A {
    @Autowired
    @Lazy
    private B b;
}

@Component
class B {
    @Autowired
    @Lazy
    private A a;
}
```

1. Spring创建A，A依赖了B，但B字段是**懒加载**的，因此先创建一个B的**代理对象**注入到A里面去放着，继续进行初始化；
2. Spring创建B，B依赖了A，而A已经创建完毕，正常注入。

因此，@Lazy解决循环依赖的关键点在于代理对象的使用。

**SpringBoot不允许循环依赖，这是一种代码缺陷；实在不行就用@Lazy权宜一下。**

## Spring事务

Spring事务实际上是对数据库事务的封装，所以只有使用支持事务的数据库（MySQL innodb引擎）时，Spring才能支持事务

### Spring两种事务管理方式

#### 1. 编程式事务管理

通过 TransactionTemplate 或者 TransactionManager 类，手动在代码中管理事务（使用较少、用于分布式）

#### 2. 声明式事务管理

基于AOP，使用XML配置或注解。最常见的是使用 @Transactional 注解public方法开启事务：

```
@Transactional(propagation = Propagation.REQUIRED)
public void aMethod {
    //do something
}
```

## Spring事务管理接口

关于事务管理，最重要的3个接口：

1. PlatformTransactionManager：平台事务管理器，事务管理核心。Spring不直接管理事务，而是提供这个接口统一规范事务管理行为，具体实现则由不同平台(jdbc,jpa)自己完成

```
public interface PlatformTransactionManager {
    //获得事务
    TransactionStatus getTransaction(@Nullable TransactionDefinition var1)
    throws TransactionException;
    //提交事务
    void commit(TransactionStatus var1) throws TransactionException;
    //回滚事务
    void rollback(TransactionStatus var1) throws TransactionException;
}
```

2. TransactionDefinition：事务定义信息，事务的具体属性（传播行为、隔离级别、超时、只读、回滚规则）。
3. TransactionStatus：事务运行状态。可以通过事务管理器获取。

```

public interface TransactionStatus{
    boolean isNewTransaction(); // 是否是新的事务
    boolean hasSavepoint(); // 是否有恢复点
    void setRollbackOnly(); // 设置为只回滚
    boolean isRollbackOnly(); // 是否为只回滚
    boolean isCompleted(); // 是否已完成
}

```

## 事务属性

事务属性 `transactionDefinition`，也是 `@Transactional` 注解的配置参数：

- `propagation`: 传播行为，默认为REQUIRED。
- `isolation`: 隔离级别，默认为DEFAULT（当前数据库的默认隔离级别，MySQL是可重复读），可指定四种隔离级别。
- `timeout`: 超时，指定事务最长执行时间，默认为-1（不设超时），超过则回滚。
- `isReadOnly`: 事务是否为只读，默认false。
- `rollbackFor`: 回滚规则，指定触发回滚的异常类型，默认为运行时异常`RuntimeException`; `Error`也回滚。

## 传播行为

**目的：解决业务层方法之间互相调用的事务问题**

```

// 例子：A的事务方法(外部)调用了B的事务方法(内部)
@Service
class A {
    @Autowired
    B b;
    @Transactional(propagation = Propagation.xxx)
    public void aMethod {
        //do something
        b.bMethod();
    }
}

@Service
class B {
    @Transactional(propagation = Propagation.xxx)
    public void bMethod {
        //do something
    }
}

```

枚举类 `Propagation`

1. `TransactionDefinition.PROPAGATION_REQUIRED` 需要

若外部方法A未开启事务，则当前方法B会新开启自己的独立事务；若外部A已开启事务，则B加入其中变成一个事务。默认值。

2. `TransactionDefinition.PROPAGATIONQUIRES_NEW` 需要新的

不管外部方法A，当前方法B总是会新开启一个独立的事务。A回滚不会影响B，但若B抛出异常、符合A的回滚规则，则会导致A也回滚。

### 3. `TransactionDefinition.PROPAGATION_NESTED` 嵌套

若外部方法A未开启事务，则B新开启一个自己的独立事务；若外部A已开启事务，则B在其内部开启一个新事物（嵌套事务）。

### 4. `TransactionDefinition.PROPAGATION_MANDATORY` 强制

若外部方法A已开启事务，则B加入；否则抛出异常

### 5. 其他则不是正确的传播，可能不会导致回滚

## isReadOnly：为什么数据库查询（读）还需要开启事务？

当事务开启`isReadOnly`，数据库会进行一些相应的优化。

MySQL默认对每个发到数据库服务器的语句开启事务，支持了服务端每次执行SQL的读一致性。

因此当执行单条SQL查询，没必要开启事务；而执行**多条SQL查询、需要保持整体前后一致时，就需要开启事务了**，否则每执行一条SQL都会读到更新的值。

## @Transactional原理

`@Transactional`基于SpringAOP，当一个public方法被其注解，Spring容器在启动时会为它创建一个代理类，当外部调用该方法时进行拦截、开启事务管理。

## AOP自调用问题

```
@Service
public class MyService {
    private void method1() {
        method2();
        //.....
    }
    @Transactional
    public void method2() {
        //.....
    }
}
```

MyService中，method1调用了`@Transactional`注解的method2，事务失效！

**Spring 事务的代理逻辑只对外部调用有效，内部调用直接绕过代理、事务逻辑无法触发。**

解决办法：

- 将该事务方法抽成单独的类
- 先`AopContext.currentProxy()`获取类的代理对象，让代理对象调用事务方法

## @Transactional使用注意事项

1. 事务要被支持：首先数据库本身要支持事务

2. 事务要能开启：原理是Spring容器代理类，意味着①**对象必须是Bean**，②**避免自调用**；

而不自调用就要外部调用，因此③只能注解在public上（在类上导致它的所有public都事务化，不推荐）

3. 事务要能回滚：正确配置propagation、rollbackFor

# JPA

## JDBC与DataSource

```
JPA(Hibernate) / MyBatis ---> DataSource ---> 数据库
```

**JDBC**是Java原生的连接数据的API，**DataSource**是它的核心组件，是用于连接的标准接口规范，可以管理**数据库连接池**。Spring中DataSource默认用Hikari实现，可以配置连接池。

`Application.yml`中配置数据库连接：

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/yuoj
    username: root
    password: 123456
    hikari:
      maximum-pool-size: 10      # 连接池中允许的最大连接数
      minimum-idle: 5           # 连接池中最小空闲连接数
      idle-timeout: 30000        # 空闲连接在被关闭之前的最长存活时间(ms)
      max-lifetime: 60000        # 连接的最大存活时间，超过此时间后即使连接是正常的也
      会被关闭(ms)
```

JPA和其他概念的关系：

```
JDBC # Java原生的管理数据库连接的底层API
- JPA # Java官方的ORM(Object-Relational Mapping)框架
  -- Hibernate # JPA规范的一种实现，被Spring Data JPA默认使用
  -- .... # 其他JPA实现

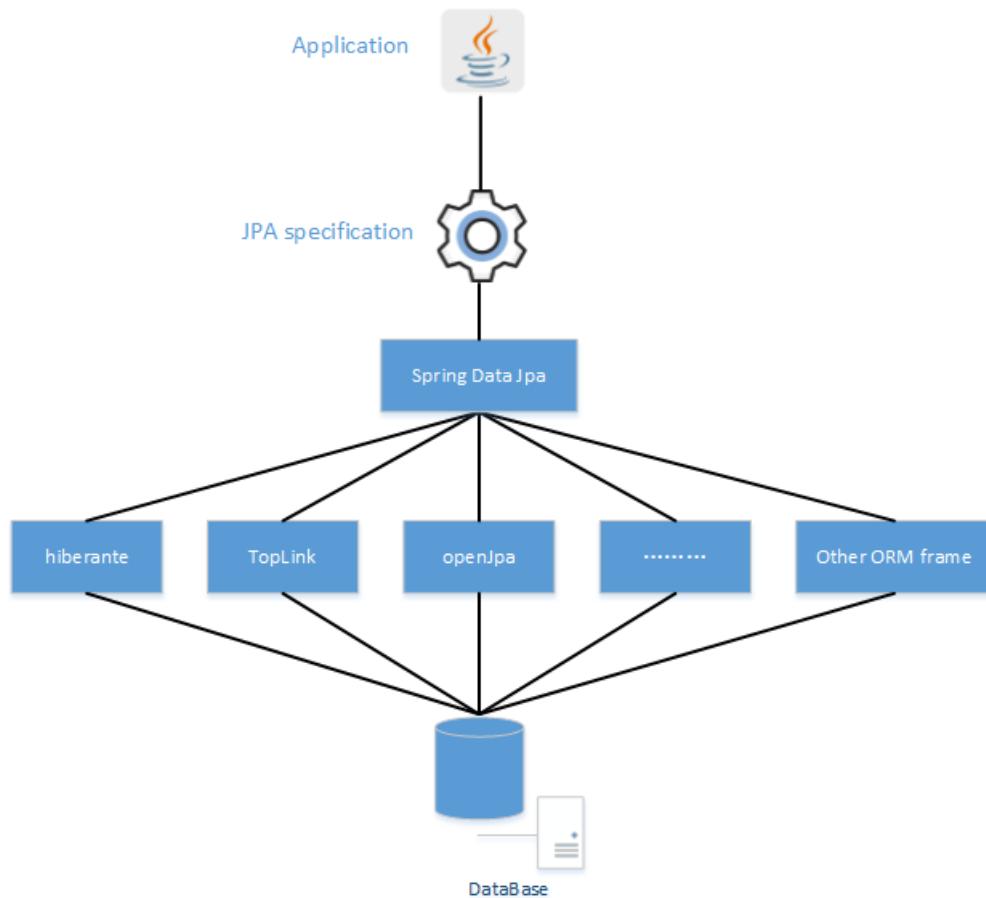
- Spring Data JPA # Spring对JPA的再封装，定义了DAO接口，更方便使用JPA。默认使用
  Hibernate

- MyBatis # 半ORM，SQL映射框架，手写SQL更灵活
  -- MyBatisPlus # 提供了更易用的API
```

## JPA、Spring Data JPA、Hibernate关系

- **JPA**: Java Persistence API，是**Java 标准中的一套ORM规范**，通过注解或者XML描述ORM(对象-关系表的映射关系)，并将实体对象持久化到数据库中。它提供了以下的工具：
  1. ORM元数据映射：XML或注解(@Entity @Table @Column @Transient)，框架自动映射、持久化
  2. Criteria API：对实体对象进行CRUD操作的API，框架自动转换成SQL
  3. JPQL：面向对象的查询语句
- **Spring Data JPA**: Spring提供的一套基于JPA规范的**DAO层接口** (Repository)，简化JPA数据访问API (Criteria API)，并且作为一个统一的**抽象层ORM框架**，方便接入不同厂商的ORM框架
- **Hibernate**: 实现了**API规范**的一种**ORM框架**，对持久化映射、数据操作进行了完整的实现封装，补充了一些JPA注解

JPA是规范；Hibernate是一种实现；SpringDataJPA对JPA再次封装，底层一般仍然是Hibernate。



spring data jpa、jpa以及ORM框架之间的关系 [sdn.net/cmx1060220219](http://sdn.net/cmx1060220219)

## JPA常用注解

### 1. 实体-关系表 映射

- **@Entity (必需) :** 映射一个类为JPA实体。按 驼峰→下划线 自动转换类名和字段名。
- **@Table:** 手动指定映射到的数据库表名，通过name属性指定表名， schema属性指定数据库名

### 2. 主键、生成策略 标记在成员变量或getter方法上

- **@Id (必需) :** 映射生成主键。若不使用@GeneratedValue则需要手动赋值才能存入。  
合法的主键类型：Java原始类型及包装类、String、BigDecimal、日期类、枚举类，以及它们的组合。  
指定联合主键，有@IdClass类注解、@EmbeddedId字段注解两种方法。
- **@GeneratedValue:** 主键生成策略，通过strategy属性指定：
  - **AUTO**：默认，JPA自动选择当前数据库最适合的，sqlServer是Identity、MySQL是自增
  - **IDENTITY**：数据库自动生成主键，Oracle 不支持
  - **SEQUENCE**：通过序列产生主键，通过 @SequenceGenerator 注解指定序列名，MySQL 不支持
  - **TABLE**：通过表产生主键，框架借由表模拟序列产生主键

### 3. 字段-列 映射 标记在成员变量或getter方法上

- **@Column:** 映射数据库表列。

注解的属性可以指定列的名称、类型：name属性手动指定列名，默认自动转换；unique属性指定该列内值唯一，默认false；nullable属性指定是否可空，默认false；length属性指定varchar类型长度，默认255；precision和scale属性指定double类型的长度和小数点位数

- **@Basic:** 简单的字段-列映射，对于没有注解的字段和getter，**默认标注为@Basic**（@Entity自动映射）
- **@Transient:** 忽略该字段。非持久化字段若不加该注解就成@Basic了。  
除了注解之外，还可以给字段定义加上final/static/transient关键字修饰。
- **@Temporal:** 时间日期精度。提供TemporalType枚举类，DATE/TIME/TIMESTAMP三种精度
- **时间日期自动更新:** JPA未定义，由ORM框架补充
  - Hibernate: @CreationTimestamp、@UpdateTimestamp
  - SpringDataJPA: @CreatedDate、@LastModifiedDate

#### 4. 实体间关系映射

**@OneToOne/@OneToMany/@ManyToOne/@ManyToMany:** 注解在字段上，表示To后面的。

常配合 @JoinColumn/@JoinColumns外键、@JoinTable关系表。

```
@Entity
public class User {
    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL) // 添加级联，默认关闭
    @JoinColumn(name = "user_id", nullable = false) // 外键列名称
    private List<Order> orders; // User 对 Order 是一对多
}

@Entity
public class Order {
    @ManyToOne // 不需要级联，因为Order是受控方
    private User user; // Order 对 User 是多对一
}
```

```
@Entity
public class Student {
    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"), // 当前实体 在关系表中的外键
        inverseJoinColumns = @JoinColumn(name = "course_id") // 关联实体 在关系表中的外键
    )
    private List<Course> courses; // Student 对 Course 是多对多
}

@Entity
public class Course {
    @ManyToMany(mappedBy = "courses")
    private List<Student> students; // Course 对 Student 也是多对多
}
```

# SpringBoot自动装配

自动装配是SpringBoot的核心

SpringBoot定义了一套接口规范，外部jar包实现该规范 (**starter**) 后，通过注解和简单配置即可将其功能引入程序之中，而无需再像过去一样进行复杂的手动配置。

## 原理

SpringBoot核心注解 `SpringBootApplication`

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
<1.>@SpringBootConfiguration
<2.>@ComponentScan
<3.>@EnableAutoConfiguration
public @interface SpringBootApplication {}


@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration //实际上它也是一个配置类
public @interface SpringBootConfiguration {}
```

可以看作是以下三个注解的集合：

- `@EnableAutoConfiguration`：引入自动装配类，启用 SpringBoot 的自动配置机制
- `@Configuration`：允许在上下文中注册额外的 bean 或导入其他配置类
- `@ComponentScan`：扫描被 `@Component` (`@Service`,`@Controller`)注解的 bean，默认会扫描启动类所在的包下所有的类，可以自定义排除某些 bean

核心实现：自动装配类 `AutoConfigurationImportSelector`

```
// AutoConfigurationImportSelector 继承关系
public class AutoConfigurationImportSelector implements DeferredImportSelector,
BeanClassLoaderAware, ResourceLoaderAware, BeanFactoryAware, EnvironmentAware,
Ordered {}

public interface DeferredImportSelector extends ImportSelector {}

public interface ImportSelector {
    String[] selectImports(AnnotationMetadata var1);
}
```

AutoConfigurationImportSelector实现了接口方法 `selectImports()`，用于获取所有需要装配的类，准备被加入IoC容器。

```

public String[] selectImports(AnnotationMetadata annotationMetadata) {
    // <1>. 判断自动装配开关是否打开
    if (!this.isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    } else {
        //<2>. 获取所有需要装配的bean
        AutoConfigurationMetadata autoConfigurationMetadata =
        AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);
        AutoConfigurationImportSelector.AutoConfigurationEntry
        autoConfigurationEntry =
        this.getAutoConfigurationEntry(autoConfigurationMetadata, annotationMetadata);
        return
        StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
    }
}

```

重点关注 `getAutoConfigurationEntry()` 方法，主要负责加载自动配置类：

```

private static final AutoConfigurationEntry EMPTY_ENTRY = new
AutoConfigurationEntry();

AutoConfigurationEntry getAutoConfigurationEntry(AutoConfigurationMetadata
autoConfigurationMetadata, AnnotationMetadata annotationMetadata) {
    //<1>. 判断是否打开自动装配，默认spring.boot.enableautoconfiguration=true，可通过
    application.yml配置
    if (!this.isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    } else {
        //<2>. 获取EnableAutoConfiguration注解中的exclude
        AnnotationAttributes attributes =
        this.getAttributes(annotationMetadata);
        //<3>. 获取需要自动装配的所有配置类，通过SpringFactoriesLoader读取META-
        INF/spring.factories
        List<String> configurations =
        this.getcandidateConfigurations(annotationMetadata, attributes);
        //<4>. 过滤掉不需要加载的配置类
        configurations = this.removeDuplicates(configurations); // 移除重复的
        Set<String> exclusions = this.getExclusions(annotationMetadata,
        attributes);
        this.checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions); // 移除注解中exclude的
        // 过滤@ConditionalOn...条件注解不满足的
        configurations = this.filter(configurations, autoConfigurationMetadata);
        this.fireAutoConfigurationImportEvents(configurations, exclusions);
        return new
        AutoConfigurationImportSelector.AutoConfigurationEntry(configurations,
        exclusions);
    }
}

```

【<3>. 获取要自动装配的所有配置类】是重点，它使用 `SpringFactoriesLoader` 读取 `META-INF/spring.factories` 文件。

不仅会读取SpringBoot自己的配置类文件 `spring-boot-autoconfigure/src/main/resources/META-INF/spring.factories`，也会读取所有**SpringBoot Starter**下面的 `META-INF/spring.factories` 文件，这就是外部jar所需要实现的starter接口。因此，通过在 `pom.xml` 里引入starter，SpringBoot进行自动装配时，就能把所有引入的外部jar装配进去，并且用条件注解过滤掉实际未使用的配置。

读取出来的这些配置类的命名都是 `xxxAutoConfiguration`，它们作用是通过条件注解 `@ConditionalOnxxx` 按需加载组件，自动装配。

```
@Configuration
// 检查相关的类:RabbitTemplate, Channel是否存在
// 存在才加载该配置类
@ConditionalOnClass({ RabbitTemplate.class, Channel.class })
@EnableConfigurationProperties(RabbitProperties.class)
@Import(RabbitAnnotationDrivenConfiguration.class)
public class RabbitAutoConfiguration {}
```

## 如何实现一个Starter

Starter相当于一个封装好的功能模块，它整合了所需要的依赖、对模块内的Bean进行自动配置。这就是Springboot自动装配。

实现流程

1. 创建：实现功能类、自动配置类、环境引入，打包到本地maven仓库
2. 使用：在目标项目内通过maven引入，通过IoC注入使用
3. 完善：使用拦截器达到开箱自动使用；增加配置项；等等

创建工程项目：`my-func-spring-boot-starter`

```
-- my-func-spring-boot-starter
  -- src
    -- main
      -- java
        -- com.whq
          -- interceptor
          -- configure
            MyFuncAutoConfiguration.java
          -- service
            MyFuncService.java
            -- impl
              MyFuncServiceImpl.java
            Application.java
      -- resources
```

开发功能类：`MyFuncService` 接口、`MyFuncServiceImpl` 实现类

```
package com.whq.service.impl;

import com.whq.service.MyfuncService;
public class MyfuncServiceImpl implements MyfuncService {
    ... // 可以注入其他依赖,比如@Autowired HttpServletRequest获得http请求
    @Override
    public void func() {...} // 自定义功能
}
```

### 创建自动配置类 xxxAutoConfiguration

```
package com.whq.configure;
@Configuration
public class MyFuncAutoConfiguration {
    @Bean
    @ConditionalOn... // 条件注解,控制是否进行自动配置
    public MyfuncService myFuncService() { return new MyfuncService(); } // 实例化
}
```

### 将自动配置类引入环境:

- SpringBoot 2: 在 resource -> META-INF -> spring.factories 文件中引入
- SpringBoot3: 在 resource -> META-INF -> spring -> org.springframework.boot.autoconfigure.Autoconfiguration.imports 内

```
com.whq.configure.MyFuncAutoConfiguration
```

这样就初步完成了自定义starter的创建，然后mvn install到本地仓库，就可以被其他项目引入了。

**使用自定义Starter**: 在pom.xml中引入，然后通过IoC就能在项目里注入并使用了

```
<dependency>
    <groupId>com.whq</groupId>
    <artifactId>my-func-spring-boot-starter</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

```
@Service
public class xxxService {
    @Autowired
    private MyfuncService myFuncService; // 注入该starter提供的功能类
    public void xxxfunc() { myFuncService.func(); } // 使用
}
```

**通过拦截器完善Starter**: 当前Starter功能仍需要在目标项目中手动调用，可以在Starter中注册拦截器来自动执行

```
// Starter项目interceptor文件夹下
public class MyFuncInterceptor implements HandlerInterceptor {
    @Autowired
    private MyFuncService myFuncService;
    // 通过preHandle拦截，在处理http请求之前自动执行myFunc
    @Override
    public boolean preHandle(HttpServletRequest ..., HttpServletResponse ...) {
        myFuncService.func();
    }
}
```

实现拦截器后需要注册拦截器，并将注册配置写入 META-INF

```
package com.whq.config
@Configuration
public class SpringMvcConfig implements WebMvcConfigurer {
    @Bean
    public MyFuncInterceptor myFuncInterceptor() {return new MyFuncInterceptor(); }
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(myFuncInterceptor()).addPathPatterns("/**");
    }
}
```

增加配置项：除了自动配置之外，也要允许用户通过 application.yml 进行配置：

```
tools:
myfunc:
  config1: simple # 对这个配置项手动配置了simple
```

这就要回到Stater项目中实现 Properties 来支持配置读取

```
package com.whq.properties;
@Data
@ConfigurationProperties("tools.myfunc") // 匹配配置项
@Component // 注入IOC
public class MyFuncProperties {
    private String config1 = "simple"; //默认为simple
}
```

在Starter的功能类里，就可以通过注入 MyFuncProperties 来读取配置项，执行相应的逻辑了

## 总结

SpringBoot核心注解 `SpringBootApplication`，通过 `@EnableAutoConfiguration` 开启自动装配，通过 `SpringFactoriesLoader` 最终加载 `META-INF/spring.factories` 中的自动配置类，实现了自动装配。

自动配置类 `xxxAutoConfiguration`，其实就是通过条件注解 `@ConditionalOnxxx` 按需加载的配置类，想要其生效必须引入 `spring-boot-starter-xxx` 包实现起步依赖。

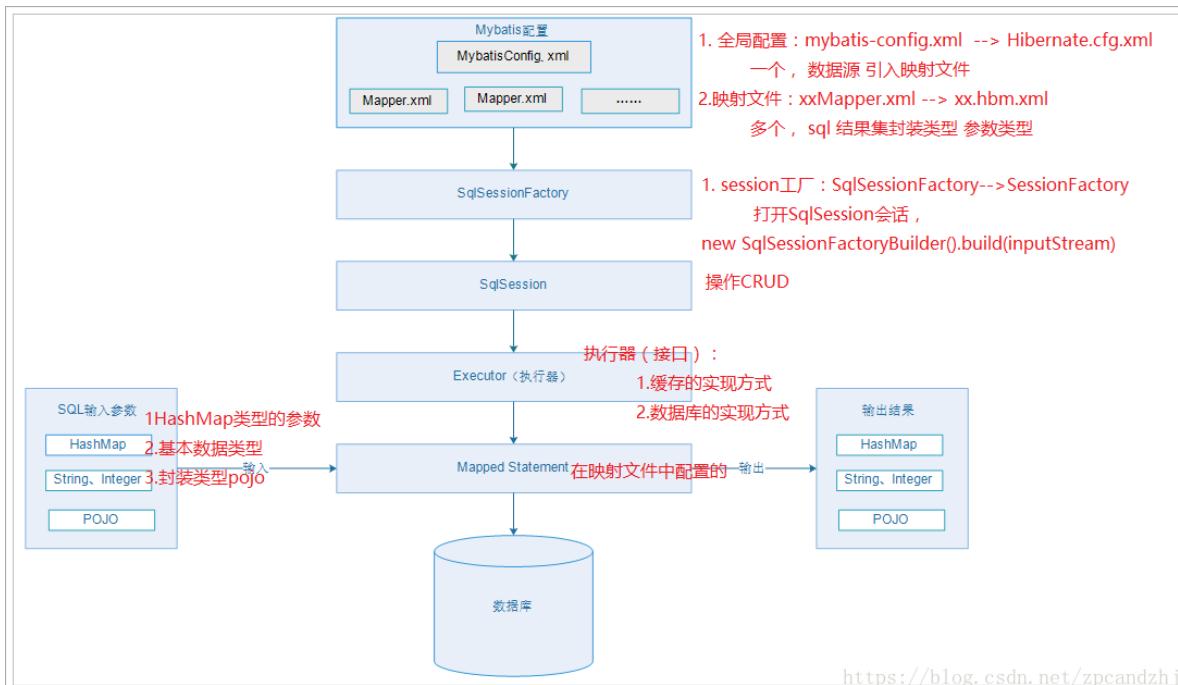
# MyBatis

连接数据库有多种方案和框架，最基础的是Java原生JDBC，每次执行sql都需要开启和关闭连接、连接配置硬编码、SQL字符串和Java耦合，不考虑。下面是一些常用框架：

特性	JPA	Spring Data JPA	MyBatis	MyBatis-Plus
类型	标准 ORM 框架	JPA 的 Spring 封装和扩展	半 ORM 框架	MyBatis 的增强版本
开发复杂度	中	简单	较高	简单
SQL 控制	自动生成 SQL，手动控制较麻烦	自动生成 SQL，但支持自定义查询	完全手写 SQL，灵活性高	提供自动 CRUD，支持手写 SQL
学习曲线	中	简单	较高	简单
与 Spring 集成	需要手动配置	无缝集成	需额外配置	需额外配置
适合场景	复杂的领域模型，标准化要求较高	简化 JPA 开发的场景	数据库复杂，SQL 灵活性要求高	SQL 灵活性要求高，同时提升开发效率

相比较JPA，MyBatis有更灵活的分离的SQL，但需手写；MyBatisPlus是MyBatis的增强版本（完全兼容），在MyBatis的xml配置和注解被注入之后，进一步反射分析实体、注入CRUD方法。

## MyBatis整体架构



## MyBatis+使用文档

[MyBatisPlus官方](#)

[MyBatis官方](#)

[MyBatis](#)：全局配置完成后，通过SqlSessionFactory创建SqlSession实例（Bean），SqlSession使用XML文件映射CRUD操作。

# MyBatis原理与使用

## 为什么说MyBatis是半自动ORM框架

- 全自动：如 Hibernate，SQL可以都由框架自动生成，开发者主要操作实体类。更关注业务逻辑。
- 半自动：需要手写SQL，或是需要用xml/注解配置sql；但框架会自动管理连接、事务、结果集映射等。更灵活高效。

## MyBatis XML映射工作原理

### 解析映射文件

在 XML 映射文件中：

- `<parameterMap>` 标签会被解析为 `ParameterMap` 对象，其每个子元素会被解析为 `ParameterMapping` 对象。
- `<resultMap>` 标签会被解析为 `ResultMap` 对象，其每个子元素会被解析为 `ResultMapping` 对象。
- 每一个 `<select>、<insert>、<update>、<delete>` 标签均会被解析为 `MappedStatement` 对象，标签内的 sql 会被解析为 `BoundSql` 对象。

最终，所有 XML 配置信息都随标签封装到一个All-In-One重量级对象 `Configuration` 内部。

### XML对应DAO接口

XML解析得到的 `MappedStatement` 的 id，与DAO接口（`xxxMapper.java`）的全限定名+方法名相对应。执行接口方法时，通过动态代理转到其对应的 `MappedStatement` 执行相应SQL。

## DAO接口工作原理

### 实现思想：XML映射

MyBatis最佳实践中，通常使用XML映射文件来对应DAO接口（Mapper接口）方法。XML映射文件中有`<select>、<insert>、<update>、<delete>`等标签，它们各自被解析为 `MappedStatement` 对象，代表相应的SQL语句。映射关系如下：

- 接口的全限名，即XML映射文件中的 namespace 值，例如 `com.mybatis3.mappers.StudentDao`。
- 接口的方法名，即XML映射文件中 `MappedStatement` 的 id 值，例如 `findstudentById`。
- 接口方法的参数，即传递给 `MappedStatement` 代表的SQL语句的参数。

`Mapper` 接口没有实现类，当调用接口方法时，接口全限名+方法名拼接字符串作为 key，可唯一定位一个 `MappedStatement`。

### 底层原理：JDK 动态代理

MyBatis运行时，使用 JDK 动态代理为DAO接口生成proxy代理对象，代理对象会拦截接口方法，转而执行对应的 `MappedStatement` 所代表的 sql，然后将 sql 执行结果返回。

DAO(Mapper) -> Proxy -> MappedStatement -> SQL

## DAO接口能重载吗

上述可得，MyBatis 由 DAO 接口方法出发去寻找 XML 映射的 `MappedStatement`，因此同一映射不能有多个。Mybatis 的 Dao 接口可以有多个重载方法，但是多个接口对应的映射必须只有一个，即 XML 内的 `id` 不能重复。

## 不同XML文件内的 id 可以重复吗

如果配置了 `namespace`，那么 `id` 可以重复，否则不行。`namespace+id` 是作为 `Map<String, MappedStatement>` 的 key 使用的，是唯一定位它们的方式。

## sqlSession

sqlSession 方法：

```
selectOne 和 selectList 主要用于查询数据。  
insert、update 和 delete 分别用于插入、更新和删除数据。  
commit 和 rollback 用于事务管理。  
close 用于关闭 SqlSession 释放资源。  
getMapper 用于获取对应的 Mapper 接口代理对象，通常是 MyBatis 的核心功能之一。
```

## 工作例子

- 最佳实践中，`sqlSessionFactory` 作为一个全局的单例 Bean 使用；Mapper 接口要调用一个方法时，才创建一个专门的 `sqlSession` 去执行这个 SQL。
- `SqlSession` 的生命周期应尽可能短，通常在一个请求或事务中创建、关闭。
- `SqlSession` 不是线程安全的，不能在多个线程间共享；每个线程应创建并独占自己的 `SqlSession` 实例。

配置接口和映射：

```
public interface UserMapper {  
    User selectUserById(Integer id); // 查询单条记录的方法  
    List<User> selectUsersByAge(Integer age); // 查询多条记录的方法  
}
```

```
<mapper namespace="com.example.mapper.UserMapper">  
    <select id="selectUserById" resultType="com.example.model.User"> <!-- 查询单个-->  
        SELECT * FROM users WHERE id = #{id}  
    </select>  
  
    <select id="selectUsersByAge" resultType="com.example.model.User"> <!-- 查询多个-->  
        SELECT * FROM users WHERE age = #{age}  
    </select>  
</mapper>
```

<mapper> 中还可以指定 `parameterMap`、`resultMap` 等

使用：

```
public class UserService {  
    private SqlSessionFactory sqlSessionFactory;  
    public UserService(SqlSessionFactory sqlSessionFactory) {  
        this.sqlSessionFactory = sqlSessionFactory;  
    }  
  
    public User getUserById(int id) {  
        try (SqlSession session = sqlSessionFactory.openSession()) {  
            UserMapper userMapper = session.getMapper(UserMapper.class); // 获取  
Mapper  
            return userMapper.selectUserById(id); // sqlSession调用selectOne  
        }  
    }  
  
    public List<User> getUsersByAge(int age) {  
        try (SqlSession session = sqlSessionFactory.openSession()) {  
            UserMapper userMapper = session.getMapper(UserMapper.class);  
            return userMapper.selectUsersByAge(age); // sqlSession调用selectList  
        }  
    }  
}
```

sqlSession会根据接口方法和XML配置，自动选取对应的对应的底层方法执行。（自动选择 selectOne 或 selectList）

## \${} 和 \${} 的区别

- `\${}`：使用 JDBC 的 PreparedStatement 参数绑定机制，先预编译SQL将 `\${}` 替换为 `?`，再在执行前将参数值绑定到 `?`。

可以防止SQL注入；Java类型自动转换为SQL类型。

主要用于传递动态参数：

```
SELECT * FROM users WHERE id = #{id};
```

- `\${}`：原文本作为字符串直接替换，会直接将参数值拼接到 SQL 中。

不能防止SQL注入，不支持类型自动转换。

主要用于动态 SQL 结构（如表名、列名等）：

```
SELECT * FROM ${tableName} WHERE ${column} = #{value};
```

## MyBatis分页

## SQL手动分页

在SQL语句中添加 `LIMIT` 和 `OFFSET` 语法、动态传参。物理分页（在数据库层面分页，只返回目标数据）。

```
// xxxMapper.xml
<select id="getPaginatedData" resultType="YourEntity">
    SELECT * FROM table_name
    WHERE conditions
    ORDER BY column_name
    LIMIT #{limit} OFFSET #{offset}
</select>
```

```
// xxxMapper.java
int limit = 10; // 每页大小
int offset = (page - 1) * limit; // 偏移量
List<YourEntity> data = mapper.getPaginatedData(limit, offset);
```

## MyBatis RowBounds

MyBatis 使用 RowBounds 对象进行分页，它是针对 ResultSet 结果集执行的内存分页（先把所有数据加载到内存再分页）。

```
// xxxMapper.xml
<select id="getAllData" resultType="YourEntity">
    SELECT * FROM your_table
</select>
```

```
// 构造 RowBounds 对象
int offset = 20; // 从第21条记录开始
int limit = 10; // 每页10条记录
RowBounds rowBounds = new RowBounds(offset, limit);
// 使用 sqlSession 查询
try (SqlSession sqlSession = sqlSessionFactory.openSession()) {
    List<YourEntity> data = sqlSession.selectList("namespace.getAllData", null,
rowBounds);
}
```

## 分页插件PageHelper

(MyBatisPlus内置了一个分页插件PaginationHelper)

原理：使用MyBatis的插件接口自定义了一个插件，拦截待执行sql重写为分页sql。实现可重用的物理分页。

使用：引入依赖、配置拦截器、在代码中使用

## MyBatis插件原理

MyBatis 仅可以编写针对 `ParameterHandler`、`ResultSetHandler`、`StatementHandler`、`Executor` 这 4 种接口的插件。插件原理基于 JDK 的动态代理（以及责任链模式，一个接口存在多个插件依次执行），为需要拦截的接口生成代理对象，每当执行这 4 种接口对象的方法时，就会进入拦截方法。

1. `Executor`：负责 SQL 的执行、事务管理等。
2. `StatementHandler`：负责 SQL 语句的生成和参数绑定。
3. `ParameterHandler`：负责处理 SQL 参数。
4. `ResultSetHandler`：负责处理查询结果集。

实现自定义插件：实现 MyBatis 的 `Interceptor` 接口、复写 `intercept()` 方法；然后给插件编写 `@Signature` 注解，指定要拦截哪一个接口的哪些方法。在配置文件中加上插件配置即可使用。

```
@Intercepts({  
    @Signature(type = Executor.class, method = "update", args =  
    {MappedStatement.class, Object.class}),  
})
```

```
// 分页插件  
@Intercepts({  
    @Signature(type = StatementHandler.class, method = "prepare", args =  
    {Connection.class, Integer.class})  
})  
public class PaginationPlugin implements Interceptor {}
```

## MyBatis关联查询

有两种实现方式：

1. 另外单独发送一个 sql 去查询关联对象，获取到了之后在代码中赋给主对象，然后返回主对象。
2. 使用嵌套查询 `join`，一部分列是 A 对象的属性值，另外一部分列是关联对象 B 的属性值，好处是只发一个 sql 查询，就可以把主对象和其关联对象查出来。

但是 `join` 查询到的结果中往往有很多重复的主对象，这时候 MyBatis 会做逻辑去重，通过

`<ResultMap>` 的 `<id>` 子标签

## MyBatis执行器

MyBatis 执行器 `Executor` 负责与数据库交互，控制 SQL 的执行、查询结果的缓存和事务管理。

MyBatis 有三种基本的 `Executor` 执行器：

- `SimpleExecutor`：每次执行 sql 都创建一个新的 Statement 对象，用完立刻关闭 Statement 对象。
- `ReuseExecutor`：每次执行 sql 时，以 sql 作为 key 查找 Statement 对象，存在就使用，不存在就创建；用完后不关闭，而是放置于 `Map<String, Statement>` 内，供下一次使用。简言之，就是复用 Statement 对象。
- `BatchExecutor` 批：执行sql写操作（不支持select，批处理是为了优化批量写），`addBatch()` 将所有 Statement 对象都添加到批处理缓存、延迟统一执行（其实也是逐一执行）

`executeBatch()`。与 JDBC 批处理相同，JDBC 不支持批量读`select`操作。

作用范围：`Executor` 的这些特点，都严格限制在 `SqlSession` 生命周期范围内。

## 如何指定执行器？

MyBatis 配置文件中指定默认执行器；或手动给 `sqlSessionFactory` 创建 `SqlSession` 的方法传递 `ExecutorType` 类型参数。

# MyBatis缓存

MyBatis缓存适用于读多写少的数据（写操作需刷新缓存）

- **一级缓存：**`SqlSession`级别的缓存，默认开启、不可关闭。同一个`SqlSession`中，执行相同查询时会从缓存取数据。存储在该`SqlSession`对象内部，不可与其他`SqlSession`共享。  
清空一级缓存：执行写操作sql；`sqlsession.clearCache()`；关闭当前`SqlSession`。
- **二级缓存：**`Mapper`级别的缓存，需要显式开启，可在不同`SqlSession`之间共享。默认存储在 MyBatis 内部，也可以放在第三方如 Redis 中。

使用二级缓存时，查询结果会被序列化存储，因此返回的实体类需要实现  
`java.io.Serializable` 序列化接口。

配置：

```
// 全局配置xml
<settings>
    <setting name="cacheEnabled" value="true"/>
</settings>
```

```
// Mapper xml
<cache
    eviction="LRU" // 缓存回收策略
    flushInterval="60000" // 刷新时间间隔ms
    size="512" // 大小，默认1024对象
    readOnly="true" // 只读，缓存数据不可修改
/>
```

二级缓存要考虑数据一致性问题。

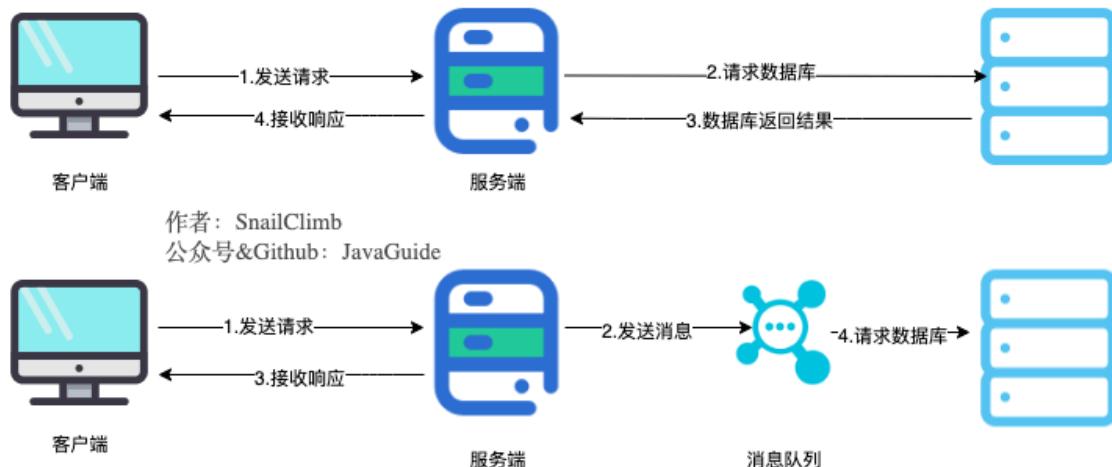
# 消息队列

## 基础知识

# 消息队列特点

功能和好处

## 1. 异步处理



用户请求通过MQ被**异步**处理，填入MQ后就立即返回一个结果（比如已提交/pending）给用户，提高体验。

## 2. 削峰/限流

先将**短时间高并发**产生的事务消息存储在消息队列中，然后后端服务再慢慢根据自己的能力去消费这些消息，这样就避免直接把后端服务打垮掉。

## 3. 降低系统耦合度

**模块之间不是直接调用**，那么新增模块或者修改模块就对其他模块影响较小，提高了系统的扩展性。MQ使用发布订阅模式时，出现新增服务、想要消费消息，只需要订阅该Topic即可，不会对原来的系统产生影响。

4. **顺序消息**: 保证消息被处理的顺序。

5. **延时/定时消息**: 可以指定一个时间让消息被消费。

此外，为了避免MQ服务器宕机导致服务丢失，**进入MQ的消息仍会存储在Producer服务器上，只有该消息真正被Consumer处理后才会删除**；当MQ宕机，Producer会选择分布式MQ集群的其他MQ重新发送消息。

带来的问题

1. 降低系统可用性：需要额外考虑**MQ宕机**的情况
2. 提升系统复杂度：需要额外考虑**重复消费、消息顺序、消息丢失**的情况
3. 数据一致性：消息的**异步性**可能导致Producer和Consumer之间不一致（比如扣费了但是订单还在pending）；**消息丢失**（扣费了但商家没有收到）。

# JMS

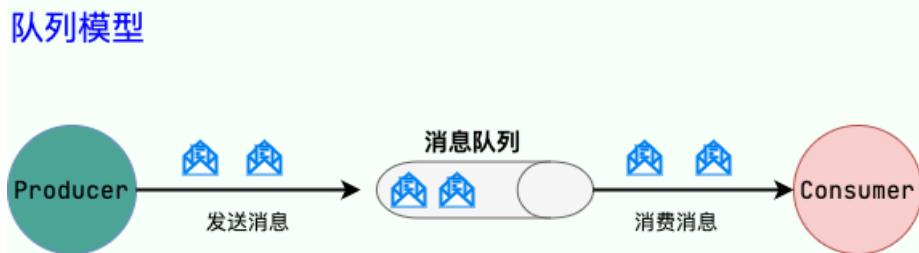
JAVA Message Service是Java的消息服务，支持了Java的异步通信。**JMS API 是一个消息服务的规范**。

## JMS五种消息数据类型

1. StreamMessage：Java 原始值的数据流
2. MapMessage：一套键值对集合
3. TextMessage：一个字符串对象
4. ObjectMessage：一个序列化的 Java 对象
5. BytesMessage：1Byte的数据流

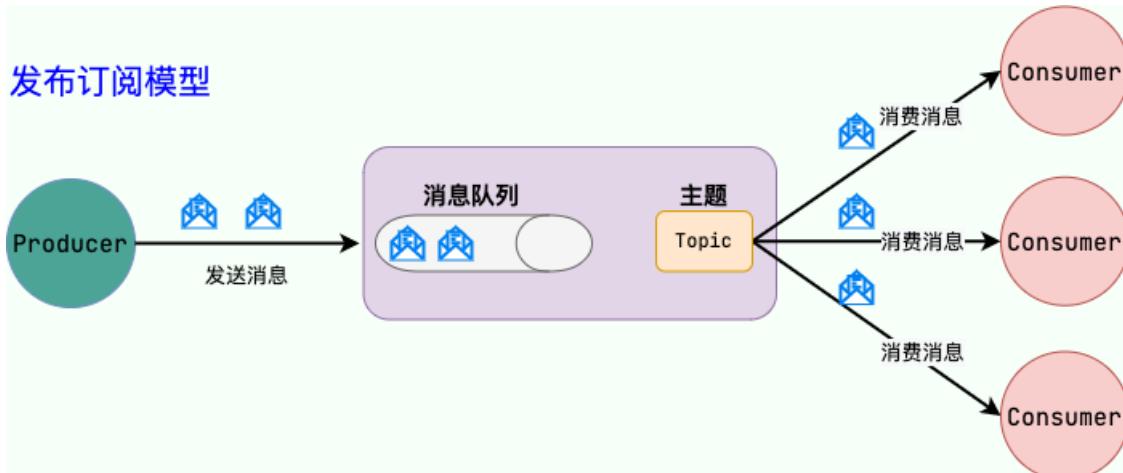
## JMS两种消息模型

### 1. 点对点 (P2P) 模型



使用**队列Queue**作为消息通信载体；满足**生产者与消费者模式**，一条消息只能被一个消费者使用，未被消费的消息在队列中保留直到被消费或超时。（但不是说MQ只能1-1，多个消费者可以轮流消费、但不能消费同一个消息）

### 2. 发布订阅 (Pub-sub) 模型



使用**主题Topic**作为消息通信载体，类似于**广播模式**；发布者发布一条消息，该消息通过主题传递给所有的订阅者。这种模式使用较多，能给系统解耦，出现新服务时只需要订阅消息即可、不会影响原系统。

## AMQP

RabbitMQ实现了AMQP，其基础架构与AMQP一致。

Advanced Message Queuing Protocol**高级消息队列协议**，属于应用层协议，是一个提供统一消息服务的开放标准，兼容JMS。**RabbitMQ**就是基于**AMQP**协议实现的。

## AMQP核心概念

### AMQP协议三层架构：

1. **Module Layer**: 协议最高层，定义了一些客户端调用的命令，客户端可以用这些命令实现自己的业务逻辑。
2. **Session Layer**: 中间层，负责将客户端命令发送给服务器、再将服务端应答返回客户端，提供可靠性同步机制和错误处理。
3. **Transport Layer**: 最底层，主要传输二进制数据流，提供帧的处理、信道复用、错误检测和数据表示等。

### AMQP模型三大组件：

1. **交换机 Exchange**: 用于把接收到的消息，根据路由规则分发到对应Queue。
2. **队列 Queue**: 用来存储消息的数据结构，位于硬盘或内存中。
3. **绑定 Binding**: 一套规则，交换机根据规则决定应该将消息分发给哪个Queue。

## JSM vs AMQP

- **定义层级**: JMS属于Java API规范，能与Java应用无缝衔接，但兼容性差；AMQP是更底层的协议，天然支持跨平台、跨语言。
- **消息数据类型**: JMS支持复杂5种类型；AMQP **仅支持 byte[]** 类型，复杂对象需要**序列化**。
- **工作模型**: JMS支持P2P和Pub-sub两种；AMQP通过Exchange路由算法，支持多种复杂路由机制来传递消息（见RabbitMQ）

## RPC vs MQ

对比	RPC	MQ
用途	简化两个服务之间远程调用对方方法的过程，使其像调用本地方法	分发消息、异步处理、削峰填谷
通信方式	双向直接网络通信	单向通信、引入中间件载体
请求时效性	立即处理	异步处理
存储架构	不需要在内部存储数据（双向直接通信）	需要存储消息

## MQ技术选型

1. **Kafka**: LinkedIn开源的分布式日志系统，可用于消息队列、可持久化消息流、流式处理平台。
2. **RocketMQ**: 阿里开源的云原生实时数据处理平台，借鉴Kafka，云原生、极简架构、丰富拓展生态。
3. **RabbitMQ**: Erlang语言实现AMQP的消息中间件，高性能、灵活路由、跨语言、易用控制台UI、支持多协议。

对比方向	内容
开发语言	Kafka和RocketMQ由Java开发， RabbitMQ由Erlang开发
吞吐量	Kafka和RocketMQ吞吐量达十万百万， RabbitMQ稍弱只有万级
并发性能/ 延时	RabbitMQ基于Erlang开发， 并发性能极强， 延时达到微秒级， 其他只能毫秒级
一致性	RocketMQ支持强一致性
可用性/架 构	都是高可用。 Kafka和RocketMQ是分布式架构， RabbitMQ核心是主从架构（但也支持分布式集群）
使用场景	Kafka适用于大数据、流处理、高吞吐量； RocketMQ适用于金融支付、数据同步、分布式事务； RabbitMQ适用于企业级灵活路由和多协议。

追求极致吞吐 + 低成本 → Kafka 

需要事务支持 + 可靠性 → RocketMQ 

需要强一致性 + 复杂路由 → RabbitMQ 

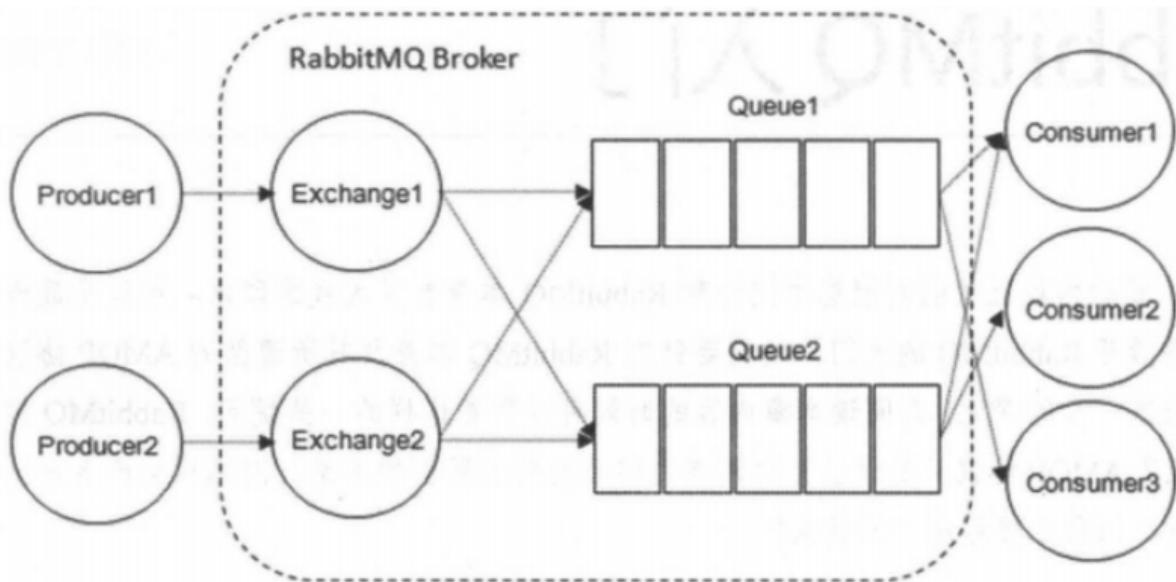
## RabbitMQ

### 特性

- **灵活路由**: 在消息进入队列之前，通过交换器来路由消息。 RabbitMQ内置交换器实现了一些典型的路由功能。
- **高可用**: 支持镜像队列的主从架构
- **多协议**: 除了原生支持 AMQP 协议，还支持 STOMP， MQTT 等多种消息中间件协议
- **多语言**: 支持几乎所有常用语言
- **易用性**: 自带一个控制台UI，可以监控和管理消息、集群中的节点等

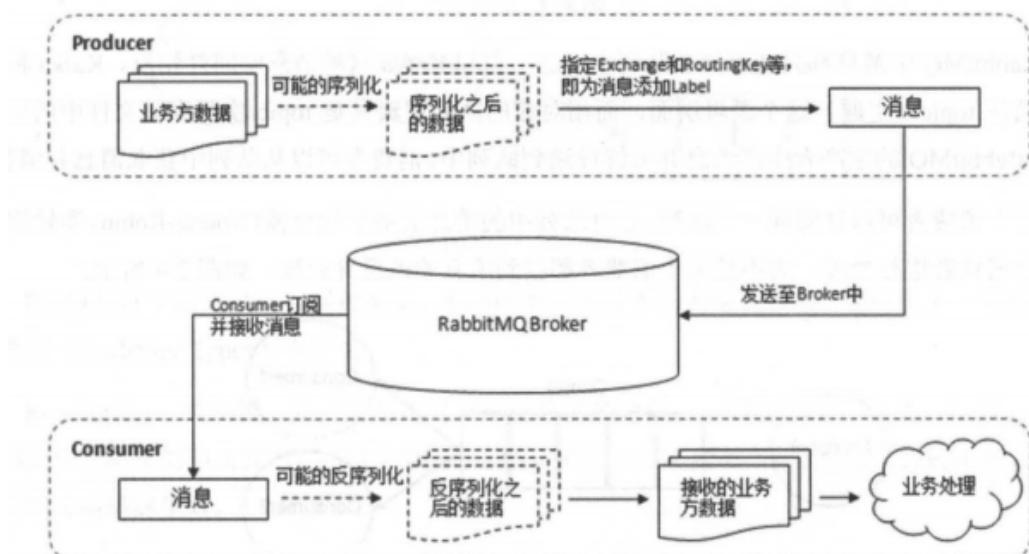
### 核心概念与架构

RabbitMQ整体上是一个生产者-消费者模型，可以接收、存储、转发消息。它更像是一个**交换机模型**。



- **Message消息**: 一般由消息头**Label**和消息体**Payload**组成。消息头包含 `routingKey` (路由键)、`priority` (优先级)、`deliveryMode` (可能要持久化)。消费时会丢弃Label、只消费Payload。
- **Exchange交换机**: 消息路由的核心组件，负责接收Producer发送的Message，根据 `routingKey` 和交换机类型(路由策略)，将其分发到对应的Queue中。若找不到符合的Queue则返还给Producer或丢弃。  
RabbitMQ用 `bindingKey` 将Exchange和Queue绑定 (可以是多对多关系)，交换机实际上是由多个 `bindingKey` 组成的路由表。  
Exchange有4种类型: `direct`(默认)、`fanout`、`topic`、`header`，对应不同的路由策略 (根据 `routingKey`, `bindingKey` 处理) 。
- **Queue消息队列**: RabbitMQ中消息存储和消费的核心单元，消息只存储在Queue中，消费也是直接面向Queue的。消息消费之后通常从Queue中删除以防止重复消费。若多个消费者订阅同一Queue，通过轮询平摊消息 (多轮遍历消费者-确认)。  
*Kafka不直接提供队列的概念，消息存储和消费以Topic为核心。Topic是逻辑概念，每个Topic有多个物理分区Partition用于实际存储；消费者订阅Topic并按Partition读取。消费后让保留在Partition以重复消费。*
- **Broker服务实例**: 一个RabbitMQ Broker是一个 RabbitMQ 服务节点实例，通常也是一个 RabbitMQ服务器。

一个典型业务处理流程中的RabbitMQ Broker、前后处理：



# 交换机类型与工作模式

默认情况下，RabbitMQ的设计是不能重复消费，即一条消息只能被一个消费者消费（一个Queue轮询分发给多个消费者，是为了提升处理速度）；若想要“重复消费”（如Pub-sub），更好的实践是分发到多个Queue，这样每个Queue都持有一份独立副本。

## Exchange Type 交换机类型

### 1. fanout

相当于广播模式，会把消息发到所有与当前交换机绑定的Queue中去，无需路由判断、速度最快

### 2. direct

根据消息的Label，把它发送到 routingKey 与 bindingKey 完全匹配的Queue。direct模式一般配合不同优先级的Queue，指派更多资源处理高优先级队列。

如下图，如果消息 `routingKey = 'warning'`，就会被发到Queue1和Queue2：

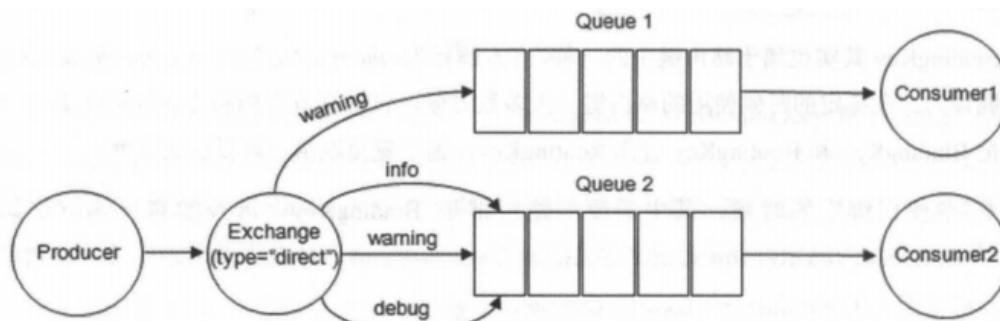
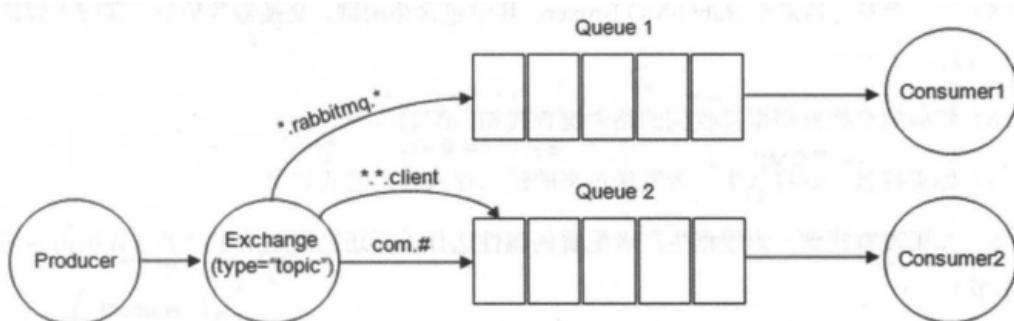


图 2-7 direct 类型的交换器

### 3. topic

也是将 `routingKey` 去匹配 `bindingKey`，但是可以动态模糊匹配：bindingKey 可以用 \* 和 # 来模糊匹配。此外，两个key的格式都是由 . 分割的字符串。

如下图，如果消息 `routingKey = 'com.rabbitmq.client'`，就会被发送到Queue1和Queue2：



### 4. headers (不推荐)

根据消息Payload中的headers属性去匹配。由于要获取消息内容，性能很差、不实用。

## RabbitMQ的工作模式

根据上述几种类型的交换机，RabbitMQ实现了灵活的路由，因此支持多种工作模式。

1. 简单模式 SimpleQueue / P2P：direct默认交换机，1生产者-1队列-1消费者
2. 任务模式 WorkQueue / TaskQueue：direct默认交换机，1生产者-1队列-n消费者，用于任务分发、负载均衡
3. 发布订阅模式 Pub-Sub：fanout交换机，多队列，用于广播消息

4. **路由模式 Routing**: direct交换机, 多队列, 用于消息分类投递
5. **主题模式 Topic**: topic交换机, 动态路由多队列, 用于复杂消息按模块分类投递

## 死信 DL

一个消息变成死信 (Dead-Letter / Dead-Message) 的情况:

1. 消息被显式拒绝 (`basicNack` / `basicRej`) , 且未配置重新入队 (`requeue = false`)
2. 消息TTL过期 (Time-to-Live, 消息过期时间, 手动设置)
3. Queue容量已满

当消息在一个队列中变成死信, 它将进入**死信交换机DLX** (DL-Exchange), DLX绑定的Queue就是**死信队列**。

## 死信与延迟队列

**延迟队列delay-queue**: 存储**延迟消息delay-message**的队列。延迟消息被Producer发出后, 等待一段时间再给Consumer消费。

AMQP协议和RabbitMQ并不原生支持延迟队列。高版本的RabbitMQ通过插件提供了这个功能。

但是RabbitMQ可以通过**消息TTL + 死信队列** 来实现延迟队列。

## RabbitMQ通信方式: Channel

TCP链接存在性能瓶颈: 创建销毁开销大、链接数受系统限制。因此, RabbitMQ使用**信道Channel**与Producer/Consumer通信。

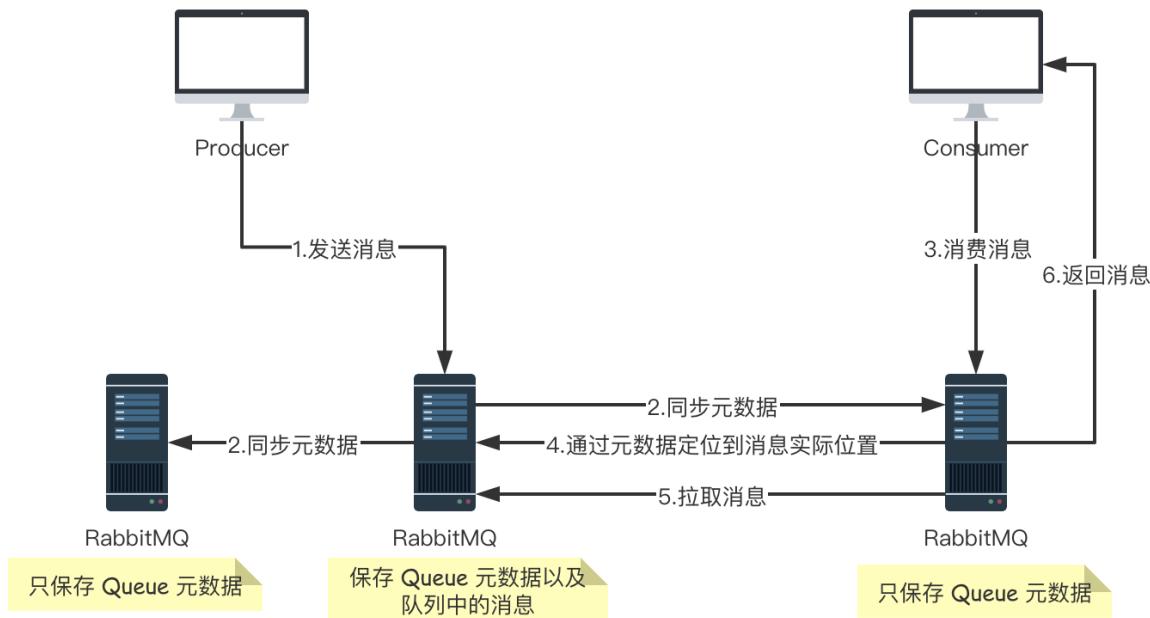
信道是建立在TCP链接上的虚拟链接, 无数量限制, 每个信道拥有唯一id, 对应RabbitMQ内的一个私有线程, 该TCP被多线程共享。

## RabbitMQ实现高可用: 集群

RabbitMQ有三种模式: 单机, 普通集群, 镜像集群。

### 普通集群模式

- **元数据同步**: 一个队列的元数据 (如队列名、绑定关系) 在集群中所有节点同步。
- **主节点唯一**: 该队列以及其消息都只存在一个节点上, 即主节点**MasterQueue**; 主节点宕机, 则队列失效、消息丢失。
- **节点转发**: 如果要从其他节点消费消息, 需要从主节点转发过来, 增加网络延迟。



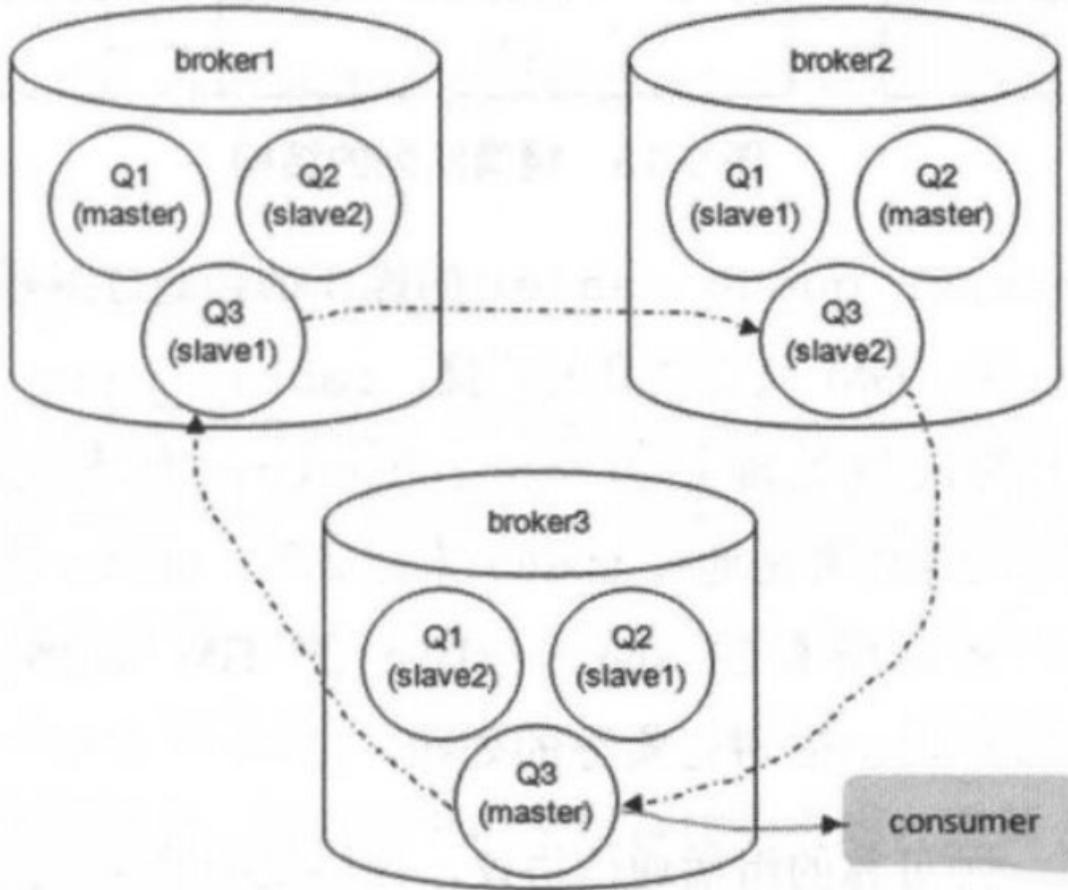
普通集群模式的核心是让多个节点服务某个队列的读写，并不是高可用，只是一个简单高效的提高吞吐量的方案。

## 镜像集群模式

- **元数据+消息同步**: 队列元数据和消息会同步到集群所有镜像节点。
- **主节点读写**: 主节点**MasterQueue**负责该队列所有的读写，镜像节点收到的操作请求都会转发到主节点上。
- **镜像备份**: 镜像节点**MirrorQueue**只作为该队列的同步备份，主节点挂了则选举最老的镜像进行主备切换。

镜像集群是RabbitMQ的高可用模式，可靠性强，但饱和的镜像同步导致了较大的网络开销和存储开销。

**镜像集群负载平衡**: 依赖“仅主节点读写”的特性，将不同队列主节点分布在不同的RabbitMQ Broker实例上。



如上图，集群中的每个Broker都包含1个队列的master和其他2个队列的slave(mirror)，则Q1的负载集中于broker1，Q2的负载集中于 broker2，Q3的负载集中于broker3。master均匀散落即可很大程度保证负载均衡。

## 如何保证消息传输的可靠性

1. **Producer - RabbitMQ过程丢失**: 事务机制 或 Confirm机制 (在Channel层面开启，二者互斥)
  - **事务机制**: Producer在Channel上开启同步事务，提交或回滚来保证确认消息。
  - **Confirm机制**: 异步确认，Producer无需同步等待，通过监听RabbitMQ的异步回调来确认，性能高。
2. **RabbitMQ内丢失**: 持久化、集群、普通模式、镜像模式
3. **RabbitMQ - Consumer过程丢失**: basicAsk机制、死信DLX、消息补偿
  - **确认机制**: `autoAck`自动确认 (发送即确认、有丢失风险) ; `basicAck`手动确认 (需要Consumer显式确认)
  - **消息补偿机制**:
    - **重新入队**: 通过 `basicNack` 或 `basicRej` (拒绝单条消息) 将消息重新入队，可能导致无限重试拖累性能
    - **延迟重试**: 配置延迟队列 (TTL + 死信队列)
    - **有限重试**: 通过消息Payload中的headers属性，记录重试次数
    - **失败记录**: 记录到日志或持久化；配置失败队列

# 设计模式

<https://refactoringguru.cn/design-patterns/catalog>

三种工厂模式：

1. 简单工厂模式：一个工厂具体类，在其内根据传入参数，直接实例化产品。**扩展性差、违法开闭原则。**
2. 工厂方法模式：一个工厂接口，多个子工厂类实现该接口，在子工厂中实例化产品。
3. 抽象工厂模式：n个工厂接口，n\*m个子工厂类实现接口，在子工厂中实例化产品；一个组合工厂类实现组合。**多产品族场景。**

# OS

## 用户态和内核态

- **用户态**：应用程序、库函数、系统调用接口、辅助工具（文本编辑器）。
- **内核态**：进程管理、内存管理、文件管理、设备管理、网络通信。用于运行一些危险的**特权指令**（内存分配、设置时钟、IO）。

分离用户态和内核态除了提升安全性稳定性，也避免了大量进程密集竞争系统资源。

## 用户态切换到内核态

1. **系统调用**：用户态主动调用系统内核提供的功能接口执行 `syscall`。实际上是一种特殊中断 Trap。
2. **中断**：CPU因外部或内部事件打断当前指令流，转去执行特定的处理程序，之后再恢复原任务。
  - 外部中断（硬件中断）：时钟中断；IO中断；网络中断
  - 内部中断（软件中断）：系统调用陷入内核；异常处理
3. **异常**：CPU遇到特殊情况无法继续执行（非法操作）

# 进程与线程

## 线程间通信

1. **互斥锁Mutex**：如Java的synchronized
2. **读写锁ReadWrite Lock**：运行多线程共同读、互斥写
3. **信号量Semaphore**
4. **事件驱动Event**：Wait/Notify，通过事件通知的方式来保持多线程同步
5. **屏障Barrier**：屏障是一种同步原语，用于等待多个线程到达某个点再一起继续执行。如Java的CyclicBarrier

## 进程间通信

1. **匿名管道Pipe**：父子进程、兄弟进程之间通信
2. **有名管道Named Pipe**：本机任意两个进程之间通信，FIFO
3. **共享内存Shared Memory**：划出一块内存，给多进程共同访问，需要依赖互斥同步操作
4. **信号Signal**：事件通知
5. **消息队列MQ**：消息构成链表、存放在内存中

# 进程调度算法

1. **FCFS(FIFO)**: First Come First Serve 先到先服务
2. **SJF**: Short Job First 短作业优先
3. **RR**: Round-Robin 时间片轮转
4. **Priority**: 优先级调度
5. **MFQ**: Multi-Level Feedback Queue 多级反馈队列，动态调整进程优先级。
  - 维护多个不同优先级的队列，通常高优先级队列放短作业、短时间片，低优先级队列放长作业、长时间片
  - 抢占式执行高优先级队列任务
  - 新进程放入高优先级，快速响应
  - 进程执行时间较长，优先级衰减，防止占用
  - 进程等待时间较长，优先级升高，防止饥饿

## 上下文 context

CPU 执行某个线程或进程时的所有必要状态信息，包括寄存器、程序计数器、栈指针、内存页表等。当发生线程切换时，操作系统会保存当前线程的上下文，然后恢复新线程的上下文，以便新线程可以从上次执行的地方继续运行。

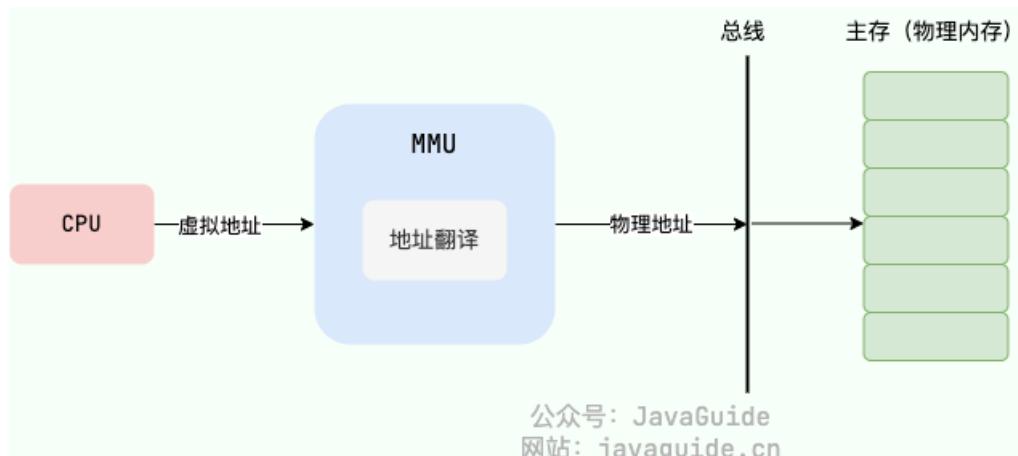
**ThreadContext** 概念在开发工具级别的实现：SpringSecurity 的 SecurityContextHolder；Java 并发的 threadLocal。

## 虚拟内存

- 隔离进程、提高安全性：每个进程都有一块对应的虚拟空间，彼此隔离
- 简化内存管理：从程序视角，拥有了属于自己的一块空间，不需要和物理地址打交道
- 更大的可使用空间：从程序视角，可以利用比实际物理内存更大的空间，因为磁盘也可以算进去

**物理地址** 是实际的物理内存中的地址（寄存器），**虚拟地址** 则是提供给程序的地址。

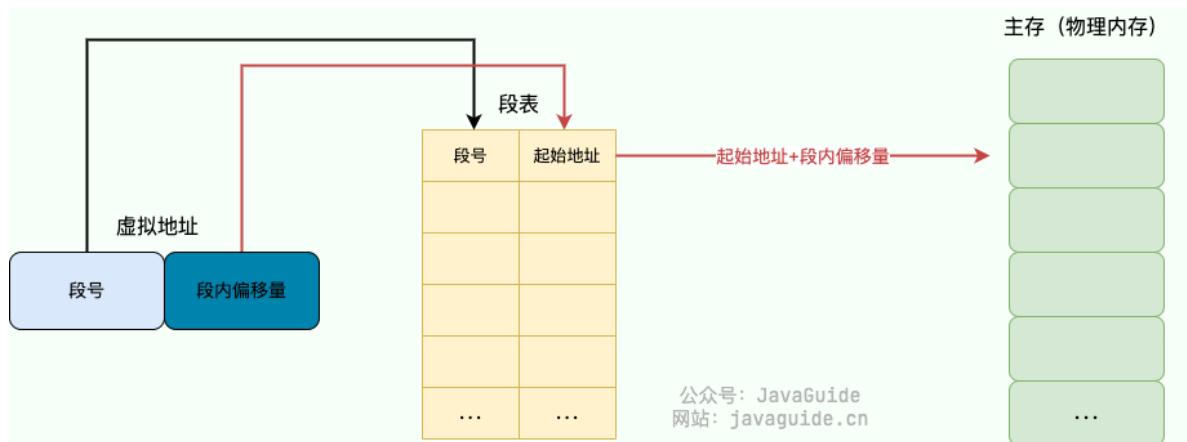
操作系统一般通过 CPU 中的一个重要组件 **MMU(Memory Management Unit, 内存管理单元)** 将虚拟地址转换为物理地址，这个过程被称为 **地址翻译/地址转换**。



映射机制：分段（连续），分页（离散），段页结合

# 分段映射

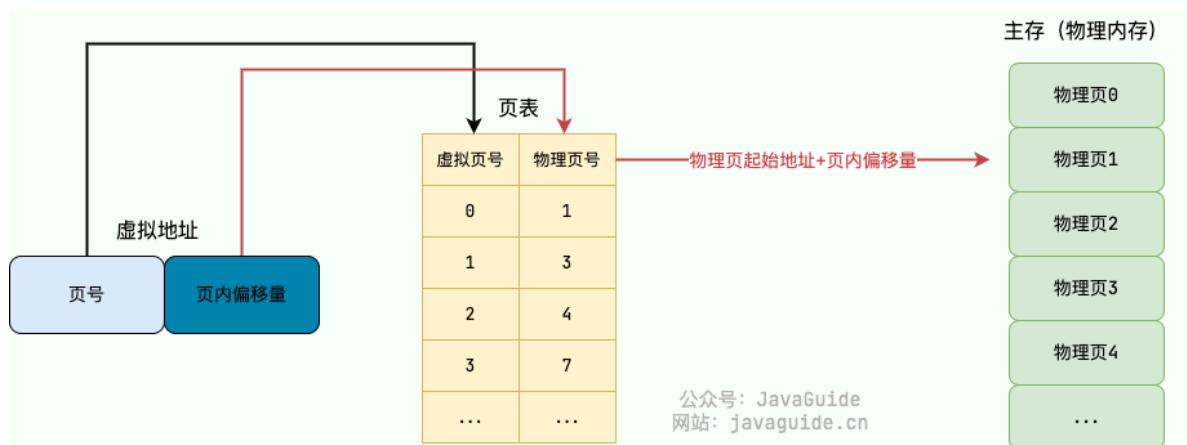
通过段表存储【段号，起始地址】，虚拟地址开头由【段号，段内偏移量】构成，计算物理地址的段位置。



段是不同长的，分段的缺点在于容易留下内存碎片、降低物理内存利用率。

# 分页映射

虚拟内存和物理内存都划为连续等长的页，通过页表【虚拟页号，物理页地址】进行映射。



通过段表，任意离散虚拟页可以被映射为任意离散物理页，解决了碎片问题。

**多级页表**：32位下一页大小4KB，那么页表项有 $2^{20}$ 个，单级页表有4MB，加载页表开销大。通过多级页表，降低了加载页表的空间开销。

**TLB快表**：相当于页表的高速缓存，根据局部性原理，缓存近期查询的虚拟页-物理页映射关系。

**换页（页面置换）**：虚拟页可能非常多，当物理页已经装不下时，需要把一部分页换进磁盘。

- **OPT**：最佳页面置换算法，但是实现很难，一般用作理论最优的标杆来评价其他算法
- **FIFO**：最简单，性能差
- **LRU**：Least Recently Used 最久未使用
- **LFU**：Least Frequently Used 最少使用

**LRU注重时间局部性，LFU注重频率局部性。对于秒杀场景，用LRU更合适，因为是短期热点+数据快速变化。**

**LRU缓存缺点**：缓存抖动（内存不足则频繁替换）；缓存污染（短期热点>长期热点）；缺乏频率考量等等

## 硬链接和软链接

---

- **硬链接**: 指向 **相同数据块 (inode)** 的 **多个不同文件名**, 它们共享相同的 inode 号。
  - 硬链接和原文件没有区别, 相当于一个文件的不同名称、彼此不影响
  - 一个文件的所有硬链接都被删除了, 该文件数据块inode才被释放
  - 不能跨文件系统, 因为 inode 号在不同的文件系统中不通用
  - 不能对目录建立硬链接, 防止循环引用、父目录 ... 模糊
- **软链接**: 指向**原文件路径的快捷方式**, 是一个独立的文件、本身有一个不同的 inode号
  - 可以跨文件系统、可以指向目录
  - 删除原文件后, 软连接仍存在, 但变成了无效链接

## 磁盘调度算法

---

1. FCFS: FIFO, 最容易饥饿
2. SSTF: shortest search time first, 最短寻道时间优先, 可能造成饥饿
3. SCAN / LOOK: SCAN就是电梯算法, 一头出发扫到磁盘尽头再返回; LOOK改进SCAN, 该方向没有寻道需求了就直接返回
4. C-SCAN / C-LOOK: C-SCAN是SCAN变体, 一头出发扫到尽头, 然后重新从起始点出发扫; C-LOOK改进类似

## 计网

---

### 网络分层模型

---

#### OSI七层模型

1. **物理层**: 负责物理介质上的透明的比特流传输, 如电信号、光信号、无线信号。 **比特Bit**
2. **数据链路层**: 将数据封装成帧, 提供介质访问控制MAC和错误检测。 **帧Frame**
3. **网络层**: IP 寻址、路由选择, 决定数据跨网际传输的路径。 **分组Packet**
4. **传输层**: 端到端数据传输通信。 **段Segment(TCP) / 数据报Datagram(UDP)**
5. **会话层**: 管理会话 (连接) 的建立、维护和终止。
6. **表示层**: 负责数据格式转换、加密、解密、压缩等。
7. **应用层**: 提供用户直接使用的网络服务, 如 HTTP、FTP、SMTP 等。

**TCP/IP四层模型**: 网络接口层 (物理层+数据链路层) ; 网络层; 传输层; 应用层 (会话层+表示层+应用层)

#### 应用层协议

- **HTTP**: 基于TCP, 传输超文本和多媒体内容, 用于Web通信。80端口

- HTTP1.1：长连接；请求头 `cache-control` 增强缓存处理(HTTP1.0只有 `Expires`)；增加 `Host` 头；增加状态码
- HTTP2.0：多路复用同时传输多个请求；二进制帧(HTTP1用文本格式)；响应可以携带服务器推送的其他资源；加密
- HTTP3.0：不使用TCP，而是基于UDP升级的QUIC；对整个数据包进行加密包括头部
- HTTPS：HTTP1.x可选，使用SSL/TLS进行加密认证，对称加密。端口443
- FTP：基于TCP，传输文件。不安全、非加密传输
- SSH：基于TCP，通过加密和认证机制实现安全传输
- DNS：基于UDP，域名解析

## 传输层协议

- TCP：面向连接，可靠传输
- UDP：无连接，尽最大可能提供传输，简单高效

## 网络层协议

- IP：定义数据包格式、进行路由寻址。IPv4、IPv6
- ARP：Address Resolution Protocol地址解析协议，解决 IP虚拟地址 - MAC物理地址 转换问题
- ICMP：用于传输网络状态、进行网络诊断。Ping使用ICMP测试网络连通性
- NAT：用于内部网和外部网的地址转换。局域网在广域网下共用一个IP，NAT用于进入该局域网后寻址到各主机
- OSPF、RIP、BGP：几种路由选择协议。路由器之间寻址广泛使用OSPF

# URL访问网页的过程

---

1. 浏览器根据输入的URL，通过DNS获取IP地址
  - DNS本地服务器，迭代查询：根服务器；顶级域名服务器(.cn)；权威域名服务器(buaa.edu)
2. 浏览器根据IP地址和端口号，向目标服务器发起TCP连接请求
  - 服务器同意并建立起TCP连接
3. 浏览器在TCP连接上，向服务器发送HTTP请求，请求获取网页内容
  - 服务器处理HTTP请求并返回响应
4. 浏览器拿到HTTP响应后，解析并渲染HTML代码，根据其中嵌入的多媒体资源URL再次向服务器发起请求，直到完全加载。
  - 服务器返回相应的多媒体资源。
5. TCP连接可以被主动关闭，也可以等服务器关闭。

## TCP

---

### 三次握手

1. Client → Server: `SYN, seq=x`

客户端发送 `SYN` 同步报文，表示意向建立连接；生成当前自己的报文初始序列号 `seq=x`

2. Server → Client: `SYN, ACK, ack=x+1, seq=y`

服务器收到 SYN 后，返回 SYN + ACK 同步+确认报文，表示同意连接。服务器生成自己的报文初始序列号 `seq=y`，并设确认号 `ack=x+1`（期望下一个收到的x+1号数据，意味着客户端SYN即x号数据已被接收到）。

### 3. Client → Server: `ACK, ack=y+1, seq=x+1`

客户端收到 SYN + ACK 后，发送 ACK 确认报文，表示连接建立成功。确认号 `ack=y+1` 确认收到了服务器的 SYN；报文序列号自增 `seq=x+1`（正常情况下增加的大小是上一个报文的数据大小，但 SYN只占1个序列号，因此此时是x+1）

三次握手后客户端和服务端都进入了ESTABLISHED状态。

## 四次挥手

前两次是结束 Client → Server 的传输，后两次是结束 Server → Client 的传输、最后关闭连接。

### 1. Client → Server: `FIN, seq=x`

客户端发送 FIN 结束报文，表示不再发送数据、请求关闭连接。FIN占用一个序列号 `seq=x`

### 2. Server → Client: `ACK, ack=x+1`

服务器收到 FIN 后，返回 ACK确认报文( ack=x+1)，表示确认了客户端的结束请求。此时客户端已停止发送数据，但是服务器依然可以继续把数据传完。

### 3. Server → Client: `FIN, seq=y`

服务器也不需要再发送数据了，主动向客户端发送 FIN 结束报文，表示要关闭连接。序列号 `seq=y`

### 4. Client → Server: `ACK, ack=y+1`

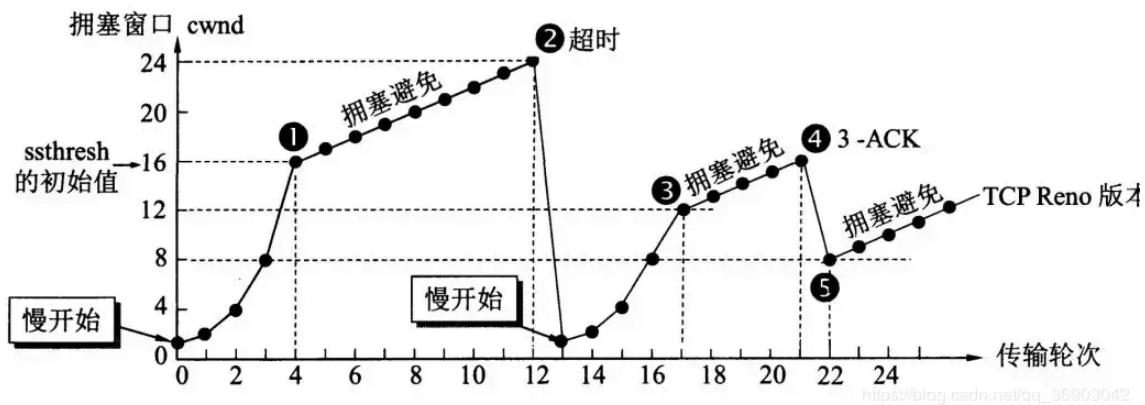
客户端收到 FIN 后，返回 ACK确认报文( ack=y+1)，表示收到。客户端最后等待 2\*MSL 最大报文生存时间，保证服务器能正确关闭（若服务器未收到ACK会重发FIN，不能太早关）。

服务器收到最后的ACK后进入CLOSED状态；客户端等待超时后也进入CLOSED状态。

## 如何保证传输可靠？

- **序列号 `seq`**：给每个数据包一个序列号，避免了数据失序、重复接收。
- **确认应答 `ACK`**：接收方必须回复一个ACK确认报文。使用累积确认机制：`ack=n` 表示期望下一个序列号是n（前面的均已收到）。
- **校验和**：TCP在发送前，计算数据的校验和，存入头部 `checksum`；接收方重新计算，并与头中的比对。
- **超时重传**：等待ACK超时则重传；**快重传**：接收到3个相同的ACK（A->B 多个数据包中的一个丢了，即使后面的送到了B，B也只会回应丢的那个期望的ACK）直接触发重传、无需等待超时，这是基于接收方反馈的重传。
- **流量控制**：TCP连接的每一方都有固定大小的缓冲区，接收方不能接纳超出缓冲区大小的数据。接收方通过**滑动窗口**控制发送方的发送速度。接收方决定窗口大小，在ACK中放入TCP头部 `window` 告诉发送方，那么发送方接下来就控制数据大小不超过 `window`。
- **拥塞控制**：发送方通过**拥塞窗口**来跟踪网络拥塞程度、控制要发送的数据量。
  - **慢启动**：从小数据量开始、探测网络拥塞程度，1 - 2 - 4 - 8翻倍；
  - **拥塞避免**：慢启动超过**拥塞阈值**`ssthreash`（初始16）进入拥塞避免阶段，缓慢增大窗口，16-17-18-19递增1；
  - 超时：超时说明网络拥塞严重，窗口降为1、`ssthreash`取之前窗口的一半，重新慢开始；
  - 3ACK（快重传）：说明有拥塞但不严重，窗口减半，`ssthreash`取之前窗口一半（=当前的窗口大小），直接进入拥塞控制阶段。

发送方的**发送窗口**（发送的数据量大小）取当前滑动窗口和拥塞窗口的较小值。



## TCP vs UDP

	TCP	UDP
是否面向连接	是	否
是否可靠	是	否
是否有状态(发送接收状态)	是	否
性能	较慢	较快
传输形式	字节流	数据报文段
首部开销	20 ~ 60 bytes	8 bytes
是否提供广播或多播服务	否	是

## HTTPS加密机制

- 握手过程**: 客户端和服务端协商加密协议，交换公钥和证书，生成会话密钥；
- 非对称加密**: 客户端使用服务器的公钥加密会话密钥，服务器使用私钥解密，双方得到相同的会话密钥。
- 对称加密**: 使用会话密钥对数据进行加密和解密，保证传输过程中的数据安全。
- 消息认证**: 使用哈希函数和消息认证码确保数据在传输过程中没有被篡改。

## 手撕

### 算法

#### 快排

先在每次递归时确定一个pivot，根据pivot给当前范围的数组划分左右分区，使左分区<pivot、右分区>pivot，然后分治。

```
void quicksort(int[] nums, int low, int high) {  
    // quicksort: 每次根据pivot划分左右分区，再分治递归.
```

```

// pivot通常是第一个或最后一个元素,但是固定pivot导致遇到最差情况变O(n^2),比如遇到已排序
数组
if (low >= high) return;
// 选取第一个元素作为pivot,那么最后它应该和左分区最后一个元素交换
int pivot = nums[low];
int i = low + 1, j = high;
while (i <= j) {
    while (i <= j && nums[i] <= pivot) i++; // 从左找到比pivot大的
    while (i <= j && nums[j] >= pivot) j--; // 从右找到比pivot小的
    // 交换,使其在正确的分区内
    if (i < j) {
        int tmp = nums[i];
        nums[i] = nums[j];
        nums[j] = tmp;
    }
}
// 最后i在右分区第一个位置,j在左分区最后一个位置
// 交换pivot和j,使得pivot在正确位置(也就是最终位置)
nums[low] = nums[j];
nums[j] = pivot;
quicksort(nums, low, j - 1);
quicksort(nums, j + 1, high);
}

```

变体：**快速选择算法**：不需要排序整个数组，只是找到第K大的元素，每次只需要递归左区间或右区间，找目标下标即可。

## 堆排序

对无序数组，先构建为大根堆（堆顶是最大但子树不一定有序），再依次取出堆顶放到数组末尾

```

public class HeapSort {
    public static void heapSort(int[] arr) {
        int n = arr.length;
        // 1. 构建最大堆
        buildMaxHeap(arr, n);
        // 2. 逐步取出最大值，并调整堆
        for (int i = n - 1; i > 0; i--) {
            swap(arr, 0, i); // 交换堆顶（最大值）和最后一个元素
            maxHeapify(arr, i, 0); // 重新调整堆（排除已排序部分）
        }
    }

    private static void buildMaxHeap(int[] arr, int n) {
        for (int i = n / 2 - 1; i >= 0; i--) {
            maxHeapify(arr, n, i);
        }
    }

    // 堆化：以当前节点为根，调整为最大堆，本质上是检查当前节点是否应该下沉
    private static void maxHeapify(int[] arr, int n, int i) {
        int largest = i; // 假设当前节点是最大值
        int left = 2 * i + 1; // 左子节点
        int right = 2 * i + 2; // 右子节点
        // 找到更大的子节点
        if (left < n && arr[left] > arr[largest]) largest = left;
        if (right < n && arr[right] > arr[largest]) largest = right;
    }
}

```

```

// 如果 largest 变了，则交换并递归调整
if (largest != i) {
    swap(arr, i, largest);
    maxHeapify(arr, n, largest); // 新下沉的元素可能还需要继续下沉
}
}

private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

public static void main(String[] args) {
    int[] arr = {3, 6, 8, 10, 1, 2, 1};
    heapSort(arr);
    System.out.println(Arrays.toString(arr)); // [1, 1, 2, 3, 6, 8, 10]
}
}

```

优先队列基于堆实现。

```

PriorityQueue<Integer> pq = new PriorityQueue<>(n, (a, b) -> b - a);
pq.offer(1);
pq.offer(2);
pq.peek(); // 2
pq.poll(); // 2
pq.poll(); // 1
pq.isEmpty(); // true

```

## Java

### 多线程交替打印ABC

#### semaphore

semaphore用 permits 代表该信号量计数。 acquire() 会使计数-1、可能阻塞； release() 使计数+1且没有上限。

设置3个共享信号量分别代表ABC线程。

```

public class Test {
    private final Semaphore sema = new Semaphore(1); // A先执行
    private final Semaphore semb = new Semaphore(0);
    private final Semaphore semc = new Semaphore(0);
    void printA() {
        for (int i = 0; i < 10; i++) {
            try {
                sema.acquire();
                System.out.println('A');
                semb.release();
            } catch (Exception e) {}
        }
    }
}

```

```

    }
}

void printB() {
    for (int i = 0; i < 10; i++) {
        try {
            semb.acquire();
            System.out.println('B');
            semc.release();
        } catch (Exception e) {}
    }
}

void printC() {
    for (int i = 0; i < 10; i++) {
        try {
            semc.acquire();
            System.out.println('C');
            sema.release();
        } catch (Exception e) {}
    }
}

public static void main(String[] args) {
    Test t = new Test();
    new Thread(t::printA).start();
    new Thread(t::printB).start();
    new Thread(t::printC).start();
}
}

```

## synchronized

设置一个锁对象，通过 `synchronized` 加锁、用 `wait()` 和 `notifyAll()` 进行线程通信。

`wait()`/`notify..()` 只能在 `synchronized` 里用。`wait()` 时会阻塞当前线程、释放掉 `synchronized` 的锁；`notifyAll()` 会唤醒所有被阻塞线程让他们竞争抢锁，而 `notify()` 是随机唤醒一个加锁。

```

public class Test {
    private int state = 0; // 共享变量控制执行顺序(0 -> A, 1 -> B, 2 -> C)
    Object lock = new Object(); // 共享锁对象

    void printA() {
        for (int i = 0; i < 10; i++) {
            synchronized(lock) {
                try {
                    while (state % 3 != 0) lock.wait(); // 不是当前线程执行时，阻塞、
                                                // 释放锁
                } catch (Exception e) {}
                System.out.println('A'); // 轮到当前线程执行了
                state++; // 状态更新
                lock.notifyAll(); // 唤醒其他所有线程
            }
        }
    }

    void printB() {
        for (int i = 0; i < 10; i++) {
            synchronized(lock) {
                try {

```

```

        while (state % 3 != 1) lock.wait();
    } catch (Exception e) {}
    System.out.println('B');
    state++;
    lock.notifyAll();
}
}

void printC() {
    for (int i = 0; i < 10; i++) {
        synchronized(lock) {
            try {
                while (state % 3 != 2) lock.wait();
            } catch (Exception e) {}
            System.out.println('C');
            state++;
            lock.notifyAll();
        }
    }
}

public static void main(String[] args) {
    Test t = new Test();
    new Thread(t::printA).start();
    new Thread(t::printB).start();
    new Thread(t::printC).start();
}
}

```

## ReentrantLock

相比于Synchronized对象锁，ReentrantLock配合Condition提供了更细粒度的多线程控制。

**ReentrantLock**本身仍然是一整个锁，但是**Condition**可以只是它的一部分，线程加锁在lock上，但是阻塞在**condition**上。

配合**Condition**，使用**await()**/**signal()**,**signalAll()**来通信，让线程可以因某条件而等待，让满足某条件的线程被唤醒。

```

public class Test {
    private int state = 0; // 共享变量控制执行顺序(0 -> A, 1 -> B, 2 -> C)
    ReentrantLock lock = new ReentrantLock();
    Condition conditionA = lock.newCondition();
    Condition conditionB = lock.newCondition();
    Condition conditionC = lock.newCondition();
    void printA() {
        for (int i = 0; i < 10; i++) {
            lock.lock();
            try {
                while (state % 3 != 0) {
                    conditionA.await();
                }
                System.out.println('A');
                state++;
                conditionB.signal();
            } catch (Exception e) {}
            finally {

```

```

        lock.unlock();
    }
}

void printB() {
    for (int i = 0; i < 10; i++) {
        lock.lock();
        try {
            while (state % 3 != 1) {
                conditionB.await();
            }
            System.out.println('B');
            state++;
            conditionC.signal();
        } catch (Exception e) {}
        finally {
            lock.unlock();
        }
    }
}

void printC() {
    for (int i = 0; i < 10; i++) {
        lock.lock();
        try {
            while (state % 3 != 2) {
                conditionC.await();
            }
            System.out.println('C');
            state++;
            conditionA.signal();
        } catch (Exception e) {}
        finally {
            lock.unlock();
        }
    }
}

public static void main(String[] args) {
    Test t = new Test();
    new Thread(t::printA).start();
    new Thread(t::printB).start();
    new Thread(t::printC).start();
}
}

```

## 线程池的使用

Future阻塞获取结果，CompletableFuture可以异步获取

`invokeAll` 等到所有都执行完了才会获得结果

```

// 创建线程池
ThreadPoolExecutor executor = new ThreadPoolExecutor(
    5, 10, 60L, TimeUnit.SECONDS,

```

```
new LinkedBlockingQueue<>()
);
// 创建任务列表
List<Callable<Integer>> tasks = new ArrayList<>();
for (int i = 0; i < 10; i++) tasks.add(() -> {...}); // 模拟任务
try {
    // invokeAll执行任务，等待所有任务执行完成
    List<Future<Integer>> futures = executor.invokeAll(tasks);
    // 打印结果
    for (Future<Integer> future: futures) println(future.get());
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
} finally {
    executor.shutdown();
}
```

`submit`单条执行、逐个阻塞获取结果

```
// 或者submit执行任务
Future<Integer> future = executor.submit(tasks[i]);
```

定时器

阻塞队列

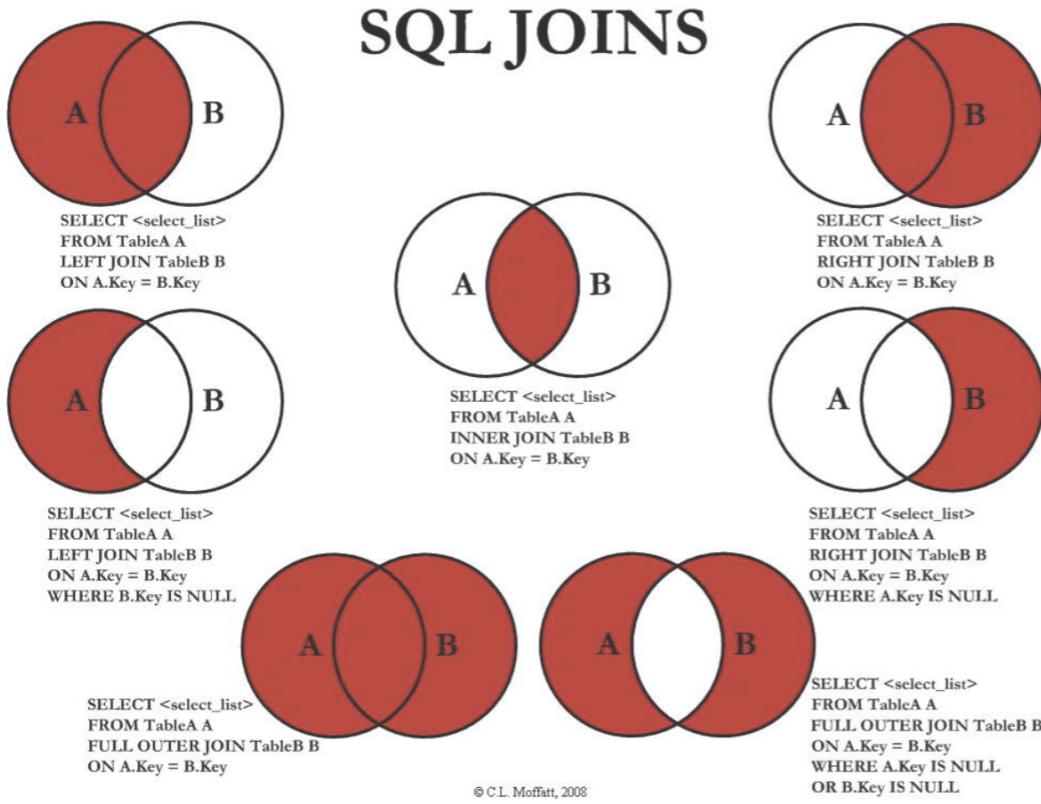
多线程定时调用

http场景根据userid每分钟统计并发量

## MySQL

### 连接

下图展示了 LEFT JOIN、RIGHT JOIN、INNER JOIN、OUTER JOIN 相关的 7 种用法。



`inner join ... on ...`: 内连接（类似自然连接），不满足就去除

`left/right join ... on ...` : 左连接，保留左表为基础、对右边检查on条件，不满足置NULL

`cross join`: 笛卡尔积

## 函数

`datediff(date1, date2)` 计算日期格式的天数差

`DATE_SUB(OrderDate, INTERVAL 2 DAY)`

`round(x, 3)` 将x保留3位小数

`count(*)` 与 `count(1)` 等效，都会把NULL值算入；而 `count(字段)` 不会算

## 配置文件

```
# 使用官方 OpenJDK 运行时镜像作为基础镜像
FROM openjdk:17-jdk-slim
# 设置工作目录
WORKDIR /app
# 复制构建好的 jar 文件到容器中（请确保路径正确）
COPY target/*.jar app.jar
# 运行 Spring Boot 应用
CMD ["java", "-jar", "app.jar"]
```

### 1. 构建 Docker 镜像

```
docker build -t myapp:latest .
```

## 2. 运行 Docker 容器

```
docker run -d -p 8080:8080 --name myapp myapp:latest
```

## 3. 查看日志

```
docker logs -f myapp
```

## 4. 停止 & 删除容器

```
docker stop myapp
```

```
docker rm myapp
```

# CI/CD

.github/workflows/deploy.yml

```
name: Java CI/CD

on:
  push:
    branches:
      - main # 仅当推送到 main 分支时触发

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: 检出代码
        uses: actions/checkout@v3

      - name: 设置 JDK 17
        uses: actions/setup-java@v3
        with:
          distribution: 'temurin'
          java-version: '17'

      - name: 编译 & 运行测试
        run: mvn clean package -DskipTests=false # 运行单元测试

      - name: 生成 JAR 文件
        run: ls -lh target/*.jar # 确保 jar 文件生成成功

      - name: 部署到服务器
        uses: appleboy/scp-action@master
        with:
          host: ${{ secrets.SERVER_HOST }}
          username: ${{ secrets.SERVER_USER }}
          password: ${{ secrets.SERVER_PASS }}
```

```
source: "target/*.jar"
target: "/home/${{ secrets.SERVER_USER }}/app"
```

# 其他

## 鉴权与授权

JWT 存基本信息 + Redis 存权限信息 + SpringSecurity进行权限控制

- JWT是无状态的，目的是在分布式场景下，不用维护有状态（数据库Redis等）信息很复杂，它签发后只存在前端；但是它无法被后端主动取消，因此不适合存权限信息（权限更改无法同步到已生效JWT）。
- Redis存Token，使用短Token+刷新机制可以兼顾安全性和性能，但是分布式场景下增加了查询和维护的复杂度。
- Spring Security可以帮助进行URL级别的权限控制、方法级别的权限控制。通过 `SecurityContextHolder` 储存用户信息，从jwt+redis中获取。

## JWT

三部分构成：`Header.Payload.Signature`。Header表示该JWT所用的加密算法（多种对称/非对称加密算法可选），Payload存放了声明 `claims` (`id, 签发时间,过期时间`) 和其他用户信息，Signature由前两者生成、用于防篡改。

JWT签发后仅存在前端，后端无法手动销毁。因此要么JWT+Redis，认证-权限分离存储；要么JWT+黑名单，给Token逻辑过期（但实际上没过期，所以还是不推荐）。

JWT无法手动过期的通用处理方案：短过期+refresh机制，在过期前给客户端重新发token；redis黑名单

## Spring Security

Spring Security 的核心是过滤器链组成，执行顺序如下：

### 1. JwtAuthenticationFilter (解析 JWT 获取用户身份)

请求一进来就尝试从请求头检查JWT，没有则放行；解析JWT拿到基本信息，从数据库/缓存里拿到权限信息，存入 `SecurityContext`。这样每次请求都会用JWT重新获取Redis里的权限，连带着更新了 `SecurityContext`、防止越权访问。

### 2. UsernamePasswordAuthenticationFilter (处理表单登录)

通常用于 `/login`，通过UserService校验登录，认证成功存入 `SecurityContext`，返回JWT

### 3. SecurityContextPersistenceFilter (存储和恢复 SecurityContext)

已默认实现(session)。每次请求都会检查 `SecurityContext`，它其实就是token，默认从 session取，可以配置为从redis取；但是它存在线程本地 `ThreadLocal` 中，随当前请求的线程而创建、销毁。

### 4. ExceptionTranslationFilter (异常处理)

拦截 `AccessDeniedException` (无权限) 和 `AuthenticationException` (未认证)，并返回适当的HTTP响应 (401 403)。

## 5. FilterSecurityInterceptor (权限检查)

所有过滤器执行完，**最后执行权限校验**。可以在这里进行基于URL的角色权限控制，无权限抛

`AccessDeniedException`

## 6. 方法级别的 @PreAuthorize 权限校验

从 `SecurityContext` 中拿到身份和权限，和方法注解需要的角色进行校验，无权限抛

`AccessDeniedException`。**基于AOP**

要开启 `@PreAuthorize`，需要配置 `SecurityConfig.java` 加上 `@EnableMethodSecurity` 注解。

## 越权访问问题

1. JWT里存了角色信息，JWT不主动过期：通过认证-权限分离解决

2. Redis里存了角色信息，Spring SecurityContext不同步：每次请求都重新获取Redis里的角色、刷新SecurityContext。或者每次更新角色的业务代码之后进行手动刷新。

## 代码沙箱的实现

原理：通过System包将用户代码存储为.java文件，放在全局唯一的文件夹下；使用Runtime类获得运行环境实例，`exec()` 执行系统命令 `javac` 来编译用户代码成.class文件，然后再执行系统命令 `java` 来运行该文件。对于每次 `exec()` 执行系统命令，都用Process类获得该进程的运行结果。

### 1. Runtime类

- `Runtime.getRuntime()` 用于获取当前Java进程的**运行实例(JVM)**。Runtime是单例模式，一个JVM对应一个 `Runtime` 对象。
- `Runtime.getRuntime().exec(cmd)` 用于执行**系统命令**，它会创建一个子进程去执行命令，返回一个 `Process` 对象代表孩子进程。`exec()` 可以直接传命令的字符串，也可以通过参数指定命令的参数和执行的文件夹位置。

### 2. Process对象

`Process` 代表了一个进程，可以通过 `Process` 控制和获取进程的输入输出等信息：

- 读取进程输出 `getInputStream()`
- 读取进程错误信息 `getErrorStream()`
- 向进程输入数据 `getOutputStream()`
- 等待进程结束 `waitFor()`
- 获得进程的退出码 `exitValue()`

### 3. 设计模式

- 工厂方法模式：本地代码沙箱、远程代码沙箱（微服务）、第三方代码沙箱。实现工厂接口创建多个子工厂，子工厂再实例化产品。
- 策略模式：Java语言策略、其他语言策略。实现 `JudgeStrategy` 接口，在 `JudgeManager` 中根据参数选用策略实例。
- 代理模式：实现 `CodeSandbox` 接口，注入沙箱实例，重写方法在前后加上日志打印。

## finalize

```

class Myobj {
    @Override
    protected void finalize() throws Throwable {
        System.out.println("User-->finalize()");
    }
}

```

Object类的方法，会在它即将被GC回收时调用。但是不推荐使用：Java的GC是自动的，因此finalize的时机是不可知的，因此不适合释放资源（连接、内存等）；此外延长了GC的时间。一个较好的使用场景是帮助该Object“再活一次”。

## IO多路复用机制

特性	select	epoll
文件描述符限制	默认1024，可调整	理论上无限制（受内存限制）
调用方式	轮询所有 fd O(N)	事件驱动 O(1)
事件通知机制	进程主动遍历 fd	事件触发（回调通知）
内核与用户空间交互	每次调用都要拷贝 fd 集合	仅在注册、修改时拷贝
适用场景	小规模并发(<1000连接)； Windows	大量并发连接(10K+ 连接)； Linux

**select**: 通过数组存储文件描述符（fd），然后每次调用时遍历整个数组，检查哪些 fd 发生了 I/O 事件。需要IO线程主动一直轮询。每次调用都要拷贝真个fd数组至内核态，当并发数增加性能下降。

**poll**: select的链表版，突破了fd大小限制，更灵活，但仍需轮询遍历

**epoll**: 采用 红黑树 存储 fd，维护 就绪事件链表，仅返回有事件的 fd；根据事件类型（读、写）进行分发，调用相应的回调函数进行处理。

- **初始化**: 创建 epoll 实例 epoll\_create，并注册 epoll\_event。
- **监听事件**: 有 fd 就绪，就会在红黑树中进行标记、假如到链表中。调用 epoll\_wait() 等待事件触发（如客户端请求），如果已经有就绪事件就直接返回。

## Dubbo - RPC

### 1. 服务提供者（Provider）启动

- 服务提供者是实现业务逻辑的微服务，它通过 Dubbo 向注册中心（如 Zookeeper 或 Nacos）注册自己的服务，告诉其他服务（消费者）自己提供了哪些接口。
- **服务接口暴露**: 在服务提供者中，服务实现类会使用 @DubboService 注解暴露一个接口（例如，UserService），Dubbo 会将该接口暴露出来供其他服务调用。

```
@DubboService
public class UserServiceImpl implements UserService {
    @Override
    public User getUserById(int id) {
        // 业务逻辑
    }
}
```

- **注册到注册中心**: 服务提供者启动时，会将该服务的信息（如接口名称、版本、服务地址等）注册到 **Zookeeper** 或 **Nacos** 等服务注册中心。其他服务可以通过这些注册信息找到该服务的地址。

## 2. 服务消费者（Consumer）启动

- 服务消费者是调用服务的微服务，它依赖于服务提供者来获取业务数据。消费者通过 `@DubboReference` 注解来引用远程服务。

```
@RestController
public class UserController {
    @DubboReference
    private UserService userService;

    @GetMapping("/getUser")
    public User getUser(int id) {
        return userService.getUserById(id);
    }
}
```

- **服务发现**: 消费者启动时，Dubbo 会从注册中心获取可用的服务提供者信息。如果注册中心中有多个实例，消费者会根据负载均衡策略选择一个实例进行调用。

## 3. 服务调用过程

- **消费者调用服务**: 消费者通过 Dubbo 提供的代理机制（Proxy）调用远程服务，实际上调用的是本地的代理对象。Dubbo 会根据调用的接口方法生成远程调用的请求，并通过网络将请求发送到服务提供者。
- **网络请求传输**: Dubbo 会根据配置的协议（如 **Dubbo 协议**、**HTTP 协议**）和序列化方式（如 **Hessian**、**Protobuf**）将方法参数序列化并传输到服务提供者。
- **服务提供者处理请求**: 服务提供者接收到请求后，反序列化参数并调用实际的业务逻辑代码。处理完请求后，服务提供者将结果通过网络返回给消费者。
- **消费者接收响应**: 消费者接收到服务提供者返回的响应后，Dubbo 会将响应结果反序列化并返回给调用者。

# 分布式CAP问题

- **Consistency**: 所有节点在任何时刻保持强一致性（弱一致性是指允许短暂的不一致，最终一致性）
- **Availability**: 系统在任何时刻保持高可用（0.9999级可用）
- **PartitionTolerance**: 分布式、非单节点，在网络分区部分断网情况下，系统整体依然可用，可以分区容错

## CAP冲突

- **AC**: 放弃P, 就是**单节点**情况, 不是分布式系统
- **CP**: 放弃A, **强调一致性**, 意味着低性能, 因为需要等待节点同步
- **AP**: 放弃C, **强调高可用**, 意味着允许部分节点在有限时间内不一致 (最终一致性)

#### Redis集群 - AP

Redis集群通过**数据分片**将数据分布到多个节点, 是去中心化的, 每个节点都可以尽可能独立服务(**P**); Redis集群**主从复制**, 每个主节点都有从节点, 主节点挂了就会进行快速选举(**A**)。Redis集群使用**异步复制**来降低延迟, 可能造成暂时的不一致 (比如没来得及复制就挂了, 会从其他节点复制数据副本)。

#### Nacos - AP / CP

- **永久**Client节点: 永久注册在Nacos中、出现故障和崩溃也留着, 使用**CP**
- **临时**Client节点: 默认, 生命周期与其代表的服务实例的健康状况相关, 故障则删除, 使用**AP**

## 排查CPU占用(Java)

---

1. Linux查看系统负载 `top -c` : %CPU 是CPU使用率, %MEM 是内存占用率。查看异常进程PID。
2. 查看其线程负载 `top -H -p <PID>` : -H 以线程展示, 查看异常线程TID。
3. 查看线程java堆栈 `jstack <PID> | grep -A 10 "<TID_16进制>"` : 分析堆栈, 看是业务代码、GC、还是锁竞争

## 大模型与RAG, CoT、ToT

---