目录

[Springboot 1](#_Toc23250528)

[Spring 3](#_Toc23250529)

[Sql 3](#_Toc23250530)

[java 7](#_Toc23250531)

[数据结构： 13](#_Toc23250532)

[JAVA异常 20](#_Toc23250533)

[springMVC原理 22](#_Toc23250534)

[JDBC步骤： 22](#_Toc23250535)

[JAVA反射机制： 23](#_Toc23250536)

[浅拷贝和深拷贝 24](#_Toc23250537)

[Get和POST，HTTP和HTTPS 25](#_Toc23250538)

[JVM 28](#_Toc23250539)

[JDK、JRE、JVM的区别联系 29](#_Toc23250540)

[JVM内存模型，内存结构 29](#_Toc23250541)

[对象的创建过程 33](#_Toc23250542)

[类的生命周期 34](#_Toc23250543)

[加载： 34](#_Toc23250544)

[连接： 34](#_Toc23250545)

[初始化： 35](#_Toc23250546)

[使用： 36](#_Toc23250547)

[卸载： 37](#_Toc23250548)

[总结： 37](#_Toc23250549)

[JVM垃圾回收器 38](#_Toc23250550)

[线程 45](#_Toc23250551)

[消息队列 53](#_Toc23250552)

[锁 57](#_Toc23250553)

[设计模式 59](#_Toc23250554)

[Git 65](#_Toc23250555)

[Redis 69](#_Toc23250556)

[Elasticsearch 69](#_Toc23250557)

[springCloud 70](#_Toc23250558)

# Springboot

**简介：**传统的spring应用程序需要配置大量的xml文件，而springboot只需极少的配置，就可以快速获得一个正常运行的spring应用程序，而且这些配置使用的都是注解的行驶，不需要再配置xml。

@SpringBootApplication：

相当于@Configuration+@EnableAutoConfiguration+@ComponentScan

@ Configuration：

@Configuration和@Bean一起使用，可以创建一个简单的spring配置类，可以用来替代相应的xml配置文件。

@Configuration的注解类标识这个类可以使用spring IOC容器作为bean定义的来源。@Bean注解告诉spring，一个带有@bean的注解方法将返回一个对象，该对象应该被注册为在spring应用程序上下文中的bean。

@EnableAutoConfiguration：

能够自动配置spring的上下文，试图猜测和配置你想要的bean类，通常会自动根据你的类路径和你的bean定义自动配置。·

@ComponentScan：

会自动扫描指定包下的全部标有@Component的类，并注册成bean，包括@Component下的子注解@Service，@Repository，@Controller

@Autowired：

@Resource和@Autowired都是做bean的注入时使用，按照类型装配依赖对象，默认情况下它要求依赖对象必须存在，如果允许null值，可以设置它的required属性为false。如果我们想使用按照名称来装配，可以结合@Qualifier注解一起使用。

@Resource：

默认按照ByName自动注入，有j2ee提供，有两个重要的属性，name和type，而spring将@Resource注解的name属性解析为bean的名字，而type属性解析为Bean的类型。所以，如果使用name属性，则使用Byname的自动注入策略，而使用type属性时则使用Bytype自动注入。如果既不指定name也不制定type属性，这时将通过反射机制使用byname的自动注入策略。

@ResponseBody：

表示该方法的返回结果直接写入HTTP response body中，一般在异步获取数据时使用，用于构建RESTful的api。在使用@RequestMapping后，返回值通常解析为跳转路径。加上@RespouseBody后返回结果不会被解析为跳转路径，会直接返回json数据。

@Controller：

用于定义控制器类，在spring项目中由控制器负责将用户发来的url请求转发到对应的服务接口（service层），一般这个注解在类中，通常方法需要配合注解@RequestMapping。

@RestController：

用于标注控制层组件，@ResponseBody和@Controller的合集。

@Repository：

用于将数据访问层（Dao层）的类标示为Spring Bean。具体只需将该注解标注在DAO类上即可。

# Spring

**简介：**spring是一套java开发框架，框架的作用是为了减少代码的冗余和模块之间的耦合，使代码逻辑更加清晰，主要是用了AOP（面向切面编程）和IOC（控制反转）容器的思想。

**AOP：**面向切面编程，主要实现的目的是针对业务处理过程中的切面进行提取，它所面对的处理过程中某个步骤或阶段，以获得逻辑过程中各部分之间低耦合性的隔离效果。

**IOC：**控制反转，是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。其中最常见的方式叫做依赖注入（DI），还有一种方式叫“依赖查找”。通过控制反转，对象在被创建的时候，由一个调控系统内所有对象的外界实体将其所依赖的对象已用传递给它。也可以说，依赖被注入到对象中。

Class A 中用到了Class B 的对象b，一般情况下，需要在A的代码中显试的new一个B的对象。而采用依赖注入技术之后，A的代码只需要定义一个私有的B对象，不需要直接new来获得这个对象，而是通过相关的容器控制程序来将B对象在外部new出来并注入到A类的引用中。而具体获取的方法、对象被获取时的状态由配置文件来指定。

**OOP：**面向对象编程，它将数据及对数据的操作行为放在一起，作为一个相互依存、补课分割的整体—对象。对于相同类型的对象进行分类、抽象后，得出共同的特征而形成了类。

Spring Core：核心类库，提供IOC服务。

Spring Context：提供框架式的Bean访问方式，以及企业级功能；

Spring AOP：AOP服务；

Spring Dao：对JDBC的抽象，简化了数据访问异常的处理；

Spring ORM：对现有的ORM框架的支持；

Spring Web：提供了基本的面向Web的综合特性，例如多文件上传；

Spring Mvc：提供面向Web应用的Model-View-Controller实现。

# Sql

存储过程：

定义：存储过程是一组为了完成特定功能的SQL语句集合，经编译后存储在服务器端的数据库中，利用存储过程可以加速SQL语句的执行。

存储过程分为系统存储过程和自定义存储过程。

系统存储过程在master数据库中，但是在其他的数据库中可以直接调用，并且在调用时不必在存储过程前加上数据库名，因为在创建一个新数据库时，系统存储过程在新的数据库中会自动创建。

自定义存储过程，由用户创建并能完成某一特定功能的存储过程，存储过程即可以有参数又有返回值，但是它与函数不同，存储过程的返回值只是指明执行是否成功。

并不能像函数那样被直接调用，只能利用execute来执行存储过程。

存储过程的优点：

提高应用程序的通用性和可移植性：存储过程创建后，可以在程序中被多次调用，而不必重新编写该存储过程的SQL语句。并且数据库专业人员可以随时对存储过程进行修改，且对程序源代码没有影响，这样就极大的提高了程序的可移植性。

可以更有效的管理用户操作数据库的权限：在Sql server数据库中，系统管理员可以通过对执行某一存储过程的权限进行限制，从而实现对相应的数据访问进行控制，避免非授权用户对数据库的访问，保证数据的安全。

可以提高SQL的速度，存储过程是编译过的，如果某一个操作包含大量的SQL代码或分别被执行多次，那么使用存储过程比直接使用单条SQL语句执行速度快的多。

减轻服务器的负担，当用户的操作是针对数据库对象的操作时，如果使用单条调用的方式，那么网络上还必须传输大量的SQL语句，如果使用存储过程，则直接发送过程的调用命令即可，降低了网络的负担。

**Where和having的区别**

Where是一个约束声明，在查询数据库的结果返回之前对数据库中的查询条件进行约束，即在结果返回之前起作用，且where后面不能使用聚合函数。

Having是一个过滤声明，所谓过滤是在查询数据库的结果返回之后进行过滤，即在结果返回之后起作用，并且having后面可以使用聚合函数

Where是group by之前进行条件筛选，而having是group by之后的进行条件筛选

**Group by注意**

指定group by时，选择列表中任一非聚合表达式内的所有列都应包含在group by列表中，或者group by表达式必须与选择列表表达式完全匹配。否则会报出：“因为该列没有包含在聚合函数或 GROUP BY 子句中。”的错误

**多条插入**

Insert into 目标表 (字段1,字段2,字段3) select 字段1,字段2,字段3 from 源表 where 条件

Insert into 目标表 select ‘xx’,’xxx’,’xxx’ union all select ‘xx’,’xxx’,’xxx’

**Sql中begin，end，go的用法**

Go向sql server发出一批Transact-SQL语句结束的信号。Go是把t-sql分批次执行。一步成功了才会执行下一步，即一步一个go。

BEGIN和END语句用于将多个T-SQL组合成一个逻辑块，例：

IF (@@ERROR <> 0)

SET @ErrorSaveVariable = @@ERROR

如果 @@ERROR 为 0，则仅跳过 SET 语句。

用 BEGIN 和 END 语句可以使 IF 语句在计算结果为 FALSE 时跳过语句块：

IF (@@ERROR <> 0)

BEGIN

SET @ErrorSaveVariable = @@ERROR

PRINT 'Error encountered, ' +

CAST(@ErrorSaveVariable AS VARCHAR(10))

END

**查询成绩比该课程平均成绩低的同学的成绩表**

select \* from SCORE sc1 where DEGREE<(select AVG(DEGREE) from SCORE sc2 where sc1.CNO=sc2.CNO)

**查询选修编号为“3-105”课程且成绩至少高于选修编号为“3-245”的同学的cno,sno和degree，并安degree从高到低排序。**

select cno,sno,DEGREE from SCORE sc1 where CNO='3-105' and DEGREE>(select DEGREE from SCORE sc2 where CNO='3-245' and sc1.SNO=sc2.SNO)

**查询所有教师和同学的name,sex,birthday**

select sname,ssex,sbirthday from STUDENT union all select tname,tsex,tbirthday from TEACHER

**查询Score表中至少有5名学生选修的并以3开头的课程的平均分数。**

select AVG(degree) from SCORE group by SNO having COUNT(1)>5 and CNO like '3%'; --12

**查询至少有2名男生的班号。**

select sclass,COUNT(1) from STUDENT st where st.SSEX='男' group by SCLASS having COUNT(1)>=2

**查询所有的存储过程**

SELECT name, definition

FROM sys.sql\_modules AS m

INNER JOIN sys.all\_objects AS o ON m.object\_id = o.object\_id

WHERE o.[type] = 'P'

**查看表结构**

sp\_help 目标表

**存储过程**

**1、**

if (exists (select \* from sys.objects where name = 'stu2'))

--如果存在名为stu2的存储过程

drop proc stu2 --则删除stu2存储过程

go

create proc stu2 --创建存储过程stu2

@sname varchar(50) --声明全局变量

as

begin

select \* from student s where s.stuname=@sname; --条件查询

end

go

exec stu2 '王男' --执行存储过程stu2

**2、**

if (exists (select \* from sys.objects where name='stu3'))

drop proc stu3

go

create proc stu3

@sname varchar(50)='王男', --默认参数

@result varchar(8) output --输出参数

as

begin

if(select COUNT(1) from student s where s.stuname=@sname)>0

set @result='存在'

else

set @result='不存在'

end

go

declare @result varchar(8)

exec stu3 '王男',@result output

print @result

go

**3、**

if (exists (select \* from sys.objects where name='stu5'))

drop proc stu5

go

create proc stu5

as

declare @sname varchar(50) --局部变量

set @sname='杨幂'

begin

select \* from student s where s.stuname=@sname;

end

go

exec stu5

**4、**

if(exists (select \* from sys.objects where name='stu1'))

drop proc stu1

go

create proc stu1

@sno varchar(8),

@result varchar(20) output

as

declare @allavg decimal(18,2)

declare @curAvg decimal(18,2)

begin

set @allavg = (select AVG(co.coursescore) from student st left join course co on st.stuno=co.stuno)

print '全部平均分'+convert(varchar(18),@allavg)

set @curAvg = (select Avg(co.coursescore) from student st left join course co on st.stuno=co.stuno where st.stuno=@sno)

print '个人平均分'+convert(varchar(18),@curAvg)

if(@allavg>@curAvg )

begin

set @result='低于平均水平'

end

else

begin

set @result='高于平均水平'

end

end

go

declare @result varchar(20)

exec stu1 '02',@result output

print @result

# java

八大基本类型：

Byte short int long float double char Boolean

修饰符：

Public：表明该成员变量或方法对所有类或对象都是可见的，所有类或对象都可以直接访问。 使用对象：类、接口、变量、方法。

Protected：表明该成员变量或方法对自己及其子类是可见的，即自己和（同包和不同包）子类具有权限访问 使用对象：变量、方法。

Default：表明该成员变量或方法只有自己和与其同一包内的类可见。使用对象：类、接口、变量、方法。

Private：表明该成员变量或方法是私有的，只有当前类对其具有访问权限。使用对象：变量、方法。



**作用域**

在计算机程序中，声明在不同地方的变量具有不同的作用域，例如局部变量、全局变量等。在Java语言中，作用域是由花括号的位置决定的，它决定了其定义的变量名的可见性与生命周期。

在Java语言中，变量的类型主要有3种：成员变量、静态变量和局部变量。

* **成员变量：**类的成员变量的作用域与类的实例化对象的作用范围相同，当类被实例化时，成员变量就会在内存中分配空间并初始化，直到这个实例化对象的生命周期结束时，成员变量的生命周期才结束。
* **静态变量：**被static修饰的成员变量称为静态变量或者全局变量，与成员变量不同的是，静态变量不依赖于特定的实例，而是被所有的实例所共享，也就是说，只要一个类被加载，JVM就会给类的静态变量分配存储空间。因此，就可以通过类名和变量名来访问静态变量。
* **局部变量：**局部变量的作用域与可见性为它所在的花括号内。

创建对象

对象是根据类创建的。在JAVA中，使用关键字new来创建一个新的对象。创建对象需要以下三步：

1. 声明：声明一个对象，包括对象名称和对象类型
2. 实例化：使用关键字new来创建一个对象。
3. 初始化：使用new创建对象时，会调用构造方法初始化对象。

关键字：

Static：

用来修饰类方法和类变量

静态变量：static关键字用来声明独立于对象的静态变量，无论一个类实例化多少对象，它的静态变量只有一份拷贝。静态变量也被称为类变量。局部变量不能被声明为静态变量。

静态方法：static关键字用来声明独立于对象的静态方法。静态方法不能使用类的非静态变量。静态方法从参数列表得到数据，然后计算这些数据。

Final修饰符：用来修饰类、方法和变量，final修饰的类不能够被继承，修饰的方法不能被继承类重新定义，修饰的变量为常量。是不可修改的。

Final变量：变量一旦赋值后，不能被重新赋值。被final修饰的实例变量必须显式指定初始值。（final修饰符通常和static修饰符一起用来创建类变量）

Final方法：类中的final方法可以被子类继承，但是不能被子类修改。声明final方法的主要目的是防止该方法的内容被修改。

Final类：不能被继承，没有类能够继承final类的任何特性。

Abstract修饰符：用来创建抽象类和抽象方法。

抽象类：抽象类不能用来实例化对象，声明抽象类的唯一目的是为了将来对该类进行扩充。一个类不能同时被abstract和final修饰。如果一个类包含抽象方法，那么该类一定要声明为抽象类，否则将出现编译错误。

抽象类可以包含抽象方法和非抽象方法。

抽象方法：抽象方法是一种没有任何实现的方法，该方法的具体实现由子类提供。

抽象方法不能被声明final和static。

任何继承抽象类的子类必须实现父类的所有抽象方法，除非该子类也是抽象类。

如果一个类包含若干个抽象方法，那么该类必须声明为抽象类。抽象类可以不包含抽象方法。

Synchronized和volatile：主要用于线程的编程。

Synchronized：它声明的方法同一时间只能被一个线程访问。可以应用于四个访问修饰符。

Transient：序列化的对象包含被transient修饰的实例变量时，java虚拟机跳过该特定的变量。该修饰符包含在定义变量的语句中，用来预处理类和变量的数据类型。

Volatile修饰符：修饰的成员变量在每次被线程访问时，都强制从共享内存中重新读取该成员变量的值。而且，党成员变量发生变化时，会强制线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

Break：跳出整个循环。

Continue：跳出本次循环。

Number类：Integer Long Byte Double Float Short 都是Number的子类

StringBuilder和StringBuffer：它们之间最大的不同在于StringBuilder的方法不是线程安全的（不能同步访问）。由于StringBuilder相较于StringBuffer有速度优势，所以多数情况下建议使用StringBuilder类。然而在要求线程安全的情况下，则必须使用StringBuffer类。

自定义异常：1、所有的异常都必须是Throwable的子类。

2、如果希望写一个检查性异常类，则需要继承Exception类。

3、如果你想写一个运行时异常类，则需要继承RuntimeException类。

继承：

1. 子类拥有父类非private的属性、方法。
2. 子类可以拥有自己的属性和方法，即子类可以对父类进行扩展。
3. 子类可以用自己的方式实现父类的方法。
4. Java的继承是单继承，可以多重继承。
5. 提高了类之间的耦合性（继承的缺点，耦合度高就会造成代码之间的联系越紧密，代码独立性越差）

继承关键字：extends，implements

Extends：一个子类只能拥有一个父类，所以extends只能继承一个类。

Implements：使用implements关键字可以变相的使java具有多继承的特性，使用范围为类继承接口的情况，可以同时继承多个接口（接口跟接口之间采用逗号分隔）。

Super：通过super关键字来实现对父类成员的访问，用来引用当前对象的父类。

This：指向自己的引用。

Final：final关键字生命类可以把类定义为不能继承的，即最终类；或者用于修饰方法，该方法不能被子类重写。

注：子类是不继承父类的构造器（构造方法或者构造函数）的。它只是调用（隐式或显式）。如果父类的构造器带有参数，则必须在子类的构造器中显式地通过super关键字调用父类的构造器并配以适当的参数列表。如果父类构造器没有参数，则在子类的构造器中不需要使用super关键字调用父类的构造器，系统会自动调用父类父类饿的无参构造器。

重写和重载：

1. 重写（Override）：

重写是子类对父类的允许访问的方法的实现过程进行重新编写，返回值和形参都不能改变，即外壳不变，核心重写。

重写的好处在于子类可以根据需要，定义特定于自己的行为。也就是说子类能够根据需要实现父类的方法。

重写方法不能抛出新的检查异常或者比被重写方法申明更加宽泛的异常。

在面向对象原则里，重写意味着可以重写任何现有方法。

1. 重载（Overload）：

重载是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。

每个重载的方法都必须有一个独一无二的参数类型列表。

最常用的地方就是构造器的重载。

注：方法重载是一个类的多态性表现，而方法重写是子类与父类的一种多态性表现。

多态：

多态是同一个行为具有多个不同表现形式或形态的能力。

多态就是同一个接口，使用不同的实例而不执行不同操作。

优点：1、消除类型之间的耦合关系

2、可替换性

3、可扩充性

4、接口性

5、灵活性

6、简化性

多态存在的三个必要条件：1、继承。2、重写。3、父类引用指向子类对象。

（例：Parent P = new Child()）

多态的实现方式：1、重写。2、接口。3、抽象类和抽象方法。

抽象类：使用abstract class 来定义抽象类

概念：在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

抽象类除了不能实例化对象之外，类的其他功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。

由于抽象类不能实例化对象，所以抽象类必须被继承才能被使用。

父类包含了子类集合的常见的方法，但是由于父类本身是抽象的，所以不能使用这些方法。

在java中抽象类表示的是一种继承关系，一个类只能继承一个抽象类，而一个类却可以实现多个接口。

抽象方法：

如果你想设计这样一个类，该类包含一个特别的成员方法，该方法的具体实现由它的子类确定，那么你可以在父类中声明该方法为抽象方法。

Abstract关键字同样可以用来声明抽象方法，抽象方法只包含一个方法名，而没有方法体。

抽象方法没有定义，方法名后面直接跟一个分号，而不是花括号。

继承抽象方法的子类必须重写该方法。

总结：

1. 抽象类不能被实例化，如果被实例化，就会报错，编译无法通过。只有抽象类的非抽象子类可以创建对象。
2. 抽象类中不一定包含抽象方法，但是有抽象方法的类必定是抽象类。
3. 抽象类中的抽象方法只是声明，不包含方法体，就是不给出方法的具体实现也就是方法的具体功能。
4. 构造方法，类方法（用static修饰的方法）补鞥呢声明为抽象方法。
5. 抽象类的子类必须给出抽象类中的抽象方法的具体实现，除非该子类也是抽象类。

接口：

概念：在Java编程语言中是一个抽象类型，是抽象方法的集合，接口通常以interface来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。

接口不是类，编写接口的方式和类很相似，但是他们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法。

除非实现接口的类是抽象类，否则该类要定义接口中的所有方法。

接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。另外，在JAVA中，接口类型可以用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。

**Java代码执行顺序**

class HelloA {

public HelloA(){

System.out.println("Hello A!父类构造方法");

}

{ System.out.println("i'm A class.父类非静态代码块");

}

static{

System.out.println("static A 父类静态代码块");

}

}

class HelloB extends HelloA {

public HelloB(){

System.out.println("Hello B! 构造方法");

}

{

System.out.println("i'm B class.非静态代码块");

}

static{

System.out.println("static B 静态代码块");

}

public static void main(String[] args) {

System.out.println("---start---");

new HelloB();

new HelloB();

System.out.println("---end---");

}

}

**打印结果**

父类静态代码块

子类静态代码块

---start---

父类非静态代码快

父类构造方法

子类非静态代码块

子类构造方法

父类非静态代码快

父类构造方法

子类非静态代码块

子类构造方法

---end---

**面试通过**

# 数据结构：

1、枚举（Enumeration）。

2、位集合（BitSet）。

3、向量（Vector）

4、栈（Stack）

5、字典（Dictionary）

6、哈希表（Hashtable）

7、属性（Properties）

Set和List的区别

1. Set接口实例存储的是无序的，不重复的数据。List接口实例存储的是有序的，可以重复的元素。
2. Set检索效率低下，删除和插入效率高，插入和删除不会引起元素位置改变
3. List和数组类似，可以动态增长，根据实际存储的数据的长度自动增长List的长度。查找元素效率高，插入删除效率低，因为会引起其他元素位置改变。

ArrayList和LinkedList的区别

1、ArrayList是数组队列，相当于动态数组；LinkedList为双向链表结构，也可作为堆栈、队列、双端队列

2、当随机访问List时（Get和Set操作），ArrayList比LinkedList效率更高，因为LinkedList是线性的数据存储方式，所以需要移动指针从前往后依次查找。

3、当对数据进行增加和删除操作时（add和remove操作），LinkedList比ArrayList的效率更高，因为ArrayList是数组，所以在其中进行增删操作时，会对操作点之后所有数据的下标索引造成影响，需要进行数据的移动。

4、从利用效率来看，ArrayList自由性较低，因为它需要手动的设置固定大小的容量，但是它的使用比较方便；而LinkedList自由性较高，能够动态的随数据量的变化而变化，但是它不便于使用。

5、ArrayList主要控件开销在于需要在List列表预留一定空间；而LinkList主要控件开销在于需要存储结点信息以及结点指针信息。

集合框架：

1. Collection是一个借口，是高度抽象出来的集合，它包含了集合的基本操作和属性。Collection包含了List和Set两大分支。
2. List是一个有序的队列，每一个元素都有它的索引，第一个元素的索引值是0。List的实现类有LinkedList，ArrayList，Vector，Stack。
3. Set是一个不允许有重复元素的集合。Set的实现类有HashSet和TreeSet。HashSet依赖于HashMap，它实际上是通过HashMap实现的；TreeSet依赖于TreeMap，它实际上是通过TreeMap实现的。
4. Map是一个映射接口，即key-value键值对。Map中的每一个元素包含一个Key和key对应的value。AbstractMap是个抽象类，它实现了Map接口中的大部分API。而HashMap，TreeMap，WeakHashMap都是继承于AbstractMap。Hashtable虽然继承于Dictionary。但它实现了Map接口。
5. Iterator是遍历集合的工具，我们通常用Iterator迭代器来遍历集合。我们说Collection依赖于Iterator，是因为Collection的实现类都要实现iterator()函数，返回一个Iterator对象，ListIterator是专门为遍历List而存在的。
6. Enumeration，作用和Iterator一样，也是遍历集合；但是Enumeration的功能要比Iterator少，Enumeration只能在Hashtable，Vector，stack中使用。
7. Arrays和Collections。是操作数组、集合的两个工具类。
8. List接口：  
   List接口继承于Collection接口，它可以定义一个允许重复的有序集合。可以通过索引来访问List中的元素，类似于java的数组。

List接口为Collection直接接口。List所代表的是有序的Collection，即它用某种特定的插入顺序来维护元素顺序。用户可以对列表中每个元素的插入位置进行精确地控制，同时可以根据元素的整数索引访问元素，并搜素列表中的元素。

实现List接口的集合有：ArrayList、LinkedList、Vector、Stack

1. ArrayList

ArrayList是一个动态数组，也是我们最常用的集合。它允许任何复合规则的元素插入甚至包括null。每一个ArrayList都有一个初始容量（10）,该容量代表了数组的大小。随着容器中的元素不断增加，容器的大小也会随着增加。在每次向容器中增加元素的同事都会进行容量检查，当快溢出时，就会进行扩容操作，所以如果我们明确所插入元素的多少，最好指定一个初始容量值，避免过多的进行扩容操作而浪费时间、效率。

Size，isEmpty，get，set，iterator和listIterator操作都以固定时间运行。Add操作以分摊的固定时间运行，也就是说，添加n个元素需要O(n)时间

ArrayList擅长随机访问，同时ArrayList是非同步的。

1. LinkedList

同样实现List解耦的LinkedList与ArrayList不同，ArrayList是一个动态数组，而LinkedList是一个双向链表。所以它除了有ArrayList的基本操作方法还额外提供了get，remover,insert方法在LinkedList的首部或尾部。

由于实现的方式不同，LinkedList不能随机访问，它所有的操作都是按照双重链表的需要执行。在列表中索引的操作将从开头或结尾遍历列表（从靠近指定索引的一端）。这样做的好处就是可以通过较低的代价在List中进行插入和删除操作。

与ArrayList一样，LinkedList也是非同步的。

1. Vector：

与ArrayList相似，但是Vector是同步的。所以说Vector是线程安全的动态数组。它的操作与ArrayList几乎一样。

1. Stack

Stack继承自Vector，实现一个后进先出的堆栈。Stack提供5个额外的方法使得Vector得以被当做堆栈使用。基于push和pop方法，还有peek方法得到栈顶的元素，empty方法测试堆栈是否为空，search方法检测一个元素在堆栈中的位置。Stack刚创建后事空栈。

1. Set

Set是一种不包括重复元素的Collection。它维持它自己的内部排序，所以随机访问没有任何意义。与List一样，它同样允许Null的存在但是仅有一个。由于Set接口的特殊性，所有传入set集合中的元素都必须不同。Set接口有三个具体实现类，分别是散列集HashSet、链式散列集LinkedHashSet和树形集TreeSet。

Set是一种不包含重复的元素的Collection，无序，即任意的两个元素都不相同，Set最多只有一个Null元素。需要注意的是：虽然Set中元素没有顺序，但是元素在set中的位置是由该元素的HashCode决定的，其具体位置其实是固定的。

1. HashSet：是一个没有重复元素的集合。它是由HashMap实现的，不保证元素的顺序（这里所说的没有顺序是指：元素插入的顺序与输出的顺序不一致），而且HashSet允许使用null元素。HashSet是非同步的，如果多个线程同时访问一个HashSet，而其中至少一个线程修改了该set，那么它必须保持外部同步。HashSet按Hash算法来存储集合的元素，因此具有很好的存取和查找查找性能。
2. LinkedHashSet：LinkedHashSet继承自HashSet，其底层是基于LinkedHashMap来实现的，有序，非同步。LinkedHashSet集合同样是根据元素的HashCode值来决定元素的存储位置，但是它同时使用链表维护元素的次序。这样使得元素看起来像是以插入顺序保存的，也就是说，当遍历该集合时候，LinkedHashSet将会以元素的添加顺序访问集合的元素。
3. TreeSet：TreeSet是一个有序集合，其底层是基于TreeMap实现的，非线程安全。TreeSet可以确保集合元素处于排序状态。TreeSet支持两种排序方式，自然排序和定制排序，其中自然排序为默认的排序方式。当我们构造TreeSet时，若使用不带参数的构造函数，则TreeSet的使用自然比较强；若用户需要使用自定义的比较器，则需要使用带比较器的参数。

注：TreeSet集合不是通过HashCode和equals函数来比较元素的，它是通过compare或者compareTo函数来判断元素是否相等，compare函数通过判断两个对象的id，相同的id判断为重复元素，不会被加入到集合中。

Map接口

Map与List、Set接口不同，它是由一系列键值对组成的集合，提供了key到value的映射。同时它也没有继承collection。在Map中它保证了key与value之间的一一对应关系。也就是说一个key对应一个value，所以它不能存在相同的key值，当然value值可以相同。

1. HashMap：以哈希数据结构实现，查找对象时通过哈希函数计算其位置，它是为快速查询而设计的，其内部定义了一个hash表数组，元素会通过哈希转换函数将元素的哈希哈希地址转换成数组中存放的索引，如果有冲突，则使用散列链表的形式将所有相同哈希地址的元素串起来，可以通过查看HashMap.Entry的源码它是一个单链表结构。
2. LinkedHashMap：是HashMap的一个子类，它保留插入的顺序，如果需要输出的顺序和输入时的相同，那么就选用LinkedHashMap。

LinkedHashMap是Map接口的哈希表和链接列表实现，具有可预知的迭代顺序。此实现提供所有可选的映射操作，并允许使用Null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

1. TreeMap：是一个有序的key-value集合，非同步，基于红黑树实现，每一个key-value结点作为红黑树的一个节点。TreeMap存储时会进行排序的，会根据key来对key-value键值对进行排序，其中排序方式也是分为两种，一种是自然排序，一种是定制排序。

自然排序：TreeMap中所有的key必须实现Comparable接口，并且所有的key都应该是同一个类的对象，否则会报ClassCastExceptioin异常。

定制排序：定义TreeMap时，创建一个comparator对象，该对象对所有的treeMap中所有的key值进行排序，采用采用idingzhi排序的时候不需要TreeMap中所有的key必须实现Comparable接口。

TreeMao判断两个元素相等的标准：两个Key通过compareTo()方法返回0，则认为这两个key相等。

如果使用自定义的类来作为TreeMap中的key值，且想让TreeMap能够良好的工作，则必须重写子弟定义类的equals()方法，TreeMap中判断相等的标准是：两个key通过equals()方法返回true，并且通过compareTo()方法比较应该返回0。

对比：

Set接口：Set不允许包含相同的元素，如果试图把两个相同元素加入同一个集合中，add方法返回false。Set判断两个对象相同不是使用==运算符，而是根据equals方法。也就是说，只要两个对象用equals方法比较返回true，Set就不会接受这两个对象。

HashSet：当向HashSet集合中存入一个元素时，HashSet会调用该对象的HashCode方法得到该对象的HashCode值，然后根据hashCode值来决定该对象在HashSet中存储的位置。

注：如果要把一个对象放入HashSet中，重写该对象对应类的equals方法，也应该重写其hashCode()方法。其规则是如果两个对象通过equals方法比较返回true时，其hashCode也应该相同。另外，对象中用作equals比较标准的属性，都应该用来计算hashCode值。

LinkedHashSet：LinkedHashSet集合同样是根据元素的hashCode值来决定元素的存储位置，但是它同时使用链表维护元素的次序。这样使得元素看起来像是以插入顺序保存的，也就是说，当遍历该集合时候，LinkedHashSet将会以元素的添加顺序访问集合的元素。

LinkedHashSet在迭代访问Set中的全部元素时，性能比HashSet好，但是插入时性能稍微逊色于HashSet。

TreeSet类：TreeSet是SortedSet接口的唯一实现类，TreeSet可以确保集合元素处于排序状态。TreeSet支持两种排序方式，自然排序和定制排序，其中自然排序为默认的排序方式。向TreeSet中加入的应该是同一个类的对象。

TreeSet判断两个对象不相等的方式是两个对象通过equals方法返回false，或者通过CompareTo方法比较没有返回0。

自然排序：自然排序使用使用要排序元素的CompareTo(Object obj)方法来比较元素之间大小关系，然后将元素按照升序排序。

定制排序：自然排序是根据集合元素的大小，以升序排列，如果要定制排序，应该使用Comparator接口，实现int compare(T o1, T o2)方法。

Iterator和LisIterator区别

我们在使用List，Set的时候，为了实现对其数据的遍历，我们经常使用到了Iterator（迭代器）。使用迭代器，你不需要干涉其遍历的过程，只需要每次取出一个你想要的数据进行处理就可以了。但是在使用的时候也是有不同的。List和Set都有Interator来获取其迭代器。对List来说，你也可以通过ListIterator取得其迭代器，两种迭代器在有些时候是不能通用的，Iterator和ListIterator主要区别在一下方面：

1. ListIte有rator有add方法，可以想List中添加对象，而Iterator不能
2. ListIterator和Iterator都有hasNext和next方法，可以实现顺序向后遍历，但是ListIterator有hasPrevious和previous方法，可以实现逆向（顺序向前）遍历。Iterator就不可以。
3. ListIterator可以定位当前的索引位置，nextIndex()和previousIndex()可以实现。Iterator没有此功能。
4. 都可以实现删除对象，但是ListIterator可以实现对象的修改，set()方法可以实现。Iterator仅能遍历，不能修改。

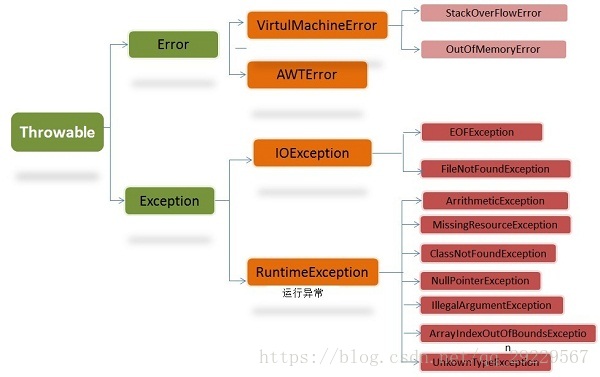
Collection和Collections区别

1. java.util.Collection是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行进行基本操作的通用方法。Collection接口在java类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有List与Set。
2. java.util.Collections是一个包装类（工具类/帮助类）。它包含有各种有关集合操作的静态多态方法。此类不能示例化，就像一个工具类，用于对集合中元素进行排序、搜索以及线程安全等各种操作，服务于java的Collection框架。

**哈希算法：**

**哈希碰撞：**

# JAVA异常



**Throwable：**有两个重要的子类：Exception（异常）和Error（错误），二者都是Java异常处理的重要子类，各自都包含大量子类。异常和错误的区别是：异常能被程序本身处理，错误无法处理。

**Error（错误）：**是程序无法处理的错误，表示运行应用程序中较严重的问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时JVM（Java虚拟机）出现的问题。例如，Java虚拟机运行错误（Virtual MachineError），当JVM不再有继续执行操作所需的内存资源时，将出现OutOfMemoryError。

这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如Java虚拟机运行错误（Virtual MachineError）、类定义错误（NoClassDefFoundError）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在Java中，错误通过Error的子类描述。

**Exception（异常）：**是程序本身可以处理的异常。其中可以分为两大类：

* 运行时异常
* 非运行时异常

**运行时异常：**RunTimeException类及其子类异常，如NullPointerException（空指针异常）、IndexOutOfBoundsException（下标越界异常）等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。

运行时异常的特点是Java编辑器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用try-catch语句捕获它，也没有用throws子句生命抛出它，也会编译通过。

**非运行时异常（编译异常）：**RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。

**异常处理机制：**Java中异常处理机制有，抛出异常、捕获异常

如果一个方法可能会出现异常，但没有能力处理这种异常，可以在方法声明处用throws或throw来抛出异常。

**Throws：**throws语句用在方法定义时声明该方法要抛出的异常类型，如果抛出的是Exception异常类型，则该方法被声明为抛出所有的异常。多个异常可使用逗号分割。

格式如下：

**public void** test() **throws** Exception,ClassNotFoundException{  
 *//****TODO***}

方法名后的Exception1，Exception2…为声明要抛出的异常列表。当方法抛出异常列表的异常时，方法将不对这些类型及其子类类型的异常作处理，而抛向调用该方法的方法，由他去处理。

**Throw：**throw总是出现在方法体中，用来抛出一个Throwable类或者其子类的实例对象异常。

格式如下：

**throw new** IOException();

程序会在throw语句后立即终止，它后面的语句执行不到（除了之后的finally），然后在包含它的所有try块中（可能在上层调用函数中）从里向外寻找含有与其匹配的catch子句的try块。

# springMVC原理

（1）客户端（浏览器）发送请求，直接请求到DispatcherServlet。（前端控制器）

（2）DispatcherServlet根据请求信息调用HandlerMapping，解析请求对应的Handler。

（3）解析到对应的Handler后，开始由HandlerAdapter适配器处理。

（4）HandlerAdapter会根据Handler来调用真正的处理器开处理请求，并处理相应的业务逻辑。

（5）处理器处理完业务后，会返回一个ModelAndView对象，Model是返回的数据对象，View是个逻辑上的View。

（6）ViewResolver会根据逻辑View查找实际的View。

（7）DispaterServlet把返回的Model传给View。

（8）通过View返回给请求者（浏览器）

# JDBC步骤：

1. 注册驱动（仅仅做一次）
2. 建立连接（Connection）
3. 创建运行SQL的语句（Statement）
4. 运行语句，得到结果集。
5. 处理运行结果（ResultSet）
6. 释放资源

# JAVA反射机制：

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能成为java语言的反射机制。

Java的反射机制允许编程人员在对类未知的情况下，获取类相关信息的方式变得更加多样灵活，调用类中相应的方法，是JAVA增强其灵活性与动态性的一种机制。

要想解剖一个类，必须先要获取到该类的字节码文件对象。而解剖使用的就是Class类中的方法，所以先要获取到每一个字节码文件对应的Class类型的对象。



**import** com.example.java.book.model.Student;  
  
*/\*\*  
 \* 获取Class对象的三种方式  
 \* 1、Object -> getClass();  
 \* 2、任何数据类型（包括基本数据类型）都有一个“静态”的class属性  
 \* 3、通过Class类的静态方法：forName(String className)（常用）  
 \*/***public class** Fanshe {  
 **public static void** main(String[] args) {  
 *//第一种方式获取Class对象* Student stu1 = **new** Student();*//new 产生一个Student对象，一个Class对象。* Class stuClass = stu1.getClass();*//获取Class对象* System.***out***.println(stuClass.getName());  
  
 *//第二种方式获取Class对象* Class stuClass2 = Student.**class**;  
 System.***out***.println(stuClass == stuClass2);*//判断第一种方式获取的Class对象和第二种方式获取的是否相同  
  
 //第三种方式获取Class对象* **try**{  
 Class stuClass3 = Class.*forName*(**"com.example.java.book.model.Student"**);*//注意此字符串必须是真实路径，就是带包名的类路径，包名.类名* System.***out***.println(stuClass2 == stuClass3);*//判断三种方式是否获取的是同一个Class对象* }**catch** (ClassNotFoundException e){  
 e.printStackTrace();  
 }  
 }  
}

注：三种方式常用第三种，第一种对象都有了还要反射干什么。第二种需要导入类的包，依赖太强，不导包就抛编译错误。一般都用第三种，一个字符串可以传入也可写在配置文件中等多种方法。

Class clazz = Class.*forName*(**"com.example.java.book.model.Student"**);

获得构造方法：Constructor[] conArray = clazz.getDeclaredConstructors();

获取普通方法：Method[] methods = clazz.getDeclaredMethods();//调用public方法

获取属性：Field[] fields2 = clazz.getDeclaredFields(); //返回所有属性

# 浅拷贝和深拷贝

Java中的数据类型分为基本数据类型和引用数据类型。对于这两种数据类型，在进行赋值操作、用作方法参数或返回值时，会有值传递和引用（地址）传递的差别。

浅拷贝：

1、对于数据类型是基本数据类型的成员变量，浅拷贝会直接进行值传递，也就是将该属性值复制一份给新的对象。因为是两份不同的数据，所以对其中一个对象的该成员变量值进行修改，不会影响另一个对象拷贝得到的数据。

2、对于数据类型是引用数据类型的成员变量，比如说成员变量是某个数组、某个类的对象等，那么浅拷贝会进行引用传递，也就是只是将该成员变量的引用值（内存地址）复制一份给新的对象。因为实际上两个对象的该成员变量都指向同一个实例。在这种情况下，在一个对象中修改该成员变量会影响到另一个对象的该成员变量的值。

深拷贝：

深拷贝对引用数据类型的成员变量的对象图中所有的对象都开辟了内存空间；

而浅拷贝只是传递地址指向，新的对象并没有对引用数据类型创建内存空间。

# Get和POST，HTTP和HTTPS

**1、get和post**

|  |  |  |
| --- | --- | --- |
|  | Get | post |
| url可见性 | 参数url可见 | url参数不可见 |
| 数据传输上 | 通过拼接url进行传递参数 | 通过body体传递参数 |
| 缓存性 | 可以缓存 | 不可以缓存 |
| 后退页面的反应 | get请求页面后退时， 不产生影响 | Post请求页面后退时， 会重新提交请求 |
| 传输数据的大小 | get一般传输数据大小不超过2k-4K （根据浏览器不同，限制不一样，但相差不大） | Post请求传输数据的大小根据 php.ini配置文件设定，可以无限大 |
| 安全性 | 原则上post要比get安全，毕竟传输参数url不可见，但也可以抓包看到信息。所以都不是很安全。 | |

Post用于修改和写入数据。get一般用于搜索排序和筛选之类的操作，目的是资源的获取，读取数据。

**Post请求过程：**

* 浏览器请求tcp连接（第一次握手）
* 服务器应答进行tcp连接（第二次握手）
* 浏览器确认，并发送Post请求头（第三次握手，这个报文比较小，所以http会在此时进行第一次数据发送）
* 服务器返回100 Continue响应
* 浏览器发送数据
* 服务器返回200 ok 响应

**Get请求过程：**

* 浏览器请求tcp连接（第一次握手）
* 服务器应答进行tcp连接（第二次握手）
* 浏览器确认，并发送get请求头和数据（第三次握手）
* 服务器返回200 OK 响应

**2、http和https**

**概念**

**Http：**超文本传输协议是互联网上应用最为广泛的一种网络协议。设计http最初的目的是为了提供一种发布和接受HTML页面的方法。它可以使浏览器更加高效。Http协议是以明文方式发送信息的，如果黑客截取了Web浏览器和服务器之间的的传输报文，就可以直接获得其中的信息。

**Https：**是以安全为目标的Http通道，是Http的安全版。Https的安全基础是SSL。SSL协议位于TCP/IP与各种应用层协议之间，为数据通讯提供安全支持。SSL协议可分为两层：**SSL记录协议，**它建立在可靠的传输协议（如TCP）之上，为高层协议提供数据封装、压缩、加密等基本功能的支持。**SSL握手协议**，它建立在SSL记录协议之上，用于在实际的数据传输开始前，通讯双方进行身份认证、协商加密算法、交换加密密钥等。

**区别**

* Https协议需要到CA申请证书，一般免费证书较少，因而需要一定费用。
* Http是超文本传输协议，信息是明文传输，https则是具有安全性的SSL加密传输协议。
* Http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
* Http的连接很简单，是无状态的。Https协议是由SSL+Http协议构建的可进行加密传输、身份认证的网络协议，比http协议安全（无状态的意思是其数据包的发送、传输和接受都是相互独立的。无连接的是指通信双方都不长久的维持对方的任何信息）

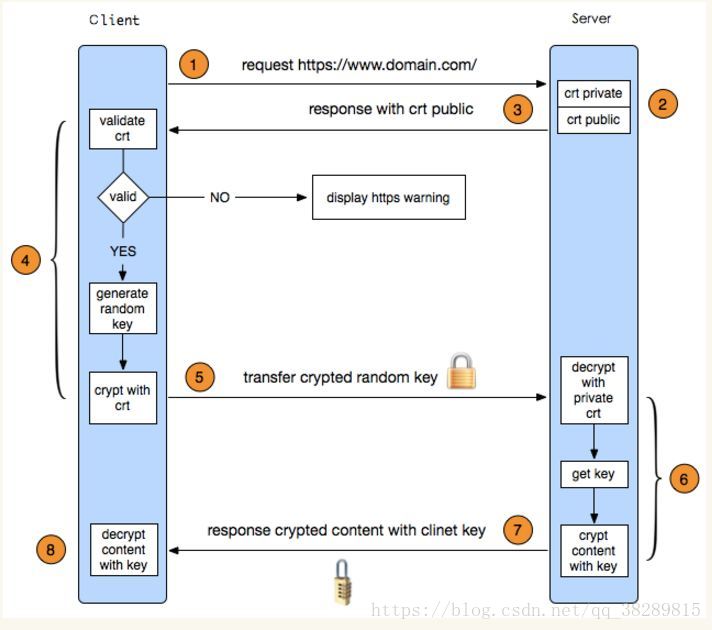
**Https的优点：**

* 使用https协议可认证用户和服务器，确保数据发送到正确的客户机和服务器。
* https 协议是由SSL+Http协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全，可防止数据在传输过程中不被窃取、修改，确保数据的完整性。
* Https是现行架构下最安全的解决方案，虽然不是绝对安全，但它大幅增加了中间人攻击的成本。

**Https的缺点：**

* Https协议握手阶段比较费时，会使页面的加载时间延长。
* Https连接缓存不如Http高效，会增加数据开销，甚至已有的安全措施也会因此而受到影响
* SSL证书通常要绑定IP，不能在同一个IP上绑定多个域名，IPV4资源不可能支撑这个消耗。
* Https协议的加密范围也比较有限。最关键的，SSL证书的信用链体系并不安全，特别是在某些国家可以控制CA根证书的情况下，中间人攻击一样可行。

**https的连接过程**



1. 客户端的浏览器向服务器发送请求，并传送客户端SSL协议的版本号，加密算法的种类，产生的随机数，以及其他服务器和客户端之间通讯所需要的各种信息。
2. 服务器向客户端传送SSL协议的版本号，加密算法的种类，随机数以及其他相关信息，同时服务器还将向客户端传送自己的证书。
3. 客户端利用服务器穿过来的信息验证服务器的合法性，服务器的合法性包括：证书是否过期，发行服务器证书的CA是否可靠，发行者证书的公钥能否正确解开服务器证书的“发行者的数字签名”，服务器证书上的域名是否和服务器的实际域名相匹配。如果合法性验证没有通过，通讯将断开；如果合法性验证通过，将继续进行第四步。
4. 用户端随机产生一个用于通讯的“对称密码”，然后用服务器的公钥（服务器的公钥从步骤2中的服务器的证书中获得）对其加密，然后将加密后的“预主密码”传给服务器。
5. 如果服务器要求客户的身份认证（在握手过程中为可选），用户可以建立一个随机数然后对其进行数据签名，将这个含有签名的随机数和客户自己的证书以及加密过的“预主密码”一起传给服务器。
6. 如果服务器要求客户的身份认证，服务器必须检验客户客户证书和签名随机数的合法性，具体的合法验证过程包括：客户的证书使用日期是否有效，为客户提供证书的CA是否可靠，发行CA的公钥是否正确解开客户证书的发行CA的数字签名，检查客户的证书是否在证书废止列表（CRL）中。检验如果没有通过，通讯立刻终端；如果验证通过，服务器将用自己的私钥解开加密的“预主密码”，然后执行一系列步骤来产生主通讯密码（客户端也将通过同样的方法产生相同的主通讯密码）。
7. 服务器和客户端用相同的主密码即“通话密码”，一个对称秘钥用于SSL协议的安全数据通讯的加解密通讯。同时在SSL通讯过程中还要完成数据通讯的完整性，防止数据通讯中的任何变化。
8. 客户端想服务器端发出信息，指明后面的数据通讯将使用的步骤7中的主密码为对称秘钥，同时通知服务器客户端的握手过程结束。
9. 服务器向客户端发出信息，指明后面的数据通讯将使用的步骤7中的主密码为对称秘钥，同时通知客户端服务器端的握手过程结束。
10. SSL的握手部分结束，SSL安全通道的数据通讯开始，客户和服务器开始使用相同的对称秘钥进行数据通讯，同时进行通讯完整性的检验。

# JVM

JVM是java虚拟机的缩写，JVM是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

引入java虚拟机后，JAVA语言在不同平台上运行时不需要重新编译。Java语言使用java虚拟机屏蔽了与具体平台相关的信息，使得java语言编译程序只需生成在Java虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。

Java虚拟机本质是一个程序，当它在命令行上启动的时候，就开始执行保存再某字节码文件中的指令，java语音的可移植性正是建立在java虚拟机的基础上。任何平台只要装有针对于该平台的Java虚拟机，字节码文件（.class）就可以在该平台上运行。这就是“一次编译，多次运行”

Java虚拟机不仅是一种跨平台的语言，而且是一种新的网络计算平台。该平台包括许多相关的技术，如符合开放接口标准的各种API、优化技术等。Java技术使同一种应用可以运行在不同的平台上。Java平台可分为两部分，即Java虚拟机和Java API类库

**Jvm类加载机制**

**概念：**虚拟机把描述类的数据从class文件加载到内存，并对数据进行校验、转换解析和初始化。最终形成可以被虚拟机最直接使用的Java类型的过程就是虚拟机的类加载机制。

# JDK、JRE、JVM的区别联系

JDK是Java开发工具包，是Sun Microsystems针对Java开发员的产品。

　　JDK中包含JRE，在JDK的安装目录下有一个名为jre的目录，里面有两个文件夹bin和lib，在这里可以认为bin里的就是jvm，lib中则是jvm工作所需要的类库，而jvm和 lib和起来就称为jre。

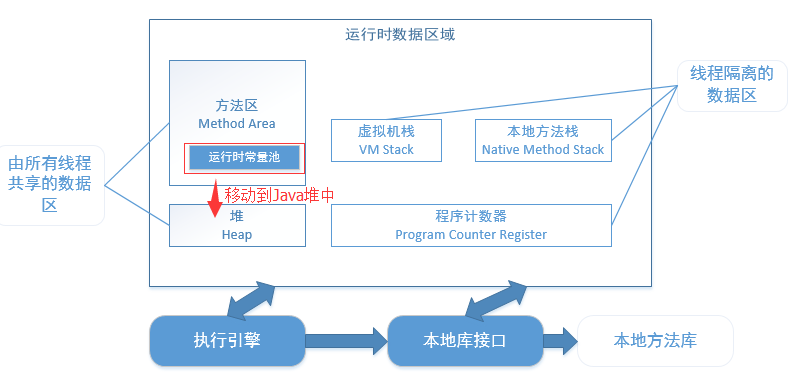
JDK是整个JAVA的核心，包括了Java运行环境JRE（Java Runtime Envirnment）、一堆Java工具（javac/java/jdb等）和Java基础的类库（即Java API 包括rt.jar）。

# JVM内存模型，内存结构

JVM内存结构：

对于大多数应用来说，Java堆（Java heap）是java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。





由上图可以清楚的看到JVM的内存空间分为3大部分：

1. 堆内存。
2. 方法区
3. 栈内存
4. 常量池

其中栈内存可以再细分为Java虚拟机栈和本地方法栈，堆内存可以划分为新生代和老年代，新生代中还可以再次再次划分为Eden区、From Surivor区和To Servivor区。

其中一部分是线程共享的，包括Java堆和方法区；另一部分是线程私有的，包括虚拟机栈和本地方法栈，以及程序计数器这一小部分内存。

**堆内存（Heap）**

对于大多数应用来说，Java堆（Java Heap）是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。

此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

堆内存是所有线程共有的。

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称为“GC”堆。从内存回收角度看，由于现在收集器基本都采用分代收集算法，所以Java堆还可以细分为：新生代和老年代；再细致一点的有Eden空间，From Survivor空间，To Survivor空间等。从内存分配的角度看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区。不过无论如何划分，都与存放内容无关，无论哪个区域，存储的都仍然是对象实例，进一步划分是为了更好的回收内存，或者更快的分配内存（如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常）

新生代（Young）与老年代（Old）的比例的值为1：2（该值可以通过参数-XX：NewRatio来指定），

默认的，Eden：From：to = 8 ：1：1（可以通过参数-XX：SurvivorRatio来设定），即：Eden = 8/10 的新生代空间大小，From = to = 1/10的新生代空间大小。

**方法区（Method Area）**

方法区也称“永久代”，它用于存储虚拟机加载的类信息、常量、静态变量、是各个线程共享的内存区域。

在JDK8之前的HotSpot JVM,存放这些“永久的”的区域叫做“永久代（permanent generation）”。永久代是一片连续的堆空间，在JVM启动之前通过在命令行设置参数-XX:MaxPermSize来设定永久代最大可分配的内存空间，默认大小是64M（64位JVM默认是85M）。

随着JDK8的到来，JVM不再有永久代（PermGen）。但类的元数据信息（metadata）还在，只不过不再是存储在连续的堆空间上，而是移动到叫做“Metaspace”的本地内存（Native memory）

方法区或永久代相关设置

-XX:PermSize=64MB 最小尺寸，初始分配

-XX:MaxPermSize=256MB 最大允许分配尺寸，按需分配

XX:+CMSClassUnloadingEnabled -XX:+CMSPermGenSweepingEnabled 设置垃圾不回收

默认大小

-server选项下默认MaxPermSize为64M

-client选项下默认MaxPermSize为32M

**虚拟机栈（JVM Stack）**

线程私有，生命周期和线程相同，描述的是Java方法执行的内存模型：每个方法被执行的时候都会创建一个“栈帧”，用于存储局部变量表（包括参数）、操作栈、方法出口等信息。每个方法被调用到执行完的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

在Java虚拟机规范中，对此区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将会抛出StackOverFlowError异常；如果虚拟机栈可以动态扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

**本地方法栈（Native Stack）**

本地方法栈（Native Method Stacks）与虚拟机所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。

在虚拟机规范中对本地方法栈中方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机直接把本地方法栈和虚拟机栈合二为一，与虚拟机栈一样也会抛出StackOverflowError和OutOfMemoryError异常。

**程序计数器（PC Register）**

占据一块较小的内存空间，可以看做当前线程所执行的字节码的行号指示器。在虚拟机概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支，循环，跳转，异常处理，线程恢复等基础功能都需要依赖这个计数器来完成。

由于JVM的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器都只会执行一条线程中的指令。因此未来线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

多线程情况下，每个线程都具有各自独立的程序计数器，所以该区域是非线程共享的内存区域。

当执行Java方法时候，计数器记录的则是正在执行的虚拟机字节码指令的地址，当执行Native方法时，计数器的值为空。

此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

**常量池**

运行时常量池是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

运行时常量池有个重要特性是动态性，Java语言不要求常量一定只在编译期才能产生，也就是并非预置入class文件中常量池的内容才能进入方法区的运行时常量池，运行期间也有可能将新的常量放入池中，这种特性使用最多的是String类的intern()方法。

既然运行时常量池是方法区的一部分，自然受到方法区内存的限制。当常量池无法再申请到内存时会抛出OutOfMemeryError异常。

**JVM内存设置**



**-Xms设置堆的最小空间大小。**

**-Xmx 设置堆的最大空间大小**

**-Xmn 设置年轻代大小**

**-XX:NewSize 设置新生代最小空间大小。**

**-XX:MaxNewSize 设置新生代最大空间大小。**

**-XX:PermSize 设置永久代最小空间大小。**

**-XX:MaxPermSize 设永久代最大空间大小。**

**-Xss 设置每个线程的堆栈大小**

**-XX:+UseParallelGC 选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代扔旧使用串行收集。**

**-XX:ParallelGCThreads=20 配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。**

配置参考：

java-Xmx3550m-Xms3550m-Xmn2g-Xss128k

-XX:ParallelGCThreads=20

-XX:+UseConcMarkSweepGC-XX:+UseParNewGC

-Xmx3550m:设置JVM最大可用内存为3550M。

-Xms3550m:设置JVM促使内存为3550m。此值可以设置与-Xmx相同,以避免每次垃圾回收完成后JVM重新分配内存。

-Xmn2g:设置年轻代大小为2G。整个堆大小=年轻代大小+年老代大小+持久代大小。持久代一般固定大小为64m,所以增大年轻代后,将会减小年老代大小。此值对系统性能影响较大,官方推荐配置为整个堆的3/8。

-Xss128k:设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M,以前每个线程堆栈大小为256K。更具应用的线程所需内存大 小进行调整。在相同物理内存下,减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的,不能无限生成,经验值在3000~5000 左右。

# 对象的创建过程

1. 当虚拟机遇到一条New指令时：会进行如下步骤

（1）、检查指令的参数（即工作中我们New的对象），能否在常量池中找到它的符号引用。

（2）、如果存在、检查符号引用的类是否被加载、解析、初始化过。如果没有则执行类的加载。

（3）、加载通过后，虚拟机将为新生对象分配内存。（所需内存大小在类加载完成后便可确定）

2、两种内存分配的方式：

**指针碰撞：**假设Java堆中的内存是绝对规整的，所有用过的内存都放在一边，空间的内存放在另一边。中间放着一个指针作为分界点的指示器，分配内存就仅仅是把指针往空闲空间那边挪动一段与对象大小相等的近距离。这种方式属于指针碰撞。

**空闲列表：**如果堆中的内存并不是规整的，已使用的内存和空闲内存相互交错，显然无法使用指针碰撞。虚拟机就必须维护一个列表，记录那些内存是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新记录表上的数据。这种方式属于空闲列表。

具体选择哪种分配方式由Java堆决定，而Java堆是否规整，则有GC收集器决定。因此使用Serial、ParNew等带Compact过程的收集器时，系统采用的分配算法是指针碰撞。而使用CMS这种基于Mark-Sweep算法收集器时，通常采用的空闲列表。

3、如何保证分配内存时线程的安全性

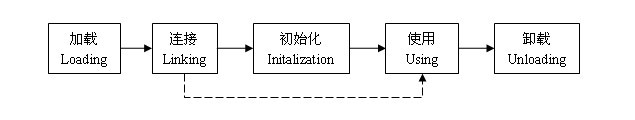
（1）、对分配内存的动作进行同步处理（实际上虚拟机采用CAS配上失败重试的机制保证了更新操作的原子性）

（2）、把分配内存的动作按照线程划分在不同的空间之中进行（即每个线程在Java堆中预先分配一小块内存（称为本地线程分配缓冲））。

# 类的生命周期

当我们编写一个java源文件后，经过编译会生成一个后缀名为class的文件，这种文件叫做字节码文件，只有这种字节码文件才能够在java虚拟机中运行，java类的生命周期就是指一个class文件从加载到卸载的全过程。

一个java类的完整的生命周期会经历**加载、连接、初始化、使用和卸载**五个阶段，当然也有在加载或者连接之后没有被初始化就直接被使用的情况，如图所示：



加载：在java中，我们经常会接触到一个词---类加载，它和这里的加载并不是一回事，通常我们说类加载指的是类的生命周期中加载、连接、初始化三个阶段。在加载阶段，java虚拟机会找到需要加载的类的信息加载到JVM的方法区中，然后在堆区中实例化一个java.lang.class对象，作为方法区中这个类的信息入口。

类的加载方式比较灵活，我们最常用的加载方式有两种，一种是根据类的全路径名找到相应的class文件，然后从class文件中读取文件内容；另一种是从jar文件中读取。另外，还有下面几种方式比较常用：

* 从网络中获取：比如10年前十分流行的Applet。
* 根据一定的规则实时生产，比如设计模式中的动态代理模式，就是根据相应的类自动生成它的代理类。
* 从非class文件中获取，其实这与直接从class文件中获取的方式本质上是一样的，这些非class文件在jvm中运行之前会被转换为可被jvm所识别的字节码文件。

对于加载的时机，各个虚拟机的做法并不一样，但是有一个原则，就是当jvm“预期”一个类将要被使用时，就会在使用它之前对这个类进行加载。比如说，在一段代码中出现了一个类的名字，jvm在执行这段代码之前并不能确定这个类是否会被使用到，于是，有些jvm会在执行前就加载这个类，而有些则在真正需要用的时候才会去加载它，这取决于具体的jvm实现。我们常用的hotspot虚拟机是采用的后者，就是说当真正用到一个类的时候才对它进行加载。

加载阶段是类的生命周期中的第一个阶段，加载阶段之后，是连接阶段。有一点需要注意，就是有时连接阶段并不会等加载阶段完全完成之后才开始，而是交叉进行，可能一个类只加载了一部分之后，连接阶段就已经开始了。但是这两个阶段总的开始时间和完成时间总是固定的，加载阶段总是在连接阶段之前开始，连接阶段总是在加载阶段完成之后完成。

连接：连接阶段比较复杂，一般会跟加载阶段和初始化阶段交叉进行，这个阶段的主要任务就是做一些加载后的验证工作以及一些初始化前的准备工作，可以细分为三个步骤：验证、准备和解析。

1. **验证：**当一个类被加载之后，必须要验证一下这个类是否合法，比如这个类是不是符合字节码的格式、变量与方法是不是有重复、数据类型是不是有效、继承与实现是否合乎标准等等。总之，这个阶段的目的就是保证加载的类是能够被jvm所运行。
2. **准备：**准备阶段的工作就是为类的静态变量分配内存并设为jvm默认的初值，对于非静态的变量，则不会为它们分配内存。有一点需要注意，这时候，静态变量的初值为jvm默认的初值，而不是我们在程序中设定的初值。Jvm默认的初值是这样的。

* 基本类型（int、long、short、char、byte、boolean、float、double）的默认值为0。
* 引用类型的默认值为null。
* 常量的默认值为我们程序中设定的值，比如我们在程序中定义final static int a = 100，则准备阶段中的a的初始值就是100。

1. **解析：**这一阶段的任务就是把常量池中的符号引用转换为直接引用。在解析阶段，jvm会将所有的类或接口名、字段名、方法名转换为具体的内存地址。

（那么什么是符号引用，什么又是直接引用呢？我们来举个例子：我们要找一个人，我们现有的信息是这个人的身份证号是1234567890。只有这个信息我们显然找不到这个人，但是通过公安局的身份系统，我们输入1234567890这个号之后，就会得到它的全部信息：比如安徽省黄山市余暇村18号张三，通过这个信息我们就能找到这个人了。这里，123456790就好比是一个符号引用，而安徽省黄山市余暇村18号张三就是直接引用。在内存中也是一样，比如我们要在内存中找一个类里面的一个叫做show的方法，显然是找不到。但是在解析阶段，jvm就会把show这个名字转换为指向方法区的的一块内存地址，比如c17164，通过c17164就可以找到show这个方法具体分配在内存的哪一个区域了。这里show就是符号引用，而c17164就是直接引用。）

**连接阶段完成之后会根据使用的情况（直接引用还是被动引用）来选择是否对类进行初始化。**

初始化：如果一个类被直接引用，就会触发类的初始化。在java中，直接引用的情况如下

* 通过new关键字实例化对象、读取或设置类的静态变量、调用类的静态方法。
* 通过反射方式执行以上三种行为。
* 初始化子类的时候，会触发父类的初始化。
* 作为程序入口直接运行时（也就是直接调用main方法）

除了以上四种情况，其他使用类的方式叫做被动引用，而被动引用不会触发类的初始化。请看主动引用的实例代码：

**class** InitClass{  
 **static** {  
 System.***out***.println(**"初始化InitClass"**);  
 }  
 **public static** String *a* = **null**;  
 **public static void** method(){}  
}  
  
**class** SubInitClass **extends** InitClass{}  
  
**public class** TestD {  
  
 */\*\*  
 \* 主动引用引起类的初始化的第四种情况就是运行Test1的main方法时  
 \* 导致Test1初始化，这一点很好理解，就不特别演示了。  
 \* 本代码演示了前三种情况，以下代码都会引起InitClass的初始化，  
 \* 但由于初始化只会进行一次，运行时请将注解去掉，依次运行查看结果。  
 \** ***@param args*** *\** ***@throws*** *Exception  
 \*/* **public static void** main(String[] args) **throws** Exception{  
 *// 主动引用引起类的初始化一: new对象、读取或设置类的静态变量、调用类的静态方法。  
 // new InitClass();  
 // InitClass.a = "";  
 // String a = InitClass.a;  
 // InitClass.method();  
  
 // 主动引用引起类的初始化二：通过反射实例化对象、读取或设置类的静态变量、调用类的静态方法。  
 // Class cls = InitClass.class;  
 // cls.newInstance();  
  
 // Field f = cls.getDeclaredField("a");  
 // f.get(null);  
 // f.set(null, "s");  
  
 // Method md = cls.getDeclaredMethod("method");  
 // md.invoke(null, null);  
  
 // 主动引用引起类的初始化三：实例化子类，引起父类初始化。  
 // new SubInitClass();* }  
}

在类的初始化阶段，只会初始化与类相关的静态赋值语句和静态语句，也就是有static关键字修饰的信息，而没有static修饰的赋值语句和执行语句在实例化对象的时候才会运行。

使用：类的使用包括主动引用和被动引用，主动引用在初始化时已经说过了，下面说一下被动引用：

* 引用父类的静态字段，只会引起父类的初始化，而不会引起子类的初始化。
* 定义类数组，不会引起类的初始化。
* 引用类的常量，不会引起类的初始化。

被动引用示例代码：

**class** InitClass1{  
 **static** {  
 System.***out***.println(**"初始化InitClass"**);  
 }  
 **public static** String *a* = **null**;  
 **public final static** String ***b*** = **"b"**;  
 **public static void** method(){}  
}  
  
**class** SubInitClass2 **extends** InitClass1{  
 **static** {  
 System.***out***.println(**"初始化SubInitClass"**);  
 }  
}  
  
**public class** TestE {  
  
 **public static void** main(String[] args) **throws** Exception{  
 *// String a = SubInitClass.a;// 引用父类的静态字段，只会引起父类初始化，而不会引起子类的初始化  
 // String b = InitClass.b;// 使用类的常量不会引起类的初始化* SubInitClass[] sc = **new** SubInitClass[10];*// 定义类数组不会引起类的初始化* }  
}

最后总结一下使用阶段：**使用阶段包括主动引用和被动引用，主动引用会引起类的初始化，而被动引用不会引起类的初始化**。

当使用阶段完成之后，java类就进入了卸载阶段。

卸载：在类使用完之后，如果满足下面的情况，类就会被卸载：

* 该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例。
* 加载该类的ClassLoader已经被回收。
* 该类对应的java.lang.Class对象没有任何地方被引用，无法在任何地方通过反射访问该类的方法。

如果以上三个条件全部满足，jvm就会在方法区垃圾回收的时候对类进行卸载，类的卸载过程其实就是在方法区中清空类信息，java类的整个生命周期就结束了。

## 总结：

对象基本上都是在jvm堆区中创建，在创建对象之前，会触发类加载（加载、连接、初始化），当类初始化完成后，根据类信息在堆区中实例化类对象，初始化非静态变量、非静态代码以及默认构造方法，当对象使用完之后会在合适的时候被jvm垃圾收集器回收。读完本文后我们知道，对象的声明周期只是类的声明周期中使用阶段的主动引用的一种情况（即实例化类对象）。而类的整个生命周期则要比对象的生命周期长的多。

# JVM垃圾回收器

前面介绍了Java内存运行时区域的各个部分，其中程序计数器、虚拟机栈、本地方法栈，3个区域随着线程的生存而生成的。内存分配和回收都是确定的。随着线程的结束内存自然就被回收了，因此不需要考虑垃圾回收的问题。而Java堆和方法区则不一样，各线程共享，内存的分配和回收都是动态的。因此垃圾收集器所关注的是这部分内存。

接下来我们讨论JVM是怎么回收这部分内存的。在进行回收前垃圾收集器第一件事情就是确定哪些对象还存活，哪些已经死去。下面介绍两种基础的回收算法。

1. 引用计数算法

给对象添加一个引用计数器，每当有一个地方引用它时计数器就+1，当引用失效时计数器就-1。只要计数器等于0的对象就是不可能再被使用的。此算法在大部分情况下都是一个不错的选择，也有一些著名的应用案例。但是JAVA虚拟机没有使用。

优点：实现简单、判断效率高。

缺点：很难解决对象之间循环引用的问题。

1. 可达性分析算法。

通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有使用任何引用链时，则说明该对象是不可用的。

优点：更加精确和严谨，可以分析出循环数据结构相互引用的情况。

缺点：实现比较复杂、需要分析大量数据，消耗大量时间、分析过程需要GC停顿（引用关系不能发生变化），即停顿所有的Java执行线程（称为“Stop the world”，是垃圾回收重点关注的问题）

主流的商用程序语言（Java，C#）在主流的实现中，都是通过可达性分析来判定对象是否存活的。

通过下图来清晰的感受gc root与对象展示的联系。所示灰色区域对象是存活的，Object5/6/7均是可回收的对象



**判断一个对象生存还是死亡**

宣告一个对象死亡，至少要经历两次标记。

1. 第一次标记

如果对象进行可达性分析算法之后没发现与GC Roots相连的引用链，那它将会第一次标记并且进行一次筛选。

筛选条件：判断此对象是否有必要执行finalize()方法。

筛选结果：当对象没有覆盖finalize()方法、或者finalize()方法已经被JVM执行过，则判定为可回收对象。如果对象有必要执行finalize()方法，则被放入F-Queue队列中。稍后在JVM自动建立、低优先级的Finalizer线程（可能多个线程）中触发这个方法。

1. 第二次标记

GC对F-Queue队列中的对象进行二次标记。

如果对象在finalize()方法中重新与引用链上的任何一个对象建立了关联，那么二次标记时则会将它移除出“即将回收”集合。如果此时对象还没成功逃脱，那么只能被回收了。

1. Finalize()方法

Finalize()是Object类的一个方法、一个对象的finalize()方法只会被系统自动调用一次，经过finalize()方法逃脱死亡的对象，第二次不会再调用。

特别说明：并不提倡在程序中调用finalize()来进行自救。建议忘掉Java程序中该方法的存在。因为它执行的时间不确定，甚至是否被执行也不确定（Java程序的不正常退出），而且运行代价高昂，无法保证各个对象的调用顺序（甚至有不同线程中调用）。

**几种常见的垃圾回收算法**

1. **标记-清除算法**

最基础的收集算法，总共分为“标记”和“清楚”两个阶段

1. 标记

标记出所有需要回收的对象。

一次标记：在经过可达性分析算法后，对象没有与GC Root相关的引用链，那么则被第一次标记。并且进行一次筛选：当对象有必要执行finalize()方法时，则把该对象放入F-Queue队列中。

二次标记：对F-Queue队列中的对象进行二次标记。在执行finalize()方法时，如果对象重新与GC Root引用链上的任意对象建立了关联，则把他移除“即将回收”集合，否则将被回收、

对于被第一次标记且被第二次标记的，就可以判定为可回收对象了。

1. 清除

两次标记后，还在“即将回收”集合的对象进行回收。

优点：基础最基础的可达性算法，后续的收集算法都是基于这种思想实现的。

缺点：标记和清楚效率不高，产生大量不连续的内存碎片，导致创建大对象时找不到连续的空间，不得不提前触发另一次的垃圾回收。

执行过程如下



1. **复制算法**

将可用内存按容量分为大小相等的两块，每次只是用其中一块，当这一块的内存用完了，就将还存活的对象复制到另一块内存上，然后再把已使用的内存空间一次清理掉。

**优点：**实现简单，效率高。解决了标记-清除算法导致的内存碎片问题。（可使用“指针碰撞”的方式分配内存）

**缺点：**代价太大，将内存缩小了一半。（可以改良，不按1：1比例划分）

效率随对象的存活率升高而降低。



1. **标记整理算法**

标记-整理算法是根据老年代的特点应运而生。

3.1 标记

标记过程和标记-清除算法一致（也是基于可达性分析算法）。

3.2 整理

和标记-清除不同的是，该算法不是针对可回收对象进行清理，而是根据存活对象进行整理。让存活对象都向一端移动，然后直接清理掉边界以外的内存。

优点：不会像复制算法那样随着存活对象的升高而降低效率，不像标记-清楚算法那样产生不连续的内存碎片。

缺点：效率问题，除了像标记-清除算法的标记过程外，还多了一步整理过程，效率更低。



1. **分代收集算法**

当前商业虚拟机的垃圾收集都是采用“分代收集”算法。

根据对象存活周期的不同将内存分为几块。一般把Java堆分为新生代和老年代。JVM根据各个年代的特点采用不同的收集算法。

新生代中，每次进行垃圾回收都会发现大量对象死去，只有少量存活，因此比较适合复制算法。只需要付出少量存活对象的复制成本就可以完成收集。

老年代中，因为对象存活率较高，没有额外的空间进行分配担保，所以适合标记-清除，标记-整理算法进行回收。

**垃圾回收器**



图中展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，则说明它们可以搭配使用。虚拟机所处的区域则表示它是属于新生代还是老年代收集器。

**新生代收集器：**Serial、ParNew、Parallel Scavenge

**老年代收集器：**CMS、Serial Old、Parallel Old

**整堆收集器：**G1

**相关概念：**

**并行收集：**指多条垃圾收集线程并行工作，但此时用户线程扔处于等待状态。

**并发收集：**指用户线程与垃圾收集线程同时工作（不一定是并行的可能会交替执行）。用户程序在继续运行，而垃圾收集程序运行在另一个CPU上。

**吞吐量：**即CPU用于运行用户代码的时间与CPU总消耗时间的比值。例如：虚拟机共运行100分钟，垃圾收集器花掉1分钟，那么吞吐量就是99％

（吞吐量 = 运行用户代码时间/（运行用户代码时间 + 垃圾收集时间））

1. **Serial收集器**

Serial收集器是最基本的、发展历史最悠久的收集器。

**特点：**单线程、简单高效（与其他收集器的单线程相比），对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。收集器进行垃圾回收时，必须暂停其他所有的工作线程，直到它结束（stop the world）

**应用场景：**适用于Client模式下的虚拟机。

**Serial/Serial Old收集器运行示意图**



1. **ParNew收集器**

ParNew收集器其实就是Serial收集器的多线程版本。

除了使用多线程外其余行为均和Serial收集器一模一样（参数控制、收集算法、Stop The World、对象分配规则、回收策略等）

特点：多线程、ParNew收集器默认开启的收集线程数与CPU的数量相同，在CPU非常多的环境中，可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。和Serial收集器一样存在Stop The World问题。

应用场景：ParNew收集器是许多运行在Server模式下的虚拟机中首选的新生代收集器，因为它是除了Serial收集器外，唯一一个能与CMS收集器配合工作的。

**ParNew/Serial Old组合收集器运行示意图如下：**



1. **Parallel Scavenge 收集器**

与吞吐量关系密切，故也称为吞吐量优先收集器。

**特点：**属于新生代收集器，也是采用复制算法的收集器，又是并行的多线程收集器。

该收集器的目标是达到一个可控制的吞吐量。还有一个值得关注的点是：GC自适应调节策略（与ParNew收集器最重要的一个区别）

**GC自适应调节策略：**Parallel Scavenge收集器可设置-XX:+UseAdptiveSizePolicy参数。当开关打开时不需要手动指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRation）、晋升老年代的对象年龄（-XX:PretenureSizeThreshold）等，虚拟机会根据系统的运行状况收集性能监控信息，动态设置这些参数以提供最优的停顿时间和最高的吞吐量，这种调节方式成为GC的自适应调节策略。

**Parallel Scavenge收集器使用两个参数控制吞吐量。**

**XX:MaxGCPauseMillis 控制最大的垃圾收集停顿时间**

**XX:GCRatio 直接设置吞吐量的大小。**

1. **Serial Old收集器**

Serial Old是Serial收集器的老年代版本。

**特点：**同样是单线程收集器，采用标记-整理算法。

**应用场景：**主要也是使用在Client模式下的虚拟机中。也可在Server模式下使用。

**Serial / Serial Old收集器工作过程图（Serial收集器图示相同）：**



1. **Parallel Old收集器**

是Parallel Scavenge收集器的老年代版本。

**特点：多线程，采用标记-整理算法。**

**应用场景：**注重高吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge+Parallel Old收集器。

**Parallel Scavenge/Parallel Old收集器工作过程图：**



1. **CMS收集器**

一种以最短回收停顿时间为目标的收集器。

**特点：基于标记-清除算法实现。并发收集、低停顿。**

**应用场景**：适用于注重服务的相应速度，希望系统停顿时间最短，给用户带来更好的体验等场景下。如web程序、B/S服务。

**CMS收集器的运行过程分为下列4步：**

**初始标记：**标记GC Roots能直接到的对象，速度很快但是仍存在Stop the world问题。

**并发标记：**进行GC Roots Tracing的过程，找出存活对象且用户线程可并发执行。

**重新标记：**为了修正并发标记期间因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录。仍然存在Stop the world问题。

**并发清除：**对标记的对象进行清除回收。

**CMS收集器的内存回收过程是与用户线程一起并发执行的。**

**CMS收集器的工作过程图：**



CMS收集器的缺点：

对CPU资源非常敏感。

无法处理浮动垃圾，可能出现Concurrent Model Failure失败而导致另一次Full GC的产生

因为采用标记-清除算法所以会存在空间碎片的问题，导致大对象无法分配空间，不得不提前出发一次Full GC。

1. **G1收集器**

一款面向服务端应用的垃圾收集器

**特点如下：**

**并行与并发**：G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop-The-World停顿时间。部分收集器原本需要停顿Java线程来执行GC动作，G1收集器仍然可以通过并发的方式让Java程序继续运行。

**分代收集：**G1能够独自管理整个Java堆，并且采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次GC的旧对象以获取更好的收集效果。

**空间整合：**G1运作期间不会产生空间碎片，收集后能提供规整的可用内存。

**可预测的停顿：**G1除了追求低停顿外，还能建立可预测的停顿时间模型。能让使用者明确指定在一个长度为M毫秒的时间段内，消耗在垃圾收集上的时间不得超过N毫秒。

如果不计算维护Remembered Set的操作，G1收集器大致可分为如下步骤：

1. **初始标记：**仅标记GC Roots能直接到的对象，并且修改TAMS（Next Top at Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可用的Region中创建新对象。（需要线程停顿，但耗时很短）。
2. **并发标记：**从GC Roots开始对堆中对象进行可达性分析，找出存活对象。（耗时较长，但可与用户程序并发执行）
3. **最终标记：**为了修正在并发标记期间因用户程序执行而导致标记产生变化的那一部分标记记录。且对象的变化记录在线程Remembered Set Logs里面，把Remembered Set Logs里面的数据合并到Remembered Set中。（需要线程停顿，但可并行执行）
4. **筛选回收：**对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划。（可并发执行）

**G1收集器运行示意图**



（链接：<https://www.cnblogs.com/chenpt/p/9803298.html>）

# 线程

**概念：**

线程被称为轻量级进程，是程序执行的最小单位，它是指在程序执行过程中，能够执行代码的一个执行单位。每个程序都至少有一个线程，也即是程序本身。

**线程的状态：**

Java语言定义了5种线程状态，在任意一个时间点，一个线程只能有且只有其中一个状态。这5种状态如下：

1. **新建（New）：**创建后尚未启动的线程处于这种状态。
2. **运行（Runable）：**Runable包括了操作系统线程状态的Running和Ready，也就是处于此状态的线程有可能正在执行，也有可能正在等待着CPU为它分配执行时间。
3. **等待（Wating）：**处于这种状态的线程不会被分配CPU执行时间。等待状态又分为无限期等待和有限期等待，处于无限期等待的线程需要被其他线程显示地唤醒，没有设置Timeout参数的Object.wait()、没有设置Timeout参数的Thread.join()方法都会使线程进入无限期等待状态；有限期等待状态无须等待被其他线程显示地唤醒，在一定时间之后它们会由系统自动唤醒，Thread.sleep()、设置了Timeout参数的Object.wait()、设置了Timeout参数的Thread.join()方法都会使线程进入有限期等待状态。
4. **阻塞（Blocked）：**线程被阻塞了，“阻塞状态”与“等待状态”的区别是，“阻塞状态”在等待着获取到一个排他锁，这个时间将在另一个线程放弃这个锁的时候发生；而“等待状态”则是在等待了一段时间或者唤醒动作的发生。在程序等待进入同步区域的时候，线程将进入这种状态。
5. **结束（Terminated）：**已终止线程的线程状态，线程已经结束执行。

**5种状态的转换图：**



线程同步的方法：

线程中有4种同步方法，分别为wait()、sleep()、notify()和notifyAll()。

1. wait()：使线程处于一种等待状态，释放所持有的对象锁。
2. sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用它时要捕获InterruptedException异常，不释放对象锁。
3. notify()：唤醒一个正在等待状态的线程。注意调用此方法时，并不能确定唤醒的是哪一个等待状态的线程，是由JVM来决定唤醒哪个线程，不是由线程优先级决定的。
4. notifyAll()：唤醒所有等待状态的线程，注意并不是给所有唤醒线程一个对象锁，而是让它们竞争。

**创建线程的四中方式**

1. **继承Thread类创建线程。**
2. 定义Thread类的子类MyThread，并重写run()方法，run()方法的方法体（线程执行体）就是线程要执行的任务。
3. 创建MyThread类的实例
4. 调用子类实例的start()方法来启动线程。

public class Mythread extends Thread {

private int i;

public void run(){//run()是线程类的核心方法

for(int i=0;i<10;i++){

System.out.println(this.getName()+":"+i);

}

}

public static void main(String[] args) {

Mythread t1=new Mythread();

Mythread t2=new Mythread();

Mythread t3=new Mythread();

t1.start();

t2.start();

t3.start();

}

1. **实现Runnable接口创建线程。**
2. 定义Runable接口的实现类，必须重写run()方法，这个run()方法和Thread中的run()方法一样，是线程的执行体。
3. 创建Runnable实现类的实例，并用这个实例作为Thread的target来创建Thread对象，这个Thread对象才是真正的线程对象。
4. 调用start()方法。

public class MyThread implements Runnable{

@Override

public void run() {

for(int i=0;i<10;i++){

System.out.println(Thread.currentThread().getName()+" : "+i);

}

}

public static void main(String[] args) {

MyThread myThread=new MyThread();

Thread thread1=new Thread(myThread,"线程1");

Thread thread2=new Thread(myThread,"线程2");

Thread thread3=new Thread(myThread,"线程3");

thread1.start();

thread2.start();

thread3.start();

}

}

1. **使用Callable和Future创建线程。**

和Runnable接口不一样，Callable接口提供了一个call()方法作为线程执行体，call()方法功能要强大。

Call()方法可以有返回值

Call()方法可以声明抛出异常

1. 创建Callable接口的实现类，并实现call()方法，然后创建该实现类的实例。
2. 使用FutureTask类来包装Callable对象，该FutureTask对象封装了Callable对象的call()方法的返回值
3. 使用FutureTask对象作为Thread对象的target创建并启动线程（因为FutureTask实现了Runnable接口）
4. 调用FutureTask对象的get()方法来获得子线程执行结束后的返回值。

public class MyThread implements Callable<String>{//Callable是一个泛型接口

@Override

public String call() throws Exception {//返回的类型就是传递过来的V类型

for(int i=0;i<10;i++){

System.out.println(Thread.currentThread().getName()+" : "+i);

}

return "Hello Tom";

}

public static void main(String[] args) throws Exception {

MyThread myThread=new MyThread();

FutureTask<String> futureTask=new FutureTask<>(myThread);

Thread t1=new Thread(futureTask,"线程1");

Thread t2=new Thread(futureTask,"线程2");

Thread t3=new Thread(futureTask,"线程3");

t1.start();

t2.start();

t3.start();

System.out.println(futureTask.get());

}

}

1. **使用线程池创建（使用java.util.concurrent.Executor接口）**

线程池提供了一个线程队列，队列中保存着所有等待状态的线程。避免了创建与销毁额外开销，提交了响应速度。

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

//线程池实现

public class ThreadPoolExecutorTest {

public static void main(String[] args) {

//创建线程池

ExecutorService executorService = Executors.newFixedThreadPool(10);

ThreadPool threadPool = new ThreadPool();

for(int i =0;i<5;i++){

//为线程池分配任务

executorService.submit(threadPool);

}

//关闭线程池

executorService.shutdown();

}

}

class ThreadPool implements Runnable {

@Override

public void run() {

for(int i = 0 ;i<10;i++){

System.out.println(Thread.currentThread().getName() + ":" + i);

}

}

}

**Start和run的区别**

Start会新开一个线程来执行，而run只是一个普通方法，相当于当前线程来调用，不会启动新线程。

**Sleep和wait的区别**

（使用setPriority设置线程优先级，使用getPriority获得线程优先级）

**Sleep：**

Sleep()方法是线程类（Thread）的静态方法，让调用线程进入睡眠状态，让出执行机会给其他线程，等到休眠时间结束后，线程进入就绪状态和其他线程一起竞争cpu的执行时间。

因为sleep()是static静态方法，他不能改变对象的机锁，当一个synchronized块中调用了sleep()方法，线程虽然进入休眠，但是对象的机锁没有被释放，其他线程依然无法访问这个对象。

**Wait：**

Wait()是Object类的方法，当一个线程执行到wait方法时，它就进入到一个和该对象相关的等待池，同时释放对象的机锁，使得其他线程能够访问，可以通过notify，notifyAll方法来唤醒等待的线程。

**样例代码：**

Service类：

**public** **class** Service {

**public** **void** mSleep(){

**synchronized**(**this**){

**try**{

System.***out***.println(" 开始睡眠 。 当前时间："+System.*currentTimeMillis*());

Thread.*sleep*(5\*1000);

**this**.notifyAll();

System.***out***.println(" 唤醒等待 。 结束时间："+System.*currentTimeMillis*());

} **catch** (Exception e){

System.***out***.println(e);

}

}

}

**public** **void** mWait(){

**synchronized**(**this**){

**try**{

System.***out***.println(" 等待开始 。 当前时间："+System.*currentTimeMillis*());

**this**.wait();

System.***out***.println(" 等待结束 。 结束时间："+System.*currentTimeMillis*());

} **catch** (Exception e){

e.printStackTrace();

}

}

}

}

SleepThread类：

**public** **class** SleepThread **implements** Runnable {

**private** Service service;

**public** SleepThread(Service service){

**this**.service = service;

}

**public** **void** run(){

service.mSleep();

}

}

WaitThread类：

**public** **class** WaitThread **implements** Runnable {

**private** Service service;

**public** WaitThread(Service service){

**this**.service = service;

}

**public** **void** run(){

service.mWait();

}

}

测试类：

**public** **class** Test {

**public** **static** **void** main(String[] args){

Service mService = **new** Service();

Thread sleepThread = **new** Thread(**new** SleepThread(mService));

Thread waitThread = **new** Thread(**new** WaitThread(mService));

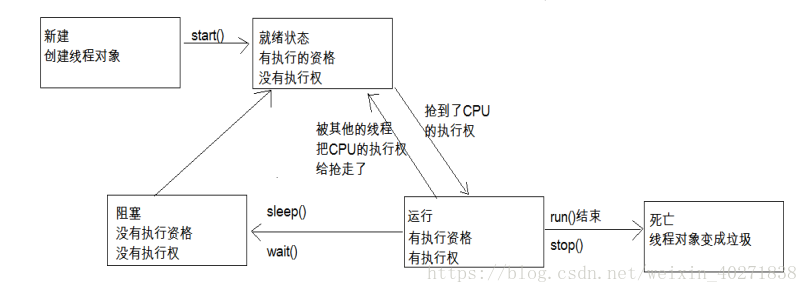
waitThread.start();

sleepThread.start();

}

}

**线程的生命周期**



**并行和并发：**

**并发：**从宏观方面来说，并发就是同时进行多种事件，实际上，这几种事件，并不是同时进行的，而是交替进行的，而由于cpu的运算速度非常的快，会造成我们一种错觉，就是同一时间内进行了多种事情。

**并发：**则是真正意义上的同时进行多种事情。这种只可以在多核CPU的基础下完成。

**线程池：**

在一个应用程序中，我们需要多次使用线程，也就意味着，我们需要多次创建并销毁线程。而创建并销毁线程的过程势必会消耗内存。而在Java中，内存资源是极其宝贵的，所以我们就提出了线程池的概念。

**线程池：**Java中开辟出了一种管理线程的概念，这个概念叫做线程池，从概念以及应用场景中，我们可以看出，线程池的好处，就是可以方便管理线程，也可以减少内存的消耗。

**如何创建一个线程池**

Java中已经提供了创建线程池的一个类：Executor。而我们创建时，一般使用它的子类：ThreadPoolExector。

public ThreadPoolExecutor(int corePoolSize,

                              int maximumPoolSize,

                              long keepAliveTime,

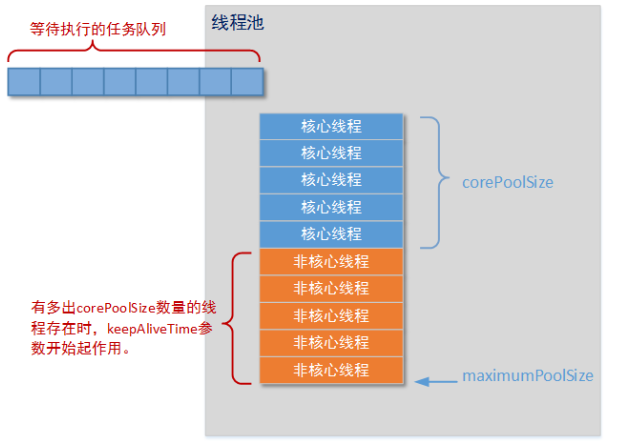
                              TimeUnit unit,

                              BlockingQueue<Runnable> workQueue,

                              ThreadFactory threadFactory,

                              RejectedExecutionHandler handler)

这是其中最重要的一个构造方法，这个方法决定了创建出来的线程池的各种属性，下面依靠一张图来更好的理解线程池和这几个参数：



**corePoolSize：**线程池中的corePoolSize就是线程池中的核心线程数量，这几个核心线程，就算是在没有用的时候，也不会被回收。

**maximumPoolSize：**就是线程池中可以容纳的最大线程的数量，

**keepAliveTime：**就是线程池中除了核心线程之外的其他的最长可以保留的时间，因为在线程池中，除了核心线程即使在无任务的情况下也不能被清楚，其余的都是有存活时间的，意思就是非核心线程可以保留的空闲时间

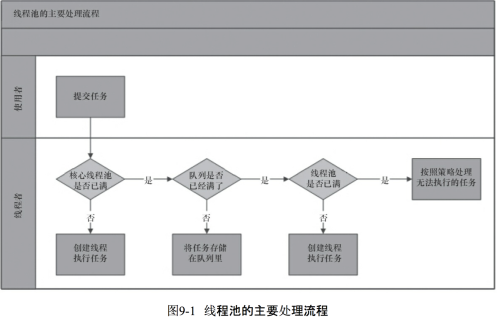
**util：**就是计算这个时间的一个单位，

**workQueue：**等待队列，任务可以存储在任务队列中等待被执行，执行的是FIFIO原则（先进先出）。

**ThreadFactory：**就是创建线程的线程工厂。

**Handler：**是一种拒绝策略，我们可以在任务满了之后，拒绝执行某些任务。

**线程池的执行流程：**



由图我们可以看出，任务进来时，首先执行判断，判断核心线程是否处于空闲状态，如果不是，核心线程就先执行任务，如果核心线程已满，则判断任务队列是否有地方存放该任务，如果有，就将任务保存再任务队列中，等待执行，如果满了，在判断最大可容纳的线程数，如果没有超出这个数量，就开创非核心线程执行任务，如果超出了，就调用Handler实现拒绝策略。

**Handler的拒绝策略：**

有四种：

第一种AbortPolicy：不执行新任务，直接抛出异常，提示线程池已满

第二种DisCardPolicy：不执行新任务，也不抛出异常。

第三种DisCardOldSetPolicy：将消息队列中的第一个任务替换为当前新进来的任务执行。

第四种CallerRunsPolicy：直接调用execute来执行当前任务

**四种常见的线程池：**

**CachedThreadPool：**可缓存的线程池，该线程池中没有核心线程，非核心线程的数量为Integer.max\_value，就是无限大，当有需要时创建线程来执行任务，没有需要时回收线程，适用于耗时少，任务量大的情况。

**SecudleThreadPool：**周期性执行任务的线程池，按照某种特定的计划执行线程中的任务，有核心线程，但也有非核心线程，非核心线程的大小也为无限大。适用于执行周期性的任务。

**SingleThreadPool：**只有一条线程来执行任务，适用于有顺序的任务的应用场景。

**FixedThreadPool：**定长的线程池，有核心线程，核心线程的即为最大的线程数量，没有非核心线程。

# 消息队列

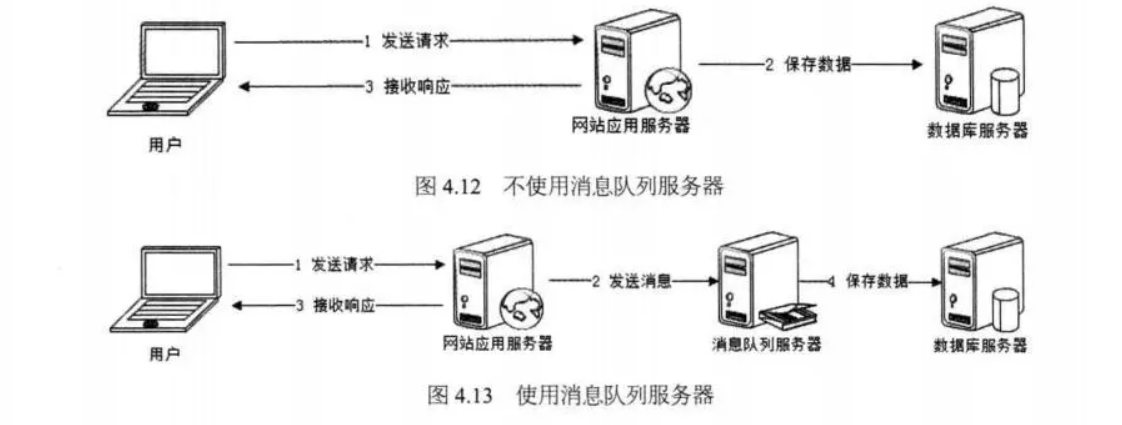
**一、什么是消息队列：**

我们可以把消息队列比作是一个存放消息的容器，当我们需要使用消息的时候可以取出消息供自己使用。消息队列是分布式系统中重要的组件，使用消息队列主要是为了通过异步处理提供系统性能和削峰、降低系统耦合性。目前使用较多的消息队列有ActiveMQ、RabbitMQ、Kafka、RocketMQ。

**二、为什么要用消息队列**

使用消息队列主要有两点好处：

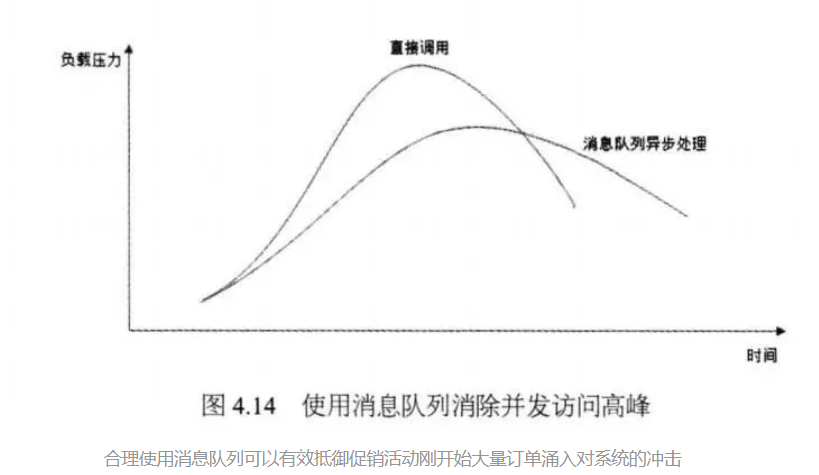
1. 通过异步处理提高系统性能（削峰、减少响应所需时间）



如图，在不使用消息队列服务器的时候，用户的请求数据直接写入数据库，在高并发的情况下数据库压力剧增，使得响应速度变慢。但是在使用消息队列之后，用户的请求数据发送给消息队列之后立即返回，再由消息队列的消费者进程从消息队列中获取数据，异步写入数据库。由于消息队列服务器处理速度快于数据库（消息队列也比数据库有更好的伸缩性），因此响应速度得到大幅改善。

通过以上分析我们可以得出消息队列具有很好的削峰作用的功能——即通过异步处理，将短时间高并发产生的事务消息存储在消息队列中，从而削平高峰期的并发事务。

例：在电子商务一些秒杀、促销活动中，合理使用消息队列可以有效抵御促销活动刚开始大量订单涌入对系统的冲击。如下图。

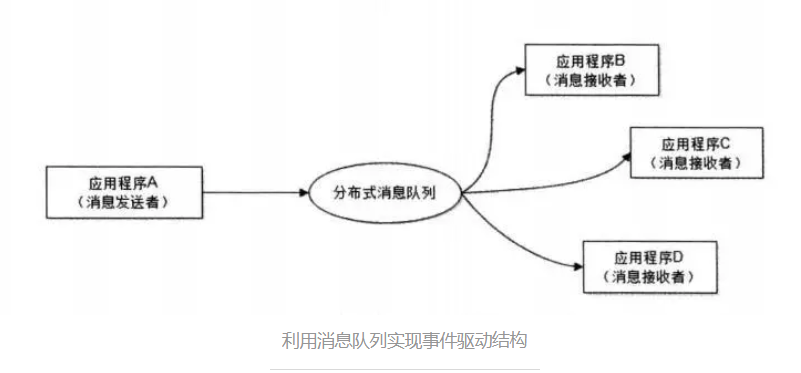


因为用户请求数据写入消息队列之后就立即返回给用户了，但是请求数据在后续的业务校验、写数据库等操作中可能失败。因此使用消息队列进行异步处理之后，需要适当修改业务流程进行配合，比如用户在提交订单之后，订单数据写入消息队列，不能立即返回用户订单提交成功，需要在消息队列的订单消费者进程真正处理完该订单之后，甚至出库后，再通过电子邮件或短信通知用户订单成功，以免交易纠纷。这就类似我们平时手机订火车票和电影票。

1. 降低系统耦合性

我们知道如果模块之间不存在直接调用，那么新增模块或者修改模块就是对其他模块影响较小，这样系统的可扩展性无疑更好一些。

我们最常见的事件驱动架构类似生产者消费者模式，在大型网站中通常利用消息队列实现事件驱动结构。如下图所示。



消息队列使用发布-订阅模式工作，消息发送者（生产者）发布消息，一个或多个消息接受者（消费者）订阅消息。从上图可以看到消息发送者（生产者）和消息接受者（消费者）之间没有直接耦合，消息发送者将消息发送至分布式消息队列即结束对消息的处理，消息接受者从分布式消息队列获取该消息后进行后续处理，并不需要知道该消息从何而来。对新增业务，只有对该类消息感兴趣，即可订阅该消息，对原有系统和业务没有任何影响，从而实现网站业务的可扩展性设计。

消息接受者对消息进行过滤、处理、包装后，构造成一个新的消息类型，将消息继续发送出去，等待其他消息接受者订阅该消息。因此基于事件（消息对象）驱动的业务架构可以是一系列流程。

另外为了避免消息队列服务器宕机造成消息丢失，会将成功发送到消息队列的消息存储在消息生产者服务器上，等消息真正被消费者服务器处理后才删除消息。在消息队列服务器宕机后，生产者服务器会选择分布式消息队列服务器集群中的其他服务器发布消息。

备注：不要认为消息队列只能利用发布-订阅模式工作，只不过在解耦这个特定业务环境下使用发布-订阅模式的。除了发布-订阅模式，还有点对点订阅模式（一个消息只有一个消费者），我们比较常用的是发布-订阅模式。另外，这两种消息模型是JMS提供的，AMQP协议还提供了5中消息模型。

**三、使用消息队列带来的一些问题**

系统可用性降低：系统可用性在某种程序上降低，为什么这样说？在加入MQ之前，你不用考虑消息丢失或者说MQ挂掉等等的情况，但是，引入MQ之后你就需要去考虑了。

系统复杂性提高：加入MQ之后，你需要保证消息没有被重复消费、处理消息丢失的情况、保证消息传递的顺序性等问题。

一致性问题：消息队列带来的异步确实可以提高系统的相应速度。但是，万一消息的真正消费者并没有正确消费消息，就会导致数据不一致的情况。

# 锁

**锁的类型：**

**可重入锁：**广义上的可重入锁指的是可重复可递归调用的锁，在外层使用锁之后，在内层仍然可以使用，并且不发生死锁（前提得是同一个对象或者class），这样的锁就叫做可重入锁、即在执行对象中所有同步方法不用再次获得锁。ReentrantLock和synchronized都是可重入锁。举个简单的例子，当一个线程执行到某个synchronized方法时，比如说method1，而在method1中会调用另外一个synchronized方法method2，此时线程不必重新去申请锁，而是可以直接执行方法method2。

**可中断锁：**在等待获取锁过程中可中断。Synchronized就是不可中断，而Lock是可中断锁。

**公平锁：**按等待获取锁的线程的等待时间进行获取，等待时间长的具有优先获取锁权利。非公平锁即无法保证锁的获取是按照请求锁的顺序进行的，这样就可能导致某个或者一些线程永远获取不到锁。Synchronized是非公平锁，它无法保证等待的线程获取锁的顺序。对于ReentrantLock和ReentrantReadWriteLock，默认情况下是非公平锁，但是可以设置为公平锁。

**读写锁：**对资源读取和写入的时候拆分为2部分处理，一个读锁和一个写锁。读的时候可以多线程一起读，写的时候必须同步地写。ReadWriteLock就是读写锁，它是一个接口，ReentrantReadWriteLock实现了这个接口。可以通过readLock()获取读锁，通过writeLock()获取写锁。

**悲观锁和乐观锁**

**乐观锁：**很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会去判断在此期间有没有人去更新这个数据（可以使用版本号等机制）。如果因为冲突失败就重试。乐观锁适用于写比较少的情况下，即冲突比较少发生，这样可以省去了锁的开销，加大了系统的整个吞吐量。

像数据库提供的类似于write\_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

**悲观锁：**总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，因此每次拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁，效率比较低。传统的关系型数据库里面就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如Java里面的同步原语synchronized关键字的实现也是悲观锁。

**乐观锁的实现方式（CAS）**

乐观锁的实现注意就两个步骤：冲突检测和数据更新。其实现方式有一种比较典型的就是Compare and Swap（CAS）。

CAS：CAS是乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

CAS操作中包含三个操作数——需要读写的内存位置（V）、进行比较的预期原值（A）和拟写入的新值（B）。如果内存位置V的值与预期原值A相匹配，那么处理器会自动将该位置值更新为新值B。否则处理器不做任何操作。无论哪种情况，它都会在CAS指令之前返回该位置的值。（在CAS的一些特殊情况下将仅返回CAS是否成功，而不提取当前值。）CAS有效地说明了“我认为位置V应该包含值A；如果包含该值，则将B放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”这其实和乐观锁的冲突检查+数据更新的原理是一样的。

**乐观锁是一种思想，CAS是这种思想的一种实现方式。**

**CAS的缺点：**

**ABA问题：**如果内存地址V初次读取的值是A，并且在准备赋值的时候检查到它的值仍然是A，那我们就能说它的值没有被其他线程改变过了么？如果在这段期间它的值曾经被改成了B，后来又被改回为A，那CAS操作就会误认为它从来没有被改变过。这个漏洞称为CAS操作的“ABA”问题。AVA并发包为了解决这个问题，提供了一个带有标记的原子引用类“AtomicStampedReference”，它可以通过控制变量值的版本来保证CAS的正确性。因此，在使用CAS前要考虑清楚“ABA”问题是否会影响程序并发的正确性，如果需要解决ABA问题，改用传统的互斥同步可能会比原子类更高效。

**循环时间长开销很大：**自旋CAS（不成功，就一直循环执行，直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。

**只能保证一个共享变量的原子操作：**当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁来保证原子性。

**死锁**

什么是死锁：两个进程都在等待对方执行完毕才能继续往下执行的时候就发生了死锁。结果就是两个进程都陷入了无限的等待中。

**产生死锁的四个必要条件：**

**互斥条件：**一个资源每次只能被一个进程使用。

**请求与保持条件：**一个进程因请求资源而阻塞时，对已获得的资源保持不放。

**不剥夺条件**：进程已获得的资源，在未使用完之前，不能强行剥夺。

**循环等待条件：**若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只有系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

**死锁的解决方法：**

* 撤销陷于死锁的全部进程；
* 逐个撤销陷于死锁的进程，直到死锁不存在；
* 从陷于死锁的进程中逐个强迫放弃所占用的资源，直至死锁消失
* 从另外一些进程那里强行剥夺足够数量的资源分配给死锁进程，以接触死锁状态

**如何确保N个线程可以访问N个资源同时又不导致死锁？**

使用多线程的时候，一种非常简单的避免死锁的方式就是：指定获取锁的顺序，并强制按照指定的顺序获取锁。因此，如果所有的线程都是以同样的顺序加锁和释放锁，就不会出现死锁了。

# 设计模式

设计模式代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。

设计模式一套被反复使用的，多数人知晓的，经过分类编目的，代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。毫无疑问，设计模式已于他人于系统都是多赢的，设计模式使代码编制真正的工程化。

共23种

创建性模式：这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用new运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。

包括：工厂模式（Factory Pattern）、抽象工厂模式（Abstract Factory Pattern）、单例模式（Singleton Pattern）、建造者模式（Builder Pattern）、原型模式（Prototype Pattern）。

结构型模式：这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。

包括：适配器模式（Adapter Pattern）、桥接模式（Bridge Pattern）、过滤器模式（Filter、Criteria Pattern）、组合模式（Composite Pattern）、装饰器模式（Decorator Pattern）、外观模式（Façade Pattern）、享元模式（Flyweight Pattern）、代理模式（Proxy Pattern）。

行为型模式：这些设计模式特别关注对象之间的通信。

包括：责任链模式（Chain of Responsibility Pattern）、命令模式（Command Pattern）、解释器模式（Interpreter Pattern）、迭代器模式（Iterator Pattern）、中介者模式（Mediator Pattern）、备忘录模式（Memento Pattern）、观察者模式（Observer Pattern）、状态模式（State Pattern）、空对象模式（Null Object Pattern）、策略模式（Strategy Pattern）、模板模式（Template Pattern）、访问者模式（Visitor Pattern）。

J2EE模式：这些设计模式特别关注表示层。这些模式是由Sun Java Center鉴定的。

包括：MVC模式（MVC Pattern）、业务代表模式（Business Delegate Pattern）、组合实体模式（Composite Entity Pattern）、数据访问对象模式（Data Access Object Pattern）、前端控制器模式（Front Controller Pattern）、拦截过滤器模式（Intercepting Filter Pattern）、服务定位器模式（Service Locator Pattern）、传输对象模式（Transfer Object Pattern）。

设计模式的六大原则：

1. 开闭原则：

对扩展开房，对修改关闭。为了使程序的扩展性好，易与维护和升级。想要达到这样的效果，我们需要使用接口和抽象类。

1. 里氏代换原则：

里氏代换原则是面向对象设计的基本原则之一。里氏代换原则中说，任何基类可以出现的地方。子类一定可以出现。LSP是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。里氏代换原则对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

1. 依赖倒转原则：

这个原则是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。

1. 接口隔离原则：

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。它还有另外一个意思是：降低类之间的耦合度。由此可见，其实设计模式就是从大型软件架构出发、便于升级和维护的软件设计思想，它强调降低依赖，降低耦合。

1. 迪米特法则，又称最少知道原则

最少知道原则是指：一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块的相对独立。

1. 合成复用原则

合成复用原则是指：尽量使用合成、聚合的方式，而不是使用继承。

1. 工厂模式

工厂模式是JAVA中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。在工厂模式中，我们在创建对象时不会对客户端暴漏创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

意义：定义一个创建对象的接口，让其子类自己决定示例化哪一个工厂类，工厂模式使其创建过程延迟到子类子类进行。

主要解决接口选择的问题。

何时使用：我们明确地计划不同条件下创建不同实例时。

如何解决：让其子类实现工厂接口，返回的也是一个抽象的产品。

关键代码：创建过程在其子类执行。

注：作为一种创建类模式，在任何需要生成复杂对象的地方，都可以使用工厂方法模式。有一点需要注意就是复杂对象适合使用工厂模式，而简单对象，特别是只需要通过new就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式。就需要引入一个工厂类，会增加系统的复杂度。

1. 抽象工厂模式：

围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

意图：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

主要解决：主要解决接口选择的问题。

1. 单例模式

单例模式是Java中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。这种模式涉及到一个单一的类，该类负责创建自己的对象，同事确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问。不需要实例化该类的对象。

注：

（1）单例类只能有一个实例。

（2）单例类必须自己创建自己的唯一实例。

（3）单例类必须给所遇其他对象提供这一实例

意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

主要解决：一个全局使用的类频繁地创建与销毁。

如何解决：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。

优点：（1）在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例。

（2）避免对资源的多重占用（比如写文件操作）。

缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

1. 建造者模式

建造者模式使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

一个Builder类会一步一步构造最终的对象。该Builder类是独立于其他对象的。

意图：将一个浮躁的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

主要解决：主要解决在软件系统中，有时候面临着“一个复杂对象”的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将他们组合在一起的算法却相对稳定。

关键代码：建造者：创建和提供实例，导演：管理建造出来的实例的依赖关系。

使用场景：1、需要生成的对象具有复杂的内部结构。2、需要生成的对象内部属性本身相互依赖。

注意：与工厂模式的区别：建造者模式更加关注与零件装配的顺序。

1. 原型模式

原型模式是用于创建重复的对象，同时又能保证性能。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式是实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。例如，一个对象需要在一个高代价的数据库操作之后被创建。我们可以缓存该对象，在下一个请求时返回它的克隆，在需要的时候更新数据库，以此来减少数据库的调用。

意图：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

主要解决：在运行期建立和删除原型。

如何解决：利用已有的一个原型对象，快速地生成和原型对象一样的实例。

优点：1、性能提高。2、逃避构造函数的约束。

缺点：1、配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。

2、必须实现Cloneable接口。

注意：与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的。浅拷贝实现Cloneable，重写，深拷贝拷贝是通过实现Serializable读取二进制流。

1. 适配器模式

作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构性模式，它结合了两个独立接口的功能。

这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。

意图：将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

主要解决：在软件系统中，常常要将一些“现存的对象”放到新的环境中，而新环境要求的接口是现对象不能满足的。

如何解决：继承或依赖。

优点：1、可以让任何两个没有关联的类一起运行。

2、提高了类的复用。

3、增加了类的透明度。

4、灵活性好。

缺点：过多地使用适配器，会让系统非常零乱，不易整体进行把握。

注意：适配器不是详细设计时候加的，而是解决正在服役的项目问题。

1. 桥接模式

桥接是用于把抽象化与实现化解耦，使得二者可以独立变化。这种类型的设计模式属于结构型模式，它通过抽象化和实现化之间的桥接结构，来实现二者的解耦。

这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。

意图：将抽象部分与实现部分分离，使它们都可以独立的变化。

主要解决：在有多种可能会变化的情况下，用继承会造成类爆炸问题，扩展起来不灵活。

如何解决：把这种多角度分类分离分离出来，让它们独立变化，减少它们之间的耦合。

关键代码：抽象类依赖实现类。

优点：1、抽象和实现的分离。2、优秀的扩展能力。3、实现细节对客户透明。

缺点：桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。

# Git



1. 安装Git

在Git官网上下载安装程序，git->Git Bash，蹦出一个类似命令行窗口的东西，说明Git安装成功。

安装完成后，还需要最后一步设置，在命令行输入：

git config --global user.name “your Name”

git config --global user.email [email@example.com](mailto:email@example.com)

查看用户名和邮箱

git config user.name

git config user.email

1. 创建本地仓库

git init

1. 将文件添加到仓库中

git add 文件名

git add . 将文件的修改，文件的新建，添加到暂存区。

git add -u --update 将文件的修改、文件的删除，添加到暂存区。

git add -A --all 将文件的修改，文件的删除，文件的新建，添加到暂存区。git commit -m “xxx” 将添加的文件提交到仓库中

git status 查看运行结果

git diff 文件名 具体查看修改

git log 历史记录

git reset --hard commit\_id 回退版本

git reflog 查看命令历史

git checkout -- 文件名 撤销修改

git add将文件添加到暂存区，git commit 将暂存区的所有内容提交到当前分支

1. 远程仓库
2. 创建SSH Key。在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有id\_rsa和id\_rsa.pub这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：

$ ssh-keygen -t rsa -C [youremail@example.com](mailto:youremail@example.com)

你需要把邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的秘钥对，id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人。

1. 登陆GitHub，打开“Account settings”，“SSH Keys”页面：

然后，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴id\_rsa.pub文件的内容：

Add key

1. 登陆GitHub，然后，在右上角找到“Create a new repo”按钮，创建一个新的仓库：

然后在本地运行命令

$ git remote add origin git@github.com:michaelliao/learngit.git

$ git push -u origin master 推送到远程仓库

由于远程库是空的，我们第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

$ git push origin master 以后推送远程仓库

$ git clone [git@github.com:michaelliao/gitskills.git](mailto:git@github.com:michaelliao/gitskills.git) 拉取远程仓库

git remote -v 查看远程仓库详情

1. 分支管理

git branch 查看所有分支，当前分支前面会标一个\*号。

git branch dev 创建分支dev

git checkout dev 切换分支dev

git checkout -b dev 创建并切换分支dev

git merge dev 将dev分支合并到当前分支

git branch -d dev 删除dev分支

1. Bug分支

git stash 将当前工作区储存起来

git stash list 查看存储的工作区

git stash apply 恢复工作区

git stash drop 删除存储的工作区

git stash pop 恢复工作区并删除存储的工作区

git stash apply stash@{0} 恢复指定的工作区

1. Git一些命令

$ git remote

为了便于管理，Git要求每个远程主机都必须指定一个主机名。git remote命令就用于管理主机名。不带选项的时候，git remote命令列出所有远程主机。

$ git remote show <主机名>

查看该主机的详细信息

$ git remote add <主机名> <网址>

用于添加远程主机

$ git remote rm <主机名>

用于删除远程主机

$ git remote rename <原主机名> <新主机名>

用于重命名远程主机

$ git fetch <远程主机名>

一旦远程主机的版本库有了更新（Git术语叫做commit），需要将这些更新取回本地，这时就要用到git fetch命令。

$ git fetch <远程主机名>

将某个远程主机的更新，全部取回本地。git fetch命令通常用来查看其他人的进程，因为它取回的代码对你本地的开发代码没有影响。默认情况下，git fetch取回所有分支（branch）的更新。如果只想取回特定分支的更新，可以指定分支名。

$ git fetch <远程主机名> <分支名>

取回某主机主机的某分支。

$ git pull

git pull命令的作用是，取回远程主机某个分支的更新，再与本地的指定分支合并。它的完整格式稍稍有点复杂。

$ git pull <远程主机名> <远程分支名>:<本地分支名>

比如，取回origin主机的next分支，与本地的master分支合并，需要写成下面这样。

$ git pull origin next:master

如果远程分支是与当前分支合并，则冒号后面的部分可以省略。

$ git pull origin next

上面命令表示，取回origin/next分支，再与当前分支合并。实质上，这等同于先做git fetch，再做git merge。

在某些场合，Git会自动在本地分支与远程分支之间，建立一种追踪关系（tracking）。比如，在git clone的时候，所有本地分支默认与远程主机的同名分支，建立追踪关系，也就是说，本地的master分支自动"追踪"origin/master分支。

git branch --set-upstream master origin/next

上面命令指定master分支追踪origin/next分支。

如果当前分支与远程分支存在追踪关系，git pull就可以省略远程分支名。

$ git pull origin

上面命令表示，本地的当前分支自动与对应的origin主机"追踪分支"（remote - tracking branch）进行合并。如果当前分支只有一个追踪分支，连远程主机名都可以省略。

$ git pull --rebase <远程主机名> <远程分支名>:<本地分支名>

如果远程主机删除了某个分支，默认情况下，git pull 不会在拉取远程分支的时候，删除对应的本地分支。这是为了防止，由于其他人操作了远程主机，导致git pull不知不觉删除了本地分支。但是，你可以改变这个行为，加上参数 -p 就会在本地删除远程已经删除的分支。

$ git push

git push命令用于将本地分支的更新，推送到远程主机。它的格式与git pull命令相仿。

$ git push <远程主机名> <本地分支名>:<远程分支名>

注意，分支推送顺序的写法是<来源地>:<目的地>，所以git pull是<远程分支>:<本地分支>，而git push是<本地分支>:<远程分支>。如果省略远程分支名，则表示将本地分支推送与之存在"追踪关系"的远程分支（通常两者同名），如果该远程分支不存在，则会被新建。

$ git push origin master

上面命令表示，将本地的master分支推送到origin主机的master分支。如果后者不存在，则会被新建。如果省略本地分支名，则表示删除指定的远程分支，因为这等同于推送一个空的本地分支到远程分支。

$ git push origin

上面命令表示，将当前分支推送到origin主机的对应分支。如果当前分支只有一个追踪分支，那么主机名都可以省略。

$ git push --all origin

上面命令表示，将所有本地分支都推送到origin主机。

如果远程主机的版本比本地版本更新，推送时Git会报错，要求先在本地做git pull合并差异，然后再推送到远程主机。这时，如果你一定要推送，可以使用--force选项。

$ git push --force origin

上面命令使用--force选项，结果导致远程主机上更新的版本被覆盖。除非你很确定要这样做，否则应该尽量避免使用--force选项。

# Redis

Redis是一个key-value存储数据库。

**特点：**

* Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。
* Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。
* Redis支持数据的备份，即master-slave模式的数据备份。

**优点：**

* 性能极高——Redis能读的速度是110000次/s，写的速度是81000次/s。
* 丰富的数据类型——Redis支持二进制案例的Strings，Lists，Hashes，Sets及Ordered Sets数据类型操作。
* 原子——Redis的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过MULTI和EXEC指令包起来。
* 丰富的特性——Redis还支持publish/subscribe，通知key过期等。

# Elasticsearch

# springCloud

SpringCloud是一些列框架的有序集合。它利用SpringBoot的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用SpringBoot的开发风格做到一键启动和部署。

1. spring
2. Java反射 classforname
3. 设计模式

单例模式 简单工厂模式 工厂方法模式 抽象工厂模式 代理模式 适配器模式 原型模型 （深拷贝，浅拷贝）

1. 自定义RPC框架
2. JVM，几种内存模型，存的信息，作用
3. 垃圾收集器的算法，有几种算法，标记清除，复制算法，标记整理，分代收集，垃圾收集器有几种。Serial，ParaNew，Parall Scavenge、Serial old, CMS,G1

JVM的参数

1. GC的种类 MinorGC、MajorGC、FullGC，
2. 
3. 并发：哪几种线程池，什么是线程，四种new线程的方式，thread接口, Runnable接口 callable接口，线程池
4. 创建线程池的大概有4-5种方式，进程和线程的区别
5. 锁
6. 过滤器和拦截器和AOP @Pointcut
7. Start 和run区别
8. 异常处理
9. 线程的状态
10. 算法，冒泡，快速，插入，希尔，堆，选择，归并，二分法查找
11. 数据结构，List，Map,Set
12. Get和Post区别
13. 多线程高并发消息队列

https://github.com/doocs/advanced-java

https://github.com/Snailclimb/JavaGuide/

Ajax缓存

Java抛出的异常

存储过程

抽象函数和接口

Oracle分页