hashcode方法：

在Object类中有一个hashCode方法，native的，hashcode主要用来获取对象的hash值，hashcode方法的主要作用是和散列集合(HashSet，HashMap，HashTable)一起使用

在向集合中插入一个值时(不可重复)，会遍历集合，是否和插入的元素重复，这样效率太低，这时可以使用hash值，当有个对象要插入到集合中时，首先计算出对象的hash值，然后根据一些运算，确定对象的位置，当下一个对象插入时，同样计算出hash值，计算出位置，如果当前位置上有元素，则比较对象是否相等(hash相等，不一定对象相等，对象相等，hash一定相等)，这样就只需比较一次

在重写equals/hashCode方法的时候，一定要验证两个逻辑是否相等

equals和 ==

对于基本数据类型(int，float….)，==比较的就是具体值，而对于引用类型，比较的是其地址是否相等

Object中的equals实际就是==，比较的是地址，而大部分类都对此方法进行重写，首先是通过==比较是否是同一个对象，然后在去判断其元素是否相等

equals和hashcode

hashCode比较的是hash值，而equals用于比较对象的具体值(equals先比较的是地址，再比较元素)

两个对象hashcode相等，equals不一定相等

两个对象equals相等，hashcode可能相等(equals先比较的是地址，再比较元素如果只是元素相等，对象不等)，hashcode就不等了

Object的hashcode是native的

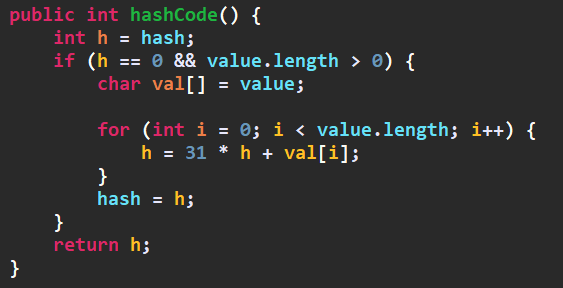
a==b和a.hashcode()==b.hashcode()的区别：

a==b，则a.hashcode()==b.hashcode()恒成立

a!=b，但a.equals(b)=true，则a.hashcode()==b.hashcode()恒成立

这种说法也不一定对，如果类继承的是Object的hashcode，则生成hashcode的策略是根据内存中的地址，如果类重写了hashcode，则有类自己决定，但是大部分生成策略都是根据类中的每个属性生成hash值，比如String的hashcode，是遍历了数组的每个char生成的

如果对于String这类重写了hashcode方法的类来说，上示是成立的，但是没有重写hashcode，使用的是Object默认的hashcode则上示不成立，所以，如果我们在重写hashcode的方法时，应尽量将属性作为hash



a!=b，但a.equals(b)=false，则不确定，可能会出现hash冲突

a.hashcode()==b.hashcode()，无法推出a==b或a.equals(b)=true

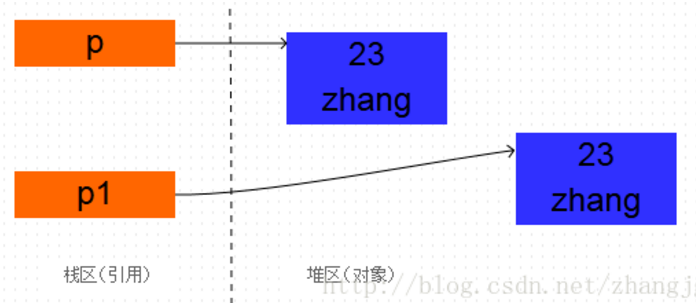
clone

创建对象的方式：new+clone

new是创建一个新的对象，而clone是对以存在的对象进行复制，两个属性完全一样，除了内存中的地址

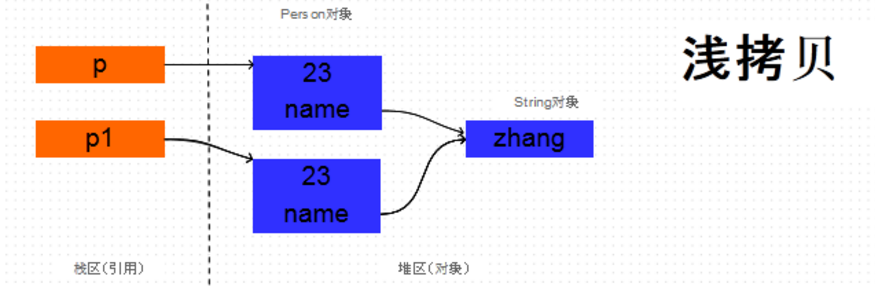
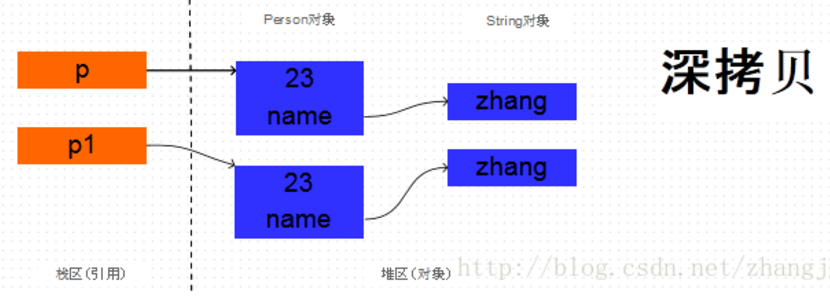
对象能被clone，必须实现Cloneable接口

clone出来的对象不是简单地引用而已，是在内存中真正存在的一个对象

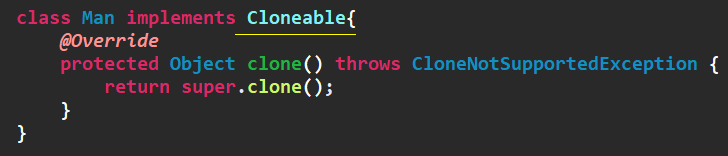
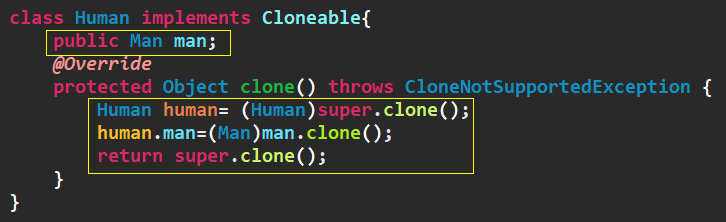


深浅拷贝

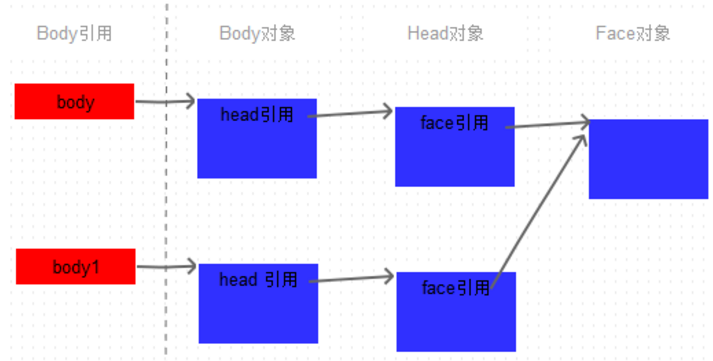
对于基本数据类型，clone执行的深拷贝，而引用类型则是浅拷贝

clone实现深拷贝

这里Human引用Man，在对Human进行clone时，Man是引用类型，所以是浅拷贝，要实现深拷贝就得去clone Man，所以在Human的clone方法中对Man进行了clone，但彻底的深拷贝是不可能实现的，只有是深拷贝，就得一直拷贝下去，直到clone了一个不能clone的类型(比如String)



能被clone，必须实现Cloneable接口，并且重写clone方法，调用父类的clone方法进行clone

String为什么没有clone？没必要，String的字符串本身就是final，常量池中只有一份，clone出来就违背了final的原则

Java常用包：

Java.lang：java进行程序设计的基础包，Object，Math，String，System，Thread，Throwable等

Java.util：集合框架，日期处理

Java.io：文件，流操作

String.split()只有String采用，StringBuffer没有

String str=" We Are Happy "

str.split(" "):分割到Happy后就不会往后分割，所以分割的结果为"","","We","Are","Happy"

str.split(" ",n)

n表示分割出来串的个数

n=0:表示不会分割后面的，即结果为"","","We","Are","Happy"

n=1：表示分割出来的串只有一个，即原来的串" We Are Happy "

n=2：2个串，"" " We Are Happy "

依次推，分割出来串最多的个数为str.lengt()-1

所以要想分割happy后的"",可以声明n=str.lengt()-1

Comparator和Comparable接口之间区别？

Comparable接口中只有一个public int compareTo(T o)方法，用于比较两个对象的大小

Comparator接口中有很多static方法(jdk1.8以后，接口中可以有静态方法的实现体)和boolean equals(Object obj)/

int compare(T o1, T o2)两个抽象方法

compareTo更适用于String，基本类型包装类中比较两个数的大小

Comparator定义的一个比较器，重点是器，一个模板，常在Collection集合中对集合元素进行排序

Hash的负载因子为什么是0.75？

一方面是大量实验的结果，还有就是0.75\*capacity(2的幂)的结果为整数

cookie的内容主要包括：名字，值，过期时间，路径和域。路径与域一起构成cookie的作用范围。

Cookie中只能保管ASCII字符串，假如需求存取Unicode字符或者二进制数据

Session中能够存取任何类型的数据，运用起来十分便当。能够把Session看做是一个Java容器类。

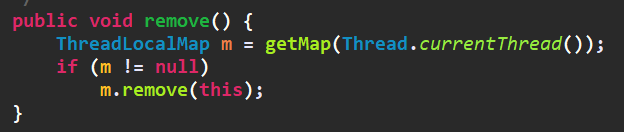
Cookie支持跨域名访问，例如将domain属性设置为“.biaodianfu.com”，则以“.biaodianfu.com”为后缀的一切域名均能够访问该Cookie。

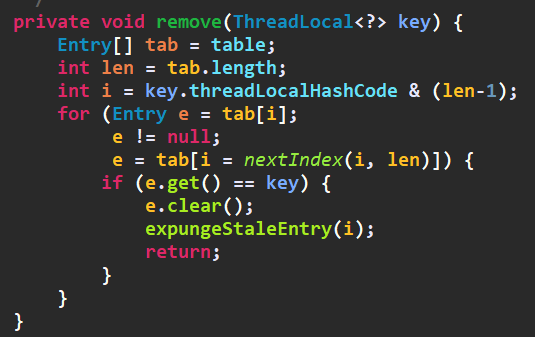
ThreadLocal提供的只是一个浅拷贝，可以通过重写ThreadLocal的initialValue()函数来自己实现深拷贝

Spring与线程安全：

对于作用域为Singleton的Bean而言是无状态的是不存在线程安全问题，对于某些bean需要线程共享的bean可以存放在ThreadLocal

ThreadLocal内存泄漏，如果ThreadLocal置为null，会导致Thread中的ThreadLocalMap的key就为null了，但是value值任然存在，而不会被引用，导致一直占有内存，所以在将ThreadLocal置为null之前，可以调用ThreadLocal的remove方法，将对应ThreadLocal的value从ThreadLocalMap中移出(ThreadLocalMap是一个数)





Singleton的Bean在Spring中的生命周期：

实例化bean

引入依赖(IOC)

如果Bean实现BeanNameAware接口，会调用setBeanName(String beanId)，将Spring配置文件中的Bean的ID传递给Bean

如果Bean实现BeanFactoryAware接口，会调用setBeanFactory()，传递的是Spring工厂本身

如果Bean实现ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传递的是ApplicationContext对象，用于获取其他Bean(和BeanFactory类似)

调用实现BeanPostProcessor接口的Bean(Bean后置处理器)的postProcessBeforeInitialization(Object String)方法

如果Bean实现InitializingBean接口，调用afterPropertiesSet对bean操作

如果配置文件中配置了init-method属性，调用指定的方法处理bean(和InitializingBean接口一样)

调用实现BeanPostProcessor接口的Bean(Bean的后置处理器)的postAfterInitialization(Object，String)方法

Bean随便使用

**new—beanNameAware—BeanFactoryAware—ApplicationContextAware—BeanPostProcessor—init—BeanPostProcessor**

当Bean被销毁之前

如果bean实现DisposableBean接口，会调用destroy方法

如果配置文件中声明destroy-method属性

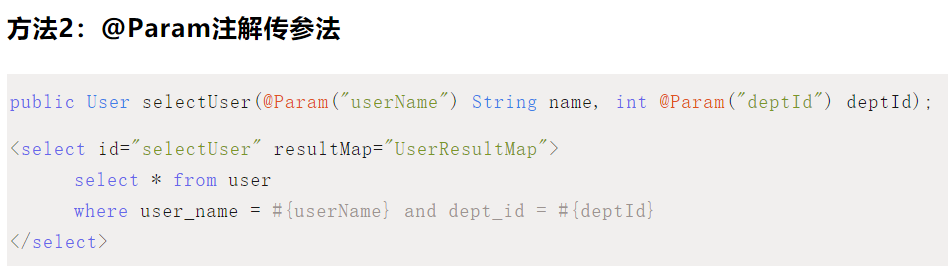
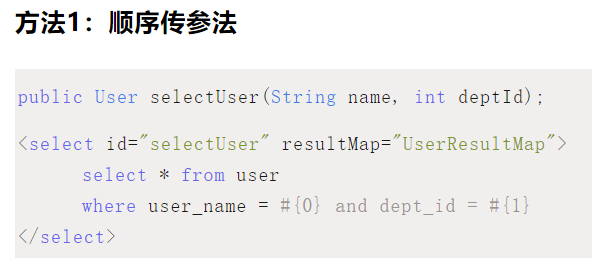
Mybatis缓存：

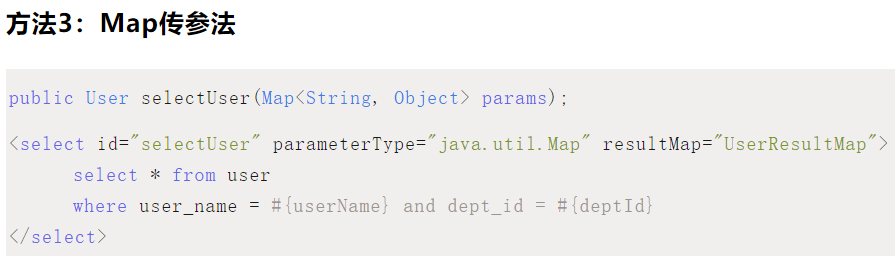
一级缓存的生命周期和SqlSession一致(多个SqlSession之间会导致幻读，重复读)

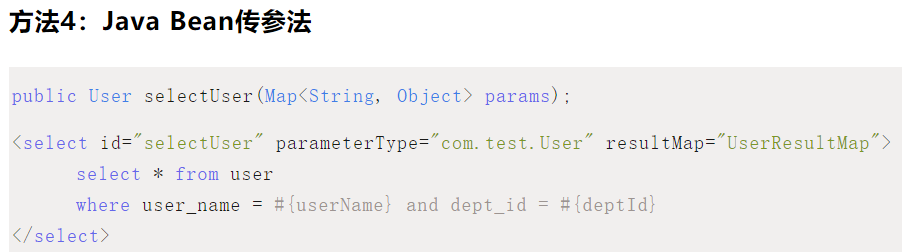
一级缓存内部是通过HashMap实现

二级缓存实现了SqlSession之间缓存数据的共享

Mybatis传递参数的方法：







可以在Spring中注入一个null 和一个空字符串

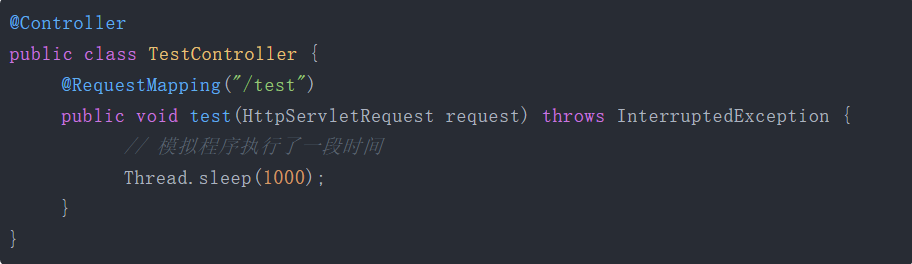
在客户端，一个浏览器能创建的 Cookie 数量最多为 300 个，并且每个不能超过 4KB，每个 Web 站点能设置的 Cookie 总数不能超过 20 个

事务：

JDBC事务主要优点是API简单，性能较好，但是JDBC事务不支持跨数据，所以分布式场景不能使用JDBC

JTA事务支持分布式事务，跨域访问，但是JTA很笨重

在Controller获取Request的方法以及线程安全：



线程安全，这里的request是方法的局部变量，每个线程唯一拥有，只有多个线程对类的属性进行操作是才会出现线程安全问题



线程安全，Request看似是类的属性，但是其实这里实际是一个Singleton的bean调用了一个Property的bean，所以每次执行方法的时候，获取的都是新的request

为什么我们调用start()方法时会执行run()方法，为什么我们不能直接调用run()方法？

当调用start方法时，才是创建线程，线程会自动执行run方法，而New Thread只是声明一个线程

java内存空间：程序计数器，java虚拟机栈，本地方法栈，堆，方法区

程序计数器：

用于记录(保存)当前线程正在执行的字节码的地址

程序计数器每个线程私有，这是一个唯一不会出现OutOfMemoryError的内存区域

java虚拟机栈：

java虚拟机栈使用描述java方法运行过程的内存模型

java 虚拟机栈会为每一个即将运行的 Java 方法创建一块叫做 “栈帧” 的区域

用于存储方法的局部变量表，基本数据类型，引用数据类型，返回值等

方法执行完毕，方法对应的栈帧将会出栈，并释放内存空间

java虚拟机栈线程私有

本地方法栈：

本地方法栈和java虚拟机栈类型，只不过本地方法栈是本地方法(native)的内存模型

堆：

用来存放对象

线程共享，java中只有一个堆

在JVM启动时创建，GC的主要场所

堆：先进先出

栈：先进后出

方法区：

存放被虚拟机加载的类信息，常量，静态变量

线程共享，方法区是堆的一个逻辑部分，整个JVM只有一个方法区

Class对象在方法区中

常量池：

一个类中的public static final变量，类被编译为class文件之后，类的所有

信息存储在class文件中，当class文件被加载到内存中，常量就会存放在常量池中

运行时可以向常量池中添加新常量

java中new一个对象的过程：

1，jvm将class文件加载到方法区中，

根据方法区中该类的信息，确定该对象的大小，一个类的所有对象大小都是一样的

JVM 在一个类被加载进方法区的时候就知道该类生产的每一个对象所需要的内存大小

2，在对内存中划分一块对应大小的内存给对象

3，为对象的成员变量进行赋初值(默认大小)

4，调用构造函数进行初始化

对象属性存放的位置：

局部变量(方法内部)：

当调用对象的方法时，JVM会创建一个方法栈，局部变量是基本数据类型时

变量名和值都保存在栈中，引用类型时，变量名保存在栈中，值保存在堆中

类变量：

基本数据变量名和值都保存在对象所在的堆中

引用类型变量名存放在栈中，对应的值存放在堆中

成员变量存在于堆内存中，随着对象的产生而存在，消失而消失。

局部变量存在于栈内存中，随着所属区域的运行而存在，结束而释放。

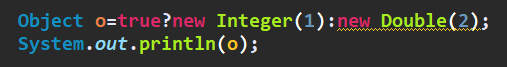
栈中存放的数据只有在执行方法的时候才会开辟，方法执行完被释放

对final属性赋值的地方：初始化，构造器，{ }

集合求长度的方法：size()

在堆中保存的是对象的属性，方法的代码保存方法区

在使用自定义类的hashcode，equals方法时必须对其重写

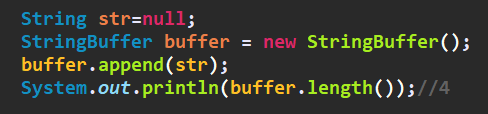


输出结果为1.0，三目运算符在输出的时候会和后面的类型保持一致

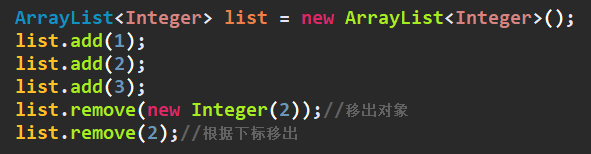
@Autowire和@Resource的区别？

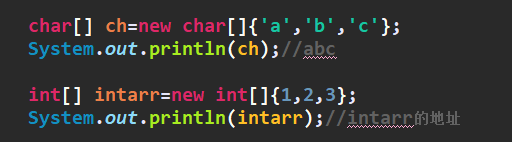
Autowire默认是按类型进行装配，Resource先是按名称进行装配，如果没有才用类型

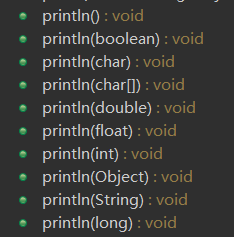
Autowire注解是Spring的，而Resource是javax的一个注解



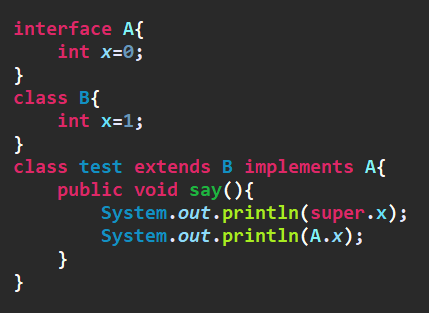
输出的结果为4，StringBuffer的append方法在追加时，append方法会判断str是否为null，如果为null，那么str的值为”null”







**Println会根据输出的类型调用对应的方法，在输出char数组时是遍历，而其他是直接输出地址**



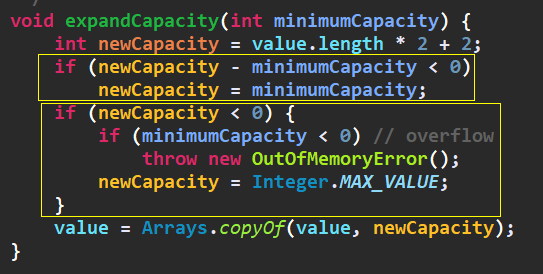
接口中的属性是常量 static final

String为什么不能像StringBuffer一样动态添加str？

String和StringBuffer底层都是采用char数组实现，但是String在创建的时候就声明了数组的大小

String str=new String(“123”) 对应底层是 char[]{‘1’,’2’,’3’}，创建好后就不能在修改

而StringBuffer在创建时，声明了一个长度为16的char数组，长度不够时会动态创建



StringBuffer的扩容方法

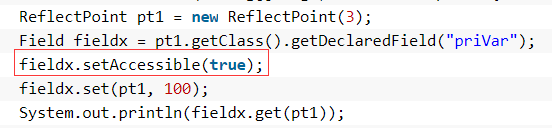
初始化长度为16，扩容机制为原来的2倍+2

第一个if判断：如果当前数组长度为20，但是追加的String的长度(minimumCapacity)为100，扩容后的长度为42，但是任然不能满足100，所以，将当前数组的长度设置为miniumCapacity

第二个if判断：扩容后，可能会超过int类型的范围，newCapacity就会为负数，只能将newCapacity设置为int的最大值，如果添加的String的长度超过了int最大值，长度也为负数，就会导致内存溢出

暴力反射：

通过反射获取类的private属性时，通过getDeclareField()获取，但是只能获取不能操作，需要设置可访问



DDL：查询语句

DML：创建表的操作

DCL：事务操作，commit，rollback

SQL：

Insert into tableName (name,dep,age) values ( , , , )

Insert into tableName (name，dep，age) select name，dep，age form tableName where

删除重复记录

Delete from dep e where e.ROWID>(

Select min(x.ROWID)

From emp x

Where x.empNo=e.empNo

);

Spring的ioc中的单例和加锁/volatile方式创建单例的区别：

加锁/volatile创建单例是在程序运行中创建的；而ioc中的单例是在程序启动时就会创建，是相互依赖的

加锁方式创建是存在并发性，防止创建多个；而ioc是在程序启动时创建，不存在并发性，所以在创建对象的时候不必加锁，直接创建即可



在调用setMessage时，可以只传入code参数

volatile关键字只能修饰类变量和实例变量。方法参数、局部变量、实例常量以及类常量都是不能用volatile关键字进行修饰的，因为volatile修改的变量会会多个线程访问并修改

并发编程三大特性：“原子性”、“有序性”、“可见性”

Java代码在执行次数达到一个阈值会触发JIT编译，一旦代码块被编译成本地机器码，下次执行的时候会直接运行编译后的本地机器码。所以这本地机器码必须被缓存起来，而缓存这个本地机器码的内存区域就是Code Cache(方法区)

RDB 有 2 种持久方式，同步 save 模式和异步 bgsave 模式。由于 save 是同步的，所以可以保证数据一致性，而 bgsave 则不能。save 导致 redis 同步阻塞

缺点：在 fork 时，需要增加内存服务器开销，因为当内存不够时，将使用**虚拟内存**，导致阻塞 Redis 运行。

老版本 Redis 无法兼容新版本 RDB。

Redis AOF：

AOF保存的是命令，但是随着时间的增长，命令会越来越多，AOF文件也会越来越大，所以需要对AOF文件进行重写，

AOF的重写机制：

Fork子进程

主进程会写到2个缓冲区，一个是原有的 “AOF 缓存区”，一个是专门为子进程准备的 “AOF 重写缓冲区”；

子进程写到到新的 AOF 文件中，批量的，默认 32m(不能太多，否则会导致硬盘阻塞)；写完后通知主进程。(子进程对原AOF文件进行操作，删除一些已过期的Key，无效的操作，然后创建新的AOF文件)

主进程把“AOF 重写缓冲区”的数据写到新 AOF 文件(重写后的命令 + 新的命令)中。

将新的 AOF 文件替换老文件。

为什么需要两个缓存区：保证重写过程中写命令不丢失。

AOF缓存区作用是保证AOF可用，AOF重写缓存区保证重写可用。这和AOF缓存策略有关，比如缓存策略是always，AOF缓存区就会一直是空的。

AOF缓存策略：

Always：调用系统 fsync 函数，直到同步到硬盘返回；严重影响redis性能。写操作不会到缓存区，而是直接写到硬盘。

everysec：先调用 OS write 函数， 写到缓冲区，然后 redis 每秒执行一次 OS fsync 函数。

no: 只执行 write OS 函数，具体同步硬盘策略由 OS 决定；不推荐，数据不安全，容易丢失数据。

AOF优于RDB加载

并发是指宏观上在一段时间内能同时运行多个程序，而并行则指同一时刻能运行多个指令。

虚拟技术把一个物理实体转换为多个逻辑实体

时分复用：CPU获取时间片

空分复用：页面置换算法

批处理系统没有太多的用户操作，在该系统中，调度算法目标是保证吞吐量和周转时间

先来先服务 first-come first-serverd（FCFS）

短作业优先 shortest job first（SJF）

最短剩余时间优先 shortest remaining time next（SRTN）

交互式系统有大量的用户交互操作，在该系统中调度算法的目标是快速地进行响应。

时间片轮转

优先级调度

多级反馈队列：

一个进程需要执行 100 个时间片，如果采用时间片轮转调度算法，那么需要交换 100 次。

多级队列是为这种需要连续执行多个时间片的进程考虑，它设置了多个队列，每个队列时间片大小都不同，

进程在第一个队列没执行完，就会被移到下一个队列。

每个队列优先权也不同，最上面的优先权最高。因此只有上一个队列没有进程在排队，才能调度当前队列上的进程。

可以将这种调度算法看成是时间片轮转调度算法和优先级调度算法的结合。