**目录**

[JDK、JRE、JVM 2](#_Toc18656)

[对象占多少存储空间 3](#_Toc28070)

[初始化&垃圾回收 4](#_Toc31401)

[finalize() 4](#_Toc22095)

[System.gc() 5](#_Toc7484)

[初始化顺序 6](#_Toc21763)

[域、方法、类的访问权限 7](#_Toc31096)

[继承 7](#_Toc5235)

[代理 8](#_Toc10081)

[@Override 8](#_Toc3109)

[final关键字 8](#_Toc27037)

[final数据 8](#_Toc16536)

[final方法 9](#_Toc30068)

[final类 9](#_Toc20905)

[初始化 9](#_Toc5144)

[多态 10](#_Toc12624)

[构造方法相关的多态 11](#_Toc12046)

[状态模式 12](#_Toc27862)

[多态小案例 13](#_Toc10468)

[接口 14](#_Toc1340)

[面向对象 14](#_Toc11421)

[设计模式 14](#_Toc30578)

[工厂模式 15](#_Toc12666)

[接口中的域 16](#_Toc23423)

[枚举 17](#_Toc7687)

[内部类 17](#_Toc31193)

[访问权限 17](#_Toc3222)

[使用内部类实现某某接口 19](#_Toc28787)

[匿名内部类 20](#_Toc1168)

[嵌套类 21](#_Toc3408)

[Collection 21](#_Toc22333)

[是否需要向上转型？ 21](#_Toc29339)

[java.util.Collections 22](#_Toc1021)

[toString() 22](#_Toc30870)

[List（链表） 22](#_Toc26367)

[常用方法 22](#_Toc9177)

[LinkedList 23](#_Toc20217)

[迭代器 24](#_Toc31420)

[快速失败(fail-fast)与安全失败(fail-safe) 24](#_Toc14714)

[Set 25](#_Toc6881)

[HashMap 25](#_Toc12985)

[月经问题 26](#_Toc14391)

[ArrayList与LinkedList的比较 26](#_Toc32031)

[异常 26](#_Toc14574)

[是否需要try-catch 27](#_Toc1500)

[throws 29](#_Toc32308)

[finally 30](#_Toc5945)

[异常丢失 31](#_Toc10164)

[异常的限制 31](#_Toc32405)

[猜想 32](#_Toc20523)

[异常总结 32](#_Toc19242)

[字符串 33](#_Toc29589)

[无意识的递归 33](#_Toc13323)

[格式化输出 33](#_Toc29623)

[正则表达式 34](#_Toc26360)

[部分匹配与完全匹配 34](#_Toc5655)

[Scanner 35](#_Toc20224)

[String类常用方法 36](#_Toc28325)

[类型信息 36](#_Toc6409)

[省代码的案例 38](#_Toc1174)

[反射相关的类 39](#_Toc11697)

[突破权限控制 40](#_Toc11082)

[动态代理 41](#_Toc21291)

[泛型 41](#_Toc5942)

[为什么要泛型 41](#_Toc32510)

[泛型的第一种用法 42](#_Toc14106)

[Comparable & Comparator 42](#_Toc24280)

[接口与抽象类的异同点 43](#_Toc23580)

[Object 43](#_Toc6347)

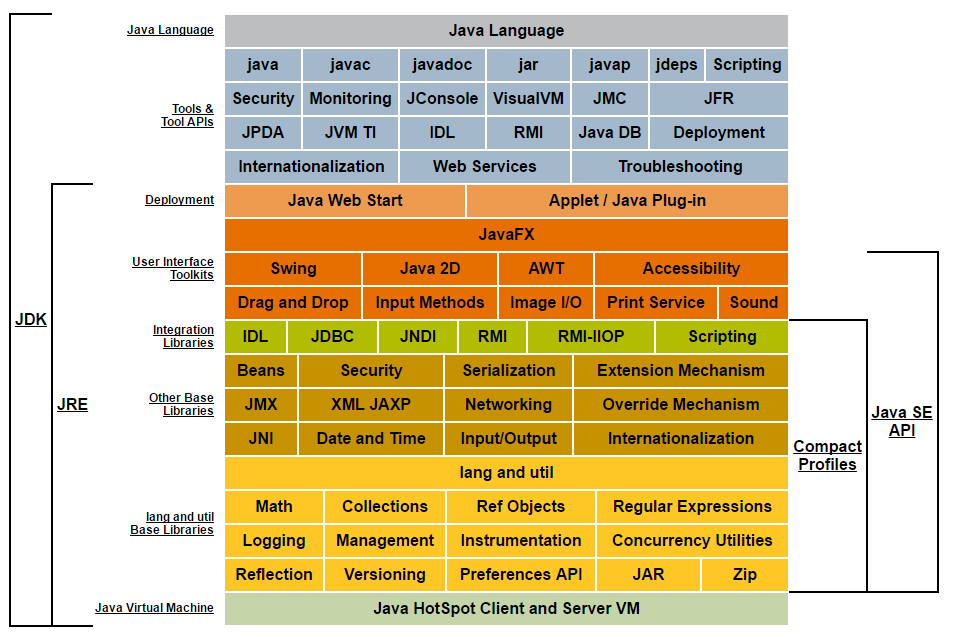
[操蛋的时间 44](#_Toc20343)

[ASCII码 45](#_Toc1719)

[ecplise插件 46](#_Toc10974)

# **JDK、JRE、JVM**

从左往右，这是一个包含关系，JDK包含JRE，JRE包含JVM。我觉得没有什么比这张图来的更实在了。



这个包含关系真是不言而喻啊，JDK由javadoc,javap,jdb等组件及JRE组成；JRE由核心类库及JVM组成；JVM是一个虚构出来的计算机，它将.class字节码文件编译成对应的机器语言。

Java 运行已编译完的.class文件

Javac 编译.java文件并生成.class文件

Javadoc 将注释变成文档

Jar 平台无关的文件格式，可以将多个Java程序，连带着.class在内合并为 一个jar文件。

**///////////////////////////////////////////////////////////////////////////////////**

# 对象占多少存储空间

http接口返回一堆数据，从数据库内查询出一堆数据，程序接收，并封装成对象。那么问题来了，这些对象在内存内占多少空间？有时候，看一个http接口返回500kb的数据我就感觉害怕，总有一种服务器要爆了的感觉。当然这是开玩笑了，服务器怎么会爆，充其量宕机。

之前一直不知道怎么计算对象在内存上的大小。其实是思维太死，一个java对象，说到底就是由field与method组成的。method我先不管，field，这个就很好理解了，基础类型，包装类型，或者类类型。不过真的仔细追踪下去，其实不管什么类类型，到头来都是由基础类型组成的，君不见数据库也就存存数字、字符串……。因此，以model类为例，这玩意就像一个套娃，不停的把基础类型包来包去。

而基础类型有多大，这是有规定的。理论上通过递归加反射，是可以计算出一个model类的对象至少占多少字节的。

boolean 占1 bit

char 占16 bit（2字节）

short 占16 bit（2字节）

int 占32bit（4字节）

long 占64bit（8字节）

float 占32bit（4字节）

double 占64bit（8字节）

至于程序本身是否在内存里占有一定空间，这他妈也是一门学问，奈何知识水平有限啊。

**///////////////////////////////////////////////////////////////////////////////////**

# 初始化&垃圾回收

## finalize()

不能妖魔化这个方法。

我们先看源码

**protected** **void** finalize() **throws** Throwable { }

一共只有1行，压根就是个空方法。

当某对象被垃圾回收时，此对象的finalize()方法会被调用，但因为默认的finalize()方法是空的，所以如果不重载它，那么什么都不会发生。

什么情况下需要重载呢。

由于垃圾回收仅能释放关键字new创建的对象。因此如果一个对象中的部分存储空间是由C的malloc()方法赋予的，那么只能调用相应的free()方法才能释放，垃圾回收机制起不了作用。解决方案是覆盖Object的finalize()方法，在里面用本地方法调用free()或其他的释放函数。

至于为什么会出现C语言的malloc()函数，是因为Java存在native方法。

还有，垃圾回收不是一定会发生的，因为垃圾回收本身有开销，所以不到程序内存不够用的时候，不会去浪费性能做垃圾回收。

*垃圾回收的工作原理“假定”是这样的：一旦垃圾回收器准备好释放对象占用的存储空间，将首先调用其finalize()方法，并且在下一次垃圾回收动作发生时，才会真正回收对象占用的内存。所以要是你打算用finalize()，就能在垃圾回收时刻做一些重要的清理工作。*

————thinking in Java

因此finalize()还有另一种用法，终结条件，就是在finalize()里加if-else，并输出相关信息，老实讲，我觉得这更像log。

## System.gc()

与Runtime.getRuntime().gc()等价

强制进行回收动作

不过看文档，这个方法也只是建议JVM此刻进行垃圾回收，然后JVM会尽可能的去释放它认为可以被回收的内存空间。这个方法不是回收全部对象，也就是说方法调用结束后，某些你希望被回收的对象可能依旧存在。

/\*\*

\* Runs the garbage collector.

\* <p>

\* Calling the <code>gc</code> method suggests that the Java Virtual

\* Machine expend effort toward recycling unused objects in order to

\* make the memory they currently occupy available for quick reuse.

\* When control returns from the method call, the Java Virtual

\* Machine has made a best effort to reclaim space from all discarded

\* objects.

\* <p>

\* The call <code>System.gc()</code> is effectively equivalent to the

\* call:

\* <blockquote><pre>

\* Runtime.getRuntime().gc()

\* </pre></blockquote>

\*

\* **@see** java.lang.Runtime#gc()

\*/

## 初始化顺序

结论第一：

先初始化静态变量静态代码块，然后初始化普通变量普通代码块，最后才是构造函数。如果有父类，那就先父类再子类。另外，静态变量及代码块只会初始化一次。

初始化代表一个动作，这个动作概括起来就是，当new一个类的对象时，为该类成员变量赋默认值。

class A{

static int j;

int i;

String str;

}

当 A a = new A(); 时，变量j第一个有默认值0，然后变量i的默认值为0，变量str的默认值为null。

粗暴来看，好像不new对象，这一切都不会发生，可事实上却存在一种特殊情况。由于static变量可以直接通过类来使用（像下面）

A.j;

因此，只要有地方这样用了静态变量，那么即便没有new对象，静态成员们还是会被

初始化。注意“们”这个字，静态变量以及静态代码块都会被初始化，很团结、很社会主义的。然后又由于静态成员只会被初始化一次，所以下次开始new对象时，静态成员们就不在参与初始化过程了。

栗子：

**public** **class** Work {

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

System.***out***.println(A.*j*);

**new** A();

}

}

**class** A {

**static** **int** *j*;

**static** {

System.***out***.println("第一个静态块：j被初始化了");

}

**static** {

System.***out***.println("第二个静态块：");

}

}

**///////////////////////////////////////////////////////////////////////////////////**

# 域、方法、类的访问权限

这个知识点很简单，对于域与方法。

private是类内访问权限。

啥都不写是包内访问权限。

protected除包内访问权限外，继承后的子类也可以访问。

public是，只要import了这个类，可以尽情访问。

有意思的是这个：

public class A{}

类的前面也有“public”关键字。其实这个关键字相当重要，因为类名前如果没有“public”，那么这个类是不能被import的。

类名前不加public也不会报错，这样就代表包内访问权限，粗暴点，只有在这个包内，才能创建这个类的对象。

类名前加private和protected就要报错了。搞不懂创始团队脑子里怎么想的，其实这个特性，基本来讲用不到，就默认类名前都要加public关键字就好了。

**///////////////////////////////////////////////////////////////////////////////////**

# 继承

new子类对象时，编译器会自动去new父类对象，然后这个父类对象就一直隐藏在子类对象里。

针对这条规则，这里头有个生僻的知识点。

new这个动作必然通过构造方法，如果父类仅有一个构造方法且无参，那么编译器会自动调用。如果父类仅有一个构造方法且带参，那么子类就必须显示调用了，不然就要报错。

**public** **class** A {

**private** **int** i;

**public** A(**int** i) {

**this**.i = i;

}

}

**public** **class** Work **extends** A {

**public** Work() {

**super**(1);

}

}

// 感觉没什么卵子用，现在的企业级项目，因为分工明确，已经不依赖构造函数这种东西了，基本都是get&set方法。

## 代理

这是一种设计模式。

public class A{

void doSomething(){

// syso ……  
}

}

public class AProxy{

A a = new A();

void doSomething(){

// syso ……

a.doSomething();

}

}

真正使用这种模式的时候不可能真的像上面那样把A对象new出来，可以通过各种配置参数来获取想要的实例。

真正使用这种模式的时候也不可能向上面那样，代理的方法只是很单纯的调用一下A对象的方法，总要在干点别点什么。

## @Override

表示子类覆盖父类的方法，可写可不写但最好还是写上。因为：（底下我百度来的）

1．可以当注释用，方便阅读．

2．告诉阅读你代码的人，这是方法的复写．

3．编译器可以给你验证@Override下面的方法名是否是你父类中所有的，如果没有则报错．

## final关键字

### final数据

如果final的是基本类型数据，该数据的值便不允许在改变。

如果final的是类类型，该引用便被固定，但也只是引用被固定，不能指向其它类对象。你要想改这个引用指向的对象还是能改的。

空白final，这个知识点就有点生僻了，这么久了我也没见过哪个傻吊是这么干的。一旦建立了一个空白final，就必须保证有个地方会初始化空白final，不然编译器会直接报错。

class A{

static final int i;

}

这段代码就是错的。

class A{

static final int i;

static {

i = 1;

}

}

这样就是对的。如果成员变量不是static的，那么不通过代码块，在构造方法里初始化也行。

有时候方法的参数列表也会被声明为final。

class A{

void bbb(final int i){

// i = 1; // 这么写就要报错了，因为i是final类型，不允许被改变

}

}

### final方法

那么这个方法不能被子类覆盖。

### **final类**

那么这个类不能被继承。

## 初始化

对，又是初始化。

**class** B{

**static** {

System.***out***.println("静态代码块被执行");

}

**public** B() {

System.***out***.println("创建B对象");

}

}

**public** **class** A **extends** B {

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

// new A();

}

}

这个例子相当生动的展示了，初始化子类前必然初始化父类这一规则。

new A()这种老生常谈就不看了，关键是把new A()这行代码注释掉，你仍能看见静态代码块内的输出语句。

因为即便只访问一个空的main方法，仍旧需要对类A进行初始化，只是这次不会初始化全部成员，而只初始化静态成员。

详细过程是，编译器尝试访问A.main()方法，为了访问A.main()方法，编译器会查看A类的所有代码，此时编译器注意到A还有个父类B，于是编译器继续加载父类B。发现父类B存在静态成员，那就初始化该静态成员。

**///////////////////////////////////////////////////////////////////////////////////**

# 多态

概念是先人们经历各种实践，越过各种雷坑总结出来的经验。

管用，却不讨喜。

多态又叫动态绑定、运行时绑定，粗暴讲就是子类覆盖父类的方法，然后父类引用调用子类对象时会采用子类的方法实现。

域、私有方法、静态方法，是不具备多态性的。

class A{

int i;

private void fo(){

System.out.println("我是A的私有方法");

}

static void foo(){

System.out.println("我是A的静态方法");

}

}

class B extends A{

int i;

private void fo(){

System.out.println("我是B的私有方法");

}

static void foo(){

System.out.println("我是B的静态方法");

}

public static void main(String[] args) {

A a = new B();

a.i;

a.fo();

a.foo();

}

}

以上代码，经管A引用背后是子类B的对象，但方法调用仍旧是A的实现。因为这是父类A的引用，而域、私有方法、静态方法，是不具备多态性的。

这个屌知识点真的有用吗。。。。懂跟不懂都不影响写代码。

## 构造方法相关的多态

这又是一个生僻的知识点。

多态是指当父类引用指向子类对象时，调用方法会采用子类实现。

**public** **class** A {

A() {

System.***out***.println("构造A对象");

foo();

}

**void** foo() {

System.***out***.println("A的foo方法");

}

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

A a = **new** B();

}

}

**class** B **extends** A {

B() {

System.***out***.println("构造B对象");

foo();

}

**void** foo() {

System.***out***.println("B的foo方法");

}

}

这个是执行后的结果：

构造A对象

B的foo方法

构造B对象

B的foo方法

因为多态的缘故，new B()对象时，A类构造方法里的foo()方法会采用B类实现。

这个屌知识点真的用得到吗。

我还是那个想法，现在企业级项目，分工非常明确，尤其是引入Spring这个框架后，连对象都不用自己new了，接触到构造方法的机会真是少之又少。就算接触到，大部分情况下，构造方法也都是空的。

## 状态模式

class Actor{

void play(){}

}

class ActorA{

void play(){

System.***out***.println("A play()");

}

}

class ActorB{

void play(){

System.***out***.println("B play()");

}

}

class Stage{

Actor actor;

void setActor(Actor a){

actor = a;

}

void show(){

actor.play();

}

public static void main(String[] args){

Stage s = new Stage();

s.setActor(new ActorA);

s.show();

s.setActor(new ActorB);

s.show();

}

}

代码不变，但通过切换状态就可以获得不同的效果。

// 只是看得懂，没实际用过。

## 多态小案例

class A {

public void start(){

a();

b();

c();

}

protected void a(){}

protected void b(){}

protected void c(){}

}

class Worker extends A{

protected void a(){

// 做点别的

}

}

这样的好处，就是定义好一个完整的执行流程，这一步做什么，下一步做什么。然后流程不动，若相应的某一步需要特殊处理，那就覆盖一下。

**///////////////////////////////////////////////////////////////////////////////////**

# 接口

## 面向对象

面向对象是一种思维。一种很好理解，但是非常难实践的思维。为了更好的理解面向对象，要先知道什么是面向过程。我自然是没那个本事解释这两个概念，但看过一个非常有趣的例子。

面向过程就是用代码去描述一件事的过程，以愤怒的小鸟为例。

红色的小鸟砸中猪，一个过程。

黄色的会加速的小鸟砸中猪，一个过程。

黑色的会爆炸的小鸟砸中猪，一个过程。

…… ……

红色的小鸟砸中小猪，一个过程。

红色的小鸟砸中大猪，一个过程。

红色的小鸟砸中一群猪，一个过程。

…… ……

这些场景用面向过程来实现的话，最后会得到一堆代码。做一件事，就要写一段代码，最后编译出来的文件数量会非常吓人。

而如果面向对象，那么定义好一个鸟接口，各自怎么飞，有什么特殊技能，会爆还是会分身交给子类实现。再定义好一个猪接口，各自什么体型，被撞后怎么死，也交给子类实现。

写代码的时候，用最顶层的鸟引用调用方法去撞，用最顶层的猪引用调用方法被撞。

然后这段代码就固定不动了。

唯一的变动是提供各式各样的鸟对象，猪对象。

所以面向对象这种思维并不难理解，就是选取一个事物，然后用代码描述它，为它定义方法，然后给别人调用。愤怒的小鸟这个例子里，我们会很容易想到将鸟和猪抽离出来，当做对象去描述。

那么换件事，假设寄东西，要把哪些玩意从事件里抽离出来？

在复杂一些，换个行业，渡轮，设计一个管理渡轮的系统，又要抽象哪些玩意？

到这份上我觉得和人的智商都没关系了，全看对另一个行业的理解程度。

## 设计模式

这个屌东西吧，一年到头都有人在吹，而且不管是哪一门编程语言都有人在吹。好像不说两句它的好，就体现不了做为程序员的高贵。

但在我心目中它的玄乎程度就像面向对象编程一样。

一样让我觉得吃屎。

摸着良心讲，确实很屌，这东西是先贤们花了数十年功夫才总结出来的宝贵经验。但问题在于，就算搞明白了它的原理，还是不知道要怎么用。

面向对象的难体现在抽象上，也就是对另一行业的理解程度。

而策略模式，我觉得它难在个人总计代码行数上。就好比你没有操过一个女人，就永远无法描述那种妩媚、温润、悦耳乃至癫狂。

我还记得的模式，有策略、状态、工厂、代理、适配器。你要我说我大概也能说一点，以前为了面试背过一点原理，但给一个具体场景，问用哪种模式？

唔，好像都可以。

说实在的，真分辨不出来，因为这几个模式拆分到最后，就是接口、继承、覆盖这几个知识点组合来组合去。

可能真得代码写得多了，看着拷来拷去的代码看到自己都要吐的时候，才会灵光一闪，高喊一声“真相只有一个”，就它最合适。

不过该背的还是要背，不然连灵光一闪的机会都没有，嗯，天赋异禀的我就不考虑了。

### 工厂模式

interface Chess{

void play();

}

class ChineseChess implements Chess{

void play{

System.***out***.println("玩中国象棋");

}

}

interface Factory{

Chess getChess();

}

class ChineseChessFactory implements Factory{

ChineseChess getChess(){

return new ChineseChess();

}

}

public class Work{

void start(Factory factory){

factory.getChess().play();

}

public static void main(String[] args){

start(new ChineseChessFactory());

// start(其他工厂)

}

}

在这个例子里，start方法永远不需要变动，想玩其他棋重新实现工厂接口和棋接口就好了。

## 接口中的域

接口中是可以存在域的。（我个人喜欢管这叫成员变量）

接口中的域自动是public static final类型的。

public interface A{

int a;

}

那么如果上周讲的那个，关于final的生僻的知识点没忘的话，你会发现这段代码会报错。

编译器会提示你，需要为域a赋初值。

public interface A{

int a = 1;

}

这样才对。

不过，我也很少看见有人在接口里定义常量，又算一个生僻的知识点吧。

**///////////////////////////////////////////////////////////////////////////////////**

# 枚举

具名值

ordinal()

values()

**///////////////////////////////////////////////////////////////////////////////////**

# 内部类

// 见过但从来没用过的特性。

class A{

class B{}

}

把一个类的定义放在另一个类的定义内，被放进去的（上面那个B），就是内部类。

内部类这个东西看得人晕乎乎的，这东西给我的感觉像设计模式一样，只有经验到了，才会自然而然的明白为什么要用它。

而且thinking in java的作者也主动建议了，不要强行使用，能两个外部类解决就两个外部类解决。

## 访问权限

由于内部类在外部类的内部，根据java的访问权限原则，内部类能够访问外部类的所有成员，反过来，外部类也能访问内部类的所有成员。

下面这段代码不会报任何错。

**public** **class** A {

**private** **void** foo() {

}

**class** B {

**private** B() {

foo();

}

}

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

A a = **new** A();

A.B b = a.**new** B();

}

}

// A.B和a.new我也不知道要怎么解释，总之如果不是在A或B的普通方法内new内部类对象，就得这么写。怎么总觉的这段话越说越绕。

这时在引入一个类Work。

**public** **class** Work {

}

问类Work对内部类B的访问权限。

我觉得首先得搞懂访问权限这4个字到底什么意思。

首先方法分为两种，静态方法与非静态方法（java 8的default方法这里不考虑）。静态方法通过类名调用，非静态方法通过对象调用。

我的理解是，如果在Work类的方法里调用类B的静态方法不报错，或者，在Work类的方法里new B对象不报错。两者取其一，一者满足，那么类Work就拥有类B的访问权限；不能，类Work就没有类B的访问权限。

（对象都能new出来，那肯定可以通过对象调用类B的方法啦）

public class A {

private class B {

}

}

public class Work {

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

A a = new A();

A.B b = a.new B();

}

}

这个端代码会报错，因为B是private，只有A能new。

public class A {

class B {

}

}

public class Work {

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

A a = new A();

A.B b = a.new B();

}

}

不报错。但是如果Work与A不在一个包内，那就要报错。

public class A {

protected class B {

}

}

// 不在一个包内

public class Work extends A {

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

A a = new A();

A.B b = a.new B();

}

}

这样就不报错

public class A {

public class B {

}

}

public随便怎么搞。

## 使用内部类实现某某接口

现在我们来思考有毛用这个问题。

单纯在一个类A里，定义一个类B，并没有太多用处，就算类B里的方法切切实实派上用场，这种写法也只是帮java程序员偷懒而已。因为完全可以把类B拎出来，与类A平级。

内部类真正开始发光发热是从它实现接口并向上转型开始的。

这他妈的就又涉及到设计模式这四个神秘的字了。

鉴于本人也不是很懂，我只能抛个栗子，ArrayList与LinkedList类，都存在一个内部类，他们的内部类都实现了Iterator接口。点开iterator()方法的源码，其实就是new一个内部类的对象给我们用。

好了，记住这条准则，内部类想有大用场，就得实现某某接口。

外部类的非static方法可以随意创建内部类对象。Static方法除了要显示指明，还要通过外部类对象：

OuterClass.InnerClass in = out.new InnerClass ;

用 .new 创建毕竟还是麻烦，所以向下面那样额外写一个方法用来创建内部类对象会方便点。

**public** **class** Outer {

**class** Inner {

// TODO....

}

**public** Inner createInner() {

**return** **new** Inner();

}

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

Outer out = **new** Outer();

Outer.Inner in = out.createInner();

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

}

}

内部类可以随意访问外部类的所有元素。那这是为什么呢？当创建内部类对象时，此内部类对象必然会获取外围类对象的引用，内部类对象访问外围类对象成员就是通过这个引用。当然这是比较底层的知识了，一般程序员接触不到。

不扣底层知识的话还有另一种理解方式，private、default、protected、public是Java的四种访问权限。外部类成员变量private即类内访问权限，内部类在外部类内，因此可以随意访问变量及方法。

// 一点感想，访问对象，访问静态类，不存在单独访问变量这种说法，都是要通过媒介的

当将内部类向上转型为其基类，尤其是转型为一个接口时，内部类就有了用武之地。这是因为此内部类——某个接口的实现——能够完全不可见，并且不可用（？）。所得的只是指向基类或接口的引用，所以能够很方便的隐藏实现细节。

## 匿名内部类

目光看向匿名内部类，情况顿时有点复杂。（好吧夸张了）匿名内部类的出现，主要是有时候，你只需要调用某接口或某类的方法，犯不着特意去写个子类来实现或继承，完了还要创建对象。因此最常见的使用方式是 return new A(){...};

// 自己总结的不知道对不对

**interface** A {

**public** **void** say();

}

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

**public** A create(**final** **int** i){

**return** **new** A(){

**private** **int** b = i;

**public** **void** say(){

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

}

};

}

}

烦的是上面那种情况，有时候匿名内部类会需要使用外部变量，这时候传进方法内的引用必须是final类型。还有一种情况是，匿名内部类继承的那个类只有一个带参构造函数，那这时候新建匿名内部类对象的伪构造函数也要带参。

return new A(i){...};

匿名内部类只是一种比较吊诡的继承。。。。。（不知道这么理解好不好）

## 嵌套类

也叫静态内部类，被static修饰的内部类。

创建静态内部类（嵌套类）对象，不需要其外围类的对象。

静态内部类（嵌套类）不能访问非静态的外围类成员。

普通内部类不能包含静态成员，静态内部类（嵌套类）可以包含一切

**///////////////////////////////////////////////////////////////////////////////////**

# Collection

### 是否需要向上转型？

List<Integer> list = new ArrayList<Integer>();

刚学java的时候，一直非常好奇，为什么非要用List引用来接ArrayList对象。由此延伸出来的问题，LinkedList对象是不是也要用List接。HashMap，TreeMap是不是也非得用Map接。

答案是不一定。

ArrayList这种类，内部没什么有新意的方法，所以用List来接很ok，这么做也比较通用。但是LinkedList还有TreeMap这种类，都有除接口外的方法，而且这些方法还很有用，所以尽量不用顶层接口接。

### java.util.Collections

这是java原生的工具类，Google的一套工具类的命名方式，像Lists什么的估计就是从这学。

Collections.addAll()方法，运行效率会比普通集合对象的addAll()高。不过，仔细看源码的话，会发现，Collections.addAll()用起来有一定限制。

public static <T> boolean addAll(Collection<? super T> c, T... elements)

第二个参数是可变参数类型，这意味着你要么塞对象，要么塞对象数组，不能塞其他集合。

可能也是因为这个限制，我很少看到有人用这个方法。

而且，这个年代，写代码一般不考虑性能，就算考虑性能也不考虑这个。

并发的瓶颈不在这。

// 问题来了，既然不重要，为甚么我要花时间记这鸟玩意。。。。。。

### toString()

集合们（ArrayList，LinkedList，HashSet，TreeSet，LinkedHashSet……）统统实现了toString()方法，因此可以直接用println（）方法输出，届时，控制台上会出现很优雅的字符串。

不过反过来数组对象就没那么好说话了，直接println()，控制台上只会出现类名和该对象的hash值。要输出内容，必须用Arrays.toString（）方法。

Arrays类也是java原生的工具类。

## List（链表）

### 常用方法

// 就是这些常用方法我一个都没用过，有些尴尬。

public void add(int index, E element)

将元素插入指定位置。

这个方法不是List接口的，只有ArrayList与LinkedList的引用才能调用。

List<E> subList(int fromIndex, int toIndex);

截取片段，区间范围 [fromIndex , toIndex)。这个方法是List接口的。

boolean retainAll(Collection<?> c);

取交集，是List接口的。

boolean containsAll(Collection<?> c);

是否包含全部。

boolean remove(Object o) &boolean removeAll(Collection<?> c)

删除单个元素，删除全部元素，不过个人推荐用迭代器删。

### LinkedList

这个类比较特殊，从数据结构的角度讲，它实现了链表，实现了堆栈，还实现了队列。

public class LinkedList<E>

extends AbstractSequentialList<E>

implements List<E>, Deque<E>, Cloneable, java.io.Serializable

// Deque是双端队列的意思，这个接口继承自Queue接口。

当然这么多年了，我也只用过链表。。。

链表都熟，不谈了。

堆栈，是一种先进后出，后进先出的数据结构，LinkedList里相应的方法为：

add（），压入；peek（），提供栈顶元素；pop（），弹出栈顶元素（提供并删除）。

队列，是一种先进先出，后进后出的数据结构，LinkedList里相应的方法为：

offer（），将元素放入队尾；peek（）/element（），返回队首元素；poll（）/remove（）方法移除并返回队首元素。

java有队列接口，Queue，所以可以这样new对象

Queue queue = new LinkedList();

## 迭代器

Iterator就不讲了。

ListIterator可以双向移动。看上去蛮吊的，不过我仔细思考了一下，就我接触到的业务场景中，不存在双向移动这种用法，以后也不可能，这个双向移动给我的感觉很鸡肋。

移动主要靠这四个方法：

boolean hasNext();

E next();

boolean hasPrevious();

E previous();

不过藉由双向移动，倒是让我想起一个算法，双指针扫描。

这个算法的意义在于破解嵌套循环，把O（n2）降为O（n），这个算法只能应用与数组。

大义是，定义两个起始位，一般是头和尾，头向后移动，尾向前移动。脑子快的肯定立马反应过来了，那么是先从头开始向后扫，还是先从尾开始向前扫？

答案是看情况，而我这种二货当时想的是头和尾一起扫。。。。

不管是先从头开始，还是先从后开始，中间肯定涉及到转换。就假设先从头开始向后扫，一边扫，一遍执行某种判断，发现当前元素符合判断，那么立马暂停，开始从尾向前扫。

我还是得给你们找个题目。

LeetCode.141. Linked List Cycle

快速失败(fail-fast)与安全失败(fail-safe)

问题来了，一个ArrayList，线程A在迭代它时，线程B对它进行了结构上的修改，比如加元素，删元素，这时会发生什么？

分两种情况。

如果用的是普通的集合对象，就是java.util包里的，那么一边迭代，一边还想改，程序会直接抛出异常，ConcurrentModificationException。

如果用的是能并发的集合对象，java.util.concurrent包底下的玩意。一边迭代，一边改，什么事都不会发生。因为，迭代这种list时，其实不是迭代list本身，而是在迭代前，先做一次自我复制，然后迭代这个副本。而修改，还是改原来的对象。所以迭代与修改不冲突，但由于迭代的是副本，因此修改后的结果，是不可能被迭代出来的。

第一种叫快速失败，第二种叫安全失败。

## Set

集合的存在感我觉得挺低的。主要是因为HashSet的唯一性是靠HashMap实现的。所谓的值唯一，其实是利用了Map的键不得重复这一特性。

看了一下Set接口的源码我觉得他存在感更低了，因为这家伙的方法与Collection一模一样（部分java 8方法不考虑），没有任何额外的功能。

这东西，等以后再说吧。

## HashMap

Map，是一个以键值对方式存储数据的容器。

取数据时会调用key的hashcode()方法，若在map中找到了相应的key，在调用equals()方法进行比较。相同，则返回数据；不同，返回null。

存数据时也是这个道理，调用hashcode()以及equals()。存在该key就替换为新值，不存在直接存入map。散列还涉及到碰撞等问题，虽说是底层，但这个知识点很重要。

***静态常量***

**static** **final** **int** ***DEFAULT\_INITIAL\_CAPACITY*** = 1 << 4;

**static** **final** **int** ***MAXIMUM\_CAPACITY*** = 1 << 30;

**static** **final** **float** ***DEFAULT\_LOAD\_FACTOR*** = 0.75f;

**static** **final** **int** ***TREEIFY\_THRESHOLD*** = 8;

**static** **final** **int** ***UNTREEIFY\_THRESHOLD*** = 6;

**static** **final** **int** ***MIN\_TREEIFY\_CAPACITY*** = 64;

// 这种东西都是二话不说直接上源码。

默认初始容量，16；最大容量，2^30；负载因子，0.75

顺带一提ArrayList的默认初始容量为10，当元素个数超过10时会扩容，扩容规则是乘1.5倍再加1。

负载因子决定了HashMap在自动扩容前的装填程度，0.75意味着填入12个键值对后，HashMap就要进行扩容。负载因子越大，装填程度越高，但相应的查找效率低，反之亦然。

当一个桶上的节点个数大于8时链表转树，小于6时树转链表。64是树的最低容量。

***遍历HashMap***

1. 内部类Map.Entry<K, V>

HashMap有个内部类，会保存键值对，只是这个造型总让我误以为这内部类也是个map，但人家只是个泛型啊。（我泛型学的真差。。。。）HashMap还有个方法：

public Set<Map.Entry<K, V>> entrySet()

那么很自然而然的：

foreach(Map.Entry<> entry：map.entrySet()){

System.out.println(entry.getKey()+””+entry.getValue());

}

1. keySet() & values()

上面那个方法是获取完整的键值对，这里就是只获取键或者只获取值。

public Set<K> keySet()

public Collection<V> values()

// 我觉得接下来的这两个方法都很无聊

3. Iterator + entrySet()

4. Iterator + keySet()

## 月经问题

// 俗称每月经常问的问题

### ArrayList与LinkedList的比较

# 异常

有些普普通通的东西，因为名字起得屌，所以时间久了它就不普通了，其中之一就是异常。

我说它普通，是因为它只是一个类而已。

public class Exception extends Throwable

public class Throwable implements Serializable

身为一个类，自然是可以被new的，new完之后自然也可以调用这个类的方法

public static void main(String[] args){

Exception ex = new Exception();

ex.printStackTrace();

}

异常做为一个类，和其他类没什么区别。它开始变得陌生，是从throw这个关键字开始的

public static void main(String[] args) {

Exception ex = new Exception();

throw ex;

}

throw 关键字后只能跟异常对象，说的专业一点就是只能跟Throwable及其子类对象。throw其他类对象，程序是没法被编译的。

那throw + 异常，这个组合能干嘛呢？

这个组合的目的是终止程序运行 ：）

我这么讲肯定是夸张了，但如果一个异常对象被throw，而你又没把它try住，那么程序会立马终止运行。

现在知道了这个组合的作用，那为什么要赋予这个组合这种作用呢？

为了告诉写代码的人，你写错了。

我们写程序难免会犯错误，比如忘记给某个引用new对象，或者某数组长度为5，而我们却企图引用第6个元素。这种情况，就好比你问我冯晓庭几把大不大，我说无可奉告，因为你不能让我评价一个没有的东西。程序也一样，面对这种本就不存在的玩意，你想让计算机这种没脑子的东西做什么？它肯定直接就挂了。

ok，程序挂了，现在问你，程序为什么挂了？

在有提示的情况下，我们自然能回答，因为某个引用我忘记给它new对象了。那什么都没有呢，怎么找？

so，这种明摆着程序无法执行的情况，不如直接主动用throw + 异常组合终止程序运行，然后抛出一堆提示信息，告诉我们，哪里，第几行，犯了什么错。快去改吧，龟儿子。

因此，像空指针异常会被抛出，是因为编译器在用对象前，帮我们做了判断。它在编译程序的时候，偷偷的给我们在某些我们看不见的地方，加了下面这种代码。

if(t == null){

throw new NullPointerException();

}

你会发现异常是一种很自然的思路，当一门语言诞生时，与之相配套的异常机制，也同时诞生了。

## 是否需要try-catch

try-catch是java的一个语法，关键字try后跟一对花括号，然后在紧跟一个catch再紧跟一对括号，一对花括号。

try {

} catch (Exception e) {

// TODO: handle exception

}

现在来讲作用，昨天说过了，一个异常被throw后，程序会直接挂掉。但只要写了try-catch，程序就可以继续跑下去。

再说的详细点，一个异常被throw后，周围所有的代码都会停止运行。但只要写了try-catch，那么只会停止运行try块内的代码，而try块外的代码不受影响。

抛出异常后，try块内的代码不管执行到哪一步，都立马放弃，进入catch块，并执行catch块里的代码。catch块里的代码执行完后，就继续执行catch块底下的代码。

所以也可以这么说，try-catch保护了它周围的代码，生化隔离。

底下两段代码执行一下就可以看出区别来了。

public static void main(String[] args){

while (true) {

System.out.println("给我五百万");

throw new RuntimeException();

}

}

public static void main(String[] args) {

while (true) {

System.out.println("给我五百万");

try {

throw new RuntimeException();

} catch (Exception e) {

}

}

}

下面来讲今天的问题，要不要写try-catch。唔，我这点知识水平其实提不出这么有深度的问题，这是从一篇文章上看过来的。try-catch这个东西实在太好用了，加了这个就可以保护其它代码，让整个流程继续跑下去。

呐，我不知道你有没有遇到过这种体验，对着流程找bug，看来看去代码没问题，但数据就是错的，然后翻日志里头毛都没有，没办法，只能在认为可能出错的环节又补一堆日志，重新发布测试。

所以这就引发了一个问题，大部分人catch块里是没有什么代码的，撑死了打条日志。这样一来出了问题会相当难排查，尤其是一些藏得很深的bug，冷不丁冒出来。打了日志还好，痛苦的是没打日志，更痛苦的是打了日志，但因为时间过久，日志文件被服务器删了。

如果最开始没加try-catch，那么只要报错，程序终止运行后，你立马就知道哪边有问题，分分钟解决掉。顺带一提，这观点是微软某某吊到天上去的大佬，有次开讲座提出的。

那么结论是，我不知道。

开玩笑，不加一个试试，最后被喷的又不是大佬。加了。。。有可能是自己给自己挖坑。

有些问题，或者大部分问题，穷究到最后，随便一句话都能当答案，越是这样就越难给出结论。

向左边扔的纸飞机看不见，朝右边放的风筝又摸不着线，猜来猜去，这样也行，那样也行，到最后就一个人傻瓜样的盯着空白的电脑屏幕瞎想。想着想着连最开始的问题都丢掉了，但还是继续想，因为这样脑子就不会停下来，不会停下来就不用去抉择。

活着真累啊。

## throws

跟在方法后面，用来表示这个方法可能抛出什么样的异常。还有一个好处，就是写了throws以后，就不会被强制处理方法内的Exception了。我一般用它来偷懒，毕竟写try-catch太烦了。

现在想想这是一个很差的习惯，有些异常该方法内清理就方法内清理，不要影响其他方法。

由于写了throws就不用再写try-catch，所以从我个人角度，我觉得throws的实现原理可能是这样的：

void g() throws Exception {

}

void g() {

try {

} catch (Exception e) {

throw e;

}

}

这两段代码是等价的。

## finally

跟在try或catch后面的代码块。

一个try后面可以跟多个catch，但只能跟一个finally。

finally的特性在于绝对会被执行到。即便存在return语句，finally里的东西还是会被执行到。

try {

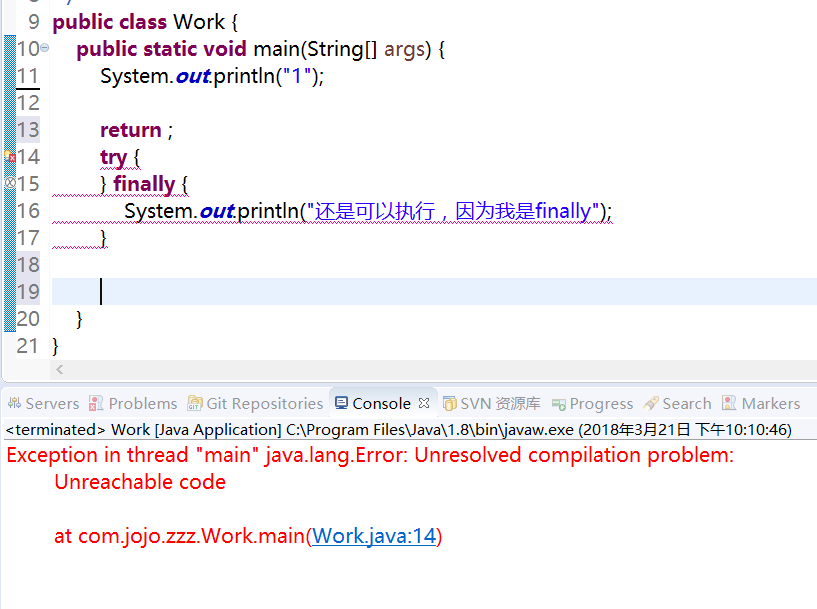
return ;

} finally {

System.out.println("还是可以执行，因为我是finally");

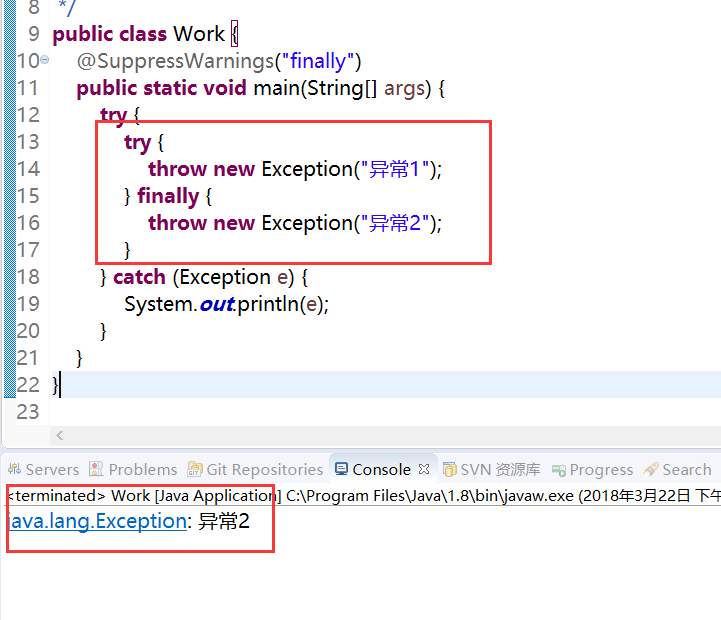
}

然后我想了想，搞不好有bug，因为return语句是在try块里面的。那移出去呢？



移出去就不准你编译，设计语言的那帮人还真是猴精猴精的。

## 异常丢失



如果finally块中的代码会抛异常，最好自己把异常给处理了，不然会覆盖try块中抛出的异常。从业务角度讲，肯定是try块里的代码重要，finally基本上就用来清理一些资源，比如调各种stream的close()方法。

因此如果一个不小心finally块中的异常覆盖了try块中的异常，导致问题不好排查，就本末倒置了。

## 异常的限制

下面这个例子非常完美。

class A{

void f() throws IOException{}

}

public class Work extends A {

void f() {}

public static void main(String[] args){

A a = new Work();

try {

a.f();

} catch (IOException e) {

e.printStackTrace();

}

}

}

如果父类或接口中的方法声明了方法要抛出的异常类型，那么，子类可以不抛异常，但是要抛也只能抛父类声明好的异常类型。

如果子类的方法不抛异常，但是new子类对象时，却用父类引用去接，那么就算子类方法不抛异常，编译器还是会强制你try住。

## 猜想

// 我不高兴做验证了

一个代码块，一次只能抛一个异常，不存在同时抛两个异常的情况。但是可以在抛出这个异常后，被上一个try捕捉后再抛一个异常，然后被上上个异常捕捉，继续抛异常。

像个链条一样。

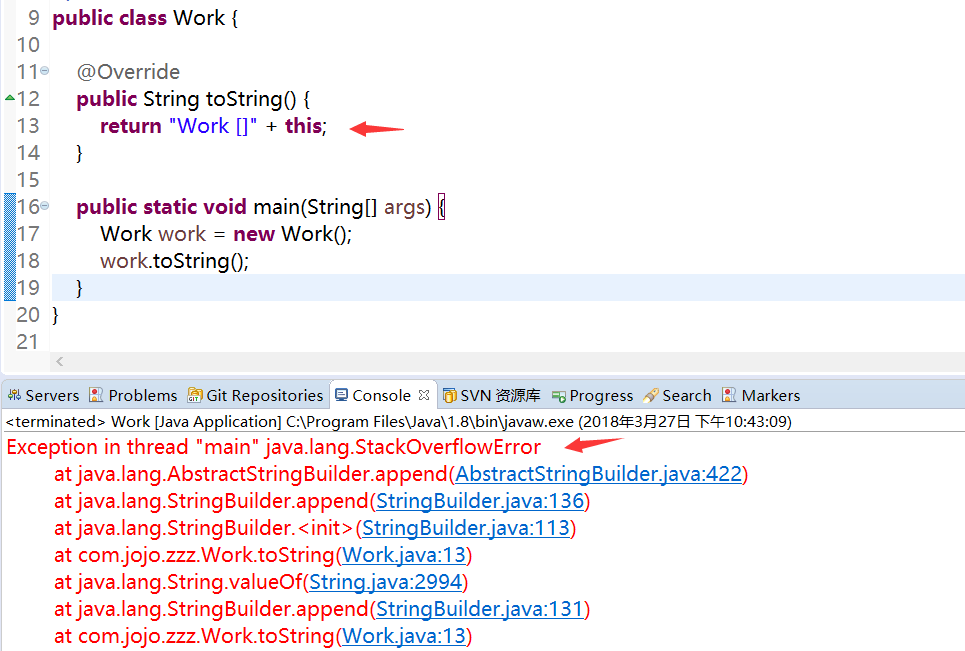
## 异常总结

遇到异常，能抛就抛，必须要用try-catch处理的情况，也请一定至少确保，catch块内绝对能把异常信息以及堆栈信息打印出来。

**///////////////////////////////////////////////////////////////////////////////////**

# 字符串

## 无意识的递归



永远不要在某个类的toString方法里，将this 与 + 操作符连在一起。

当编译器看见一个String对象后面跟着一个“+”，而后面的对象却不是String类型的，那么，编译器会试着将不是String类型的对象，转成String。在上面那个例子里就是把this转换成String。

怎么转换呢，通过调用对象的toString方法。

在上面那个例子里，就出现了不停的自己调自己的情况，也就是递归。

不要这么写，非得这么写，解决办法也有，把this改成super.toString()。但还是不要这么写。

## 格式化输出

这5个字看的我很摸不着头脑。

反正，我见到的，就是在说两个功能。第一，用占位符代替 + 操作符；第二，将某一类型的值用另一种类型输出，比如，char类型以int型打印到控制台。

第一点，用占位符代替 + 操作符。

大概这么个意思

System.out.println("输出一个整数：" + 11);

System.out.format("输出一个整数：%d", 11);

事先声明，没有优劣之分，这种东西全靠个人喜好，我是不怎么喜欢这种占位符的。你看，输出一个整数，因此要用%d占位符，如果输出其它类型参数，就要换另一种类型的占位符，写错了还会抛异常。另一点，+ 符，虽然通过shift建才能显示出来，但%也要通过shift建，完了还要在按一个d键，从操作上讲多了一步。

讨厌！

题外话，虽然上面那个占位符我很讨厌，不过logback的我却很喜欢。

logger.error("输出日志，一个对象：" + new Object() + " 两个对象：" + new Object());

logger.error("输出日志，一个对象：{} 两个对象：", new Object(), new Object());

打日志比较麻烦的地方，在于要拼接的对象往往不止一个，写 + 号是不累，烦的是“”引号，拼来拼去，写的不顺畅。

这时候用logback的 {} 占位符就和爽利，你看，我只需要一对引号就好了，而且，虽然 {} 大括号也要用shift键才能按出来，但是 {} 与字母们处于同一区域，离手不远，按起来不费劲。相比较于写 + 与 “”这个占位符真的好。

第二点：

char ch = 'u';

System.out.println(String.format("%s", ch));// 字符串格式输出

System.out.println(String.format("%b", ch));// boolean类型输出

System.out.println(String.format("%h", ch));// ?

你相信我，在这个年代这东西没吊用。

除了在阿宅面前炫技

## 正则表达式

这个东西的语法非常简单，一看就懂，难得是考验人的想象力。一门组合的艺术。

<a\\s(href=\"([\\w:\\|%\\./\\-=\u4e00-\u9fa5]\*)\")[\\s\\w\"=]\*>\u8fc5\u96f7\u002d\u7535\u9a74</a>

上面那个正则是用来获取网页内的磁力链的，我写的长的要死，回过头我自己都看不懂。。。。

想来我也没资格说这东西怎么用，那些操作符我也不想背，还是讲讲遇到过的坑吧。

### 部分匹配与完全匹配

在Java里使用正则表达式一般看 Pattern 与 Matcher 类。但这两个东西写起来太长，总是被我习惯性忽略。在这之前我一直用String类的matches方法。

现在提问，"abc".matches("a") 返回true还是false？

今天我看到这行代码还是会情不自禁想说true，因为字符“a”存在于“abc”内，所以能被匹配到，应该返回true。

可错了啊，matches方法会将正则表达式与整个字符串做比较，整个符合才会返回true。正则表达式a，只能匹配字符a，不能匹配字符串abc。"abc".matches("abc") 这样就返回true了。

在一个长的要死的字符串中找相应的内容，只能靠Matcher的find方法。

Pattern p = Pattern.compile("a");

Matcher m = p.matcher("aaaaab");

while (m.find()) {

System.out.println(m.group());

}

目前来说我就是自己玩，都不怎么用正则了，那会还想着用它爬爬磁力链，后来发现Jsoup可以直接提取，selenium更是能动态加载。屌的一批。要说什么地方还能用到它的话，估计就是处理没什么规则，中英文各种特殊符号还乱飞的文本了。

## Scanner

扫描器。

刚学java时接触的类，只知它能让我从控制台输入参数，因此觉得没什么卵用。今天看了下它的源码，发现这货还可以用来扫描文件，输入流，字符串等，也就某种程度上简化正则表达式的代码。

所以还是觉得没什么卵用。

主要我现在commons和guava用的比较多，就算真写正则，我也这么搞。

String filepath = "D:\\Workspace\\test\\qwe.txt";

BufferedReader reader = Files.newReader(new File(filepath), Charset.forName("UTF-8"));

String string = IOUtils.toString(reader);

String regex = "正则表达式";

Pattern pattern = Pattern.compile(regex);

Matcher matcher = pattern.matcher(string);

while (matcher.find()) {

System.out.println(matcher.group());

}

## String类常用方法

// 根类Object的方法就不算了。

String是不可变对象，也就是说，无论你做什么操作该字符串对象都是不会变的。看到着你一定觉得扯淡，因为如果对象不可变，那我截取或是拼接后得到的字符串又是哪来的？

全是new出来的，你对字符串A做的任何更改都会被new成字符串B返回给你，感觉没变化是因为引用没变。

String str = “shabi”;

str = str.subString(0，3);

看上去像是从字符串“shabi”中截取字符串“sha”,其实是new了一个新字符串然后把“sha”赋进去。此刻引用str指向新字符串“sha”，丢失了引用的“shabi”只能静静等待垃圾回收。

subString源码：

**return** (beginIndex == 0) ? **this** : **new** String(value, beginIndex, subLen);

**///////////////////////////////////////////////////////////////////////////////////**

# 类型信息

多态就是靠这个实现的，但对于我来说，还不需要懂那么多，能理清Class还有相关的那几个类（反射）就差不多了。

Class类：

public final class Class<T> implements java.io.Serializable,

GenericDeclaration,

Type,

AnnotatedElement

二话不说看源码是最爽的。

泛型先不看，Class也是一个普普通通的类，就是名字奇怪了点，和关键字class一模一样。

通过Class可以在运行时获得类的相关信息，比如这个类内有哪些字段，有哪些方法，类的名字，是否接口等等。

我他妈当时就死在运行时这3个字上了，第一我不懂什么叫运行时，第二我无法理解，明明我自己定义的类，字段方法我都清楚，干嘛还要靠这个Class来获取相关信息。

今天我依旧似懂非懂，但亲身用过几回后，至少生理上没那么反感了。

怎么说呢，写代码时经常遇到这种需求，把类A的对象转换为类B的对象。

class A {

int i;

int j;

int k;

}

class B {

int i;

int m;

int n;

}

像这么两个类，那么对象new完后，我得不停的调b的set方法，a的get方法。特别某些时候经常发现两个类字段名一样，然后哼哧哼哧写一遍，纯体力活。例子里，字段少，真正项目中，至少15个。

Field[] fields = B.class.getDeclaredFields();

List<Field> list = Arrays.asList(fields);

Collections.sort(list, new Comparator<Field>() {

@Override

public int compare(Field o1, Field o2) {

return o1.getName().compareTo(o2.getName());

}

});

for (Field field : fields) {

StringBuffer sb = new StringBuffer();

sb.append("b.set");

String temp = field.getName().substring(0, 1).toUpperCase() + field.getName().substring(1);

sb.append(temp);

sb.append("(").append("a.get" + temp + "()").append(");");

System.out.println(sb.toString());

}

而通过这段代码，首先我省的写b的set方法了，其次，我还解决了一部分类A与类B字段重名的情况。剩下唯一要做的就是改改a的get方法。

那这么一来是不是就觉得反射挺屌的了，其实很多代码生成器，比如通过某实体类生成对应的service，controller，以及，jsp，就是通过反射实现的。

上面那个例子的作用虽然只是偷懒，但至少说明了一个观点：我们有时候确实不清楚，自己定义好的类及其中的字段与方法，会在什么时候被什么样的方式调用。

## 省代码的案例

与普普通通写代码比，反射的效率肯定低，不过还是那句话，这年头内存与cpu的性价比都爆棚。性能的瓶颈不在反射这，这东西就算要优化也留到最后。

现在假设有一个Book接口，它有很多实现，比如ChineseBook，，JapaneseBook，FrenchBook…… ……

然后有一个List<Book>对象，里面塞了很多Book对象，各种子类实现版本都有。

需求来了，我想要一个功能，当我说出ChineseBook时，立马告诉我List<Book>对象内有几个ChineseBook对象，说出EnglishBook立马告诉我List<Book>对象内有几个EnglishBook对象，依次类推。

如何实现？

// 划水都划的这么优雅 : )

代码我就直接贴出来了，第一个方法时普通实现，第二个方法时反射实现。这里假设list是定长的。

**public** **class** Work {

**private** **static** List<Book> *bookList*; // 你可以假设这个list里有成百上千的对象

**public** **static** **int** getBookCountByType(String typeName) {

**int** result = 0;

**switch** (typeName) {

**case** "ChineseBook":

**for** (Book book : *bookList*) {

**if** (book **instanceof** ChineseBook) {

result++;

}

}

**break**;

**case** "EnglishBook":

**for** (Book book : *bookList*) {

**if** (book **instanceof** EnglishBook) {

result++;

}

}

**break**;

**case** "JapaneseBook":

**for** (Book book : *bookList*) {

**if** (book **instanceof** JapaneseBook) {

result++;

}

}

**break**;

// 要是有其他类型，那就继续写

**default**:

**break**;

}

**return** result;

}

**public** **static** **int** getBookCountByType(Class<? **extends** Book> clazz) {

**int** result = 0;

**for** (Book book : *bookList*) {

**if** (clazz.isInstance(book)) {

result++;

}

}

**return** result;

}

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

*bookList* = Lists.*newArrayList*(**new** ChineseBook());

*bookList*.add(**new** JapaneseBook());

*bookList*.add(**new** EnglishBook());

System.***out***.println(*getBookCountByType*(ChineseBook.**class**));

System.***out***.println(*getBookCountByType*("ChineseBook"));

}

}

当然普通实现我写的也不算好，不过，不用反射的话，无论如何都绕不开不停写判断逻辑的代码，是个单纯的体力活。

然后就是两个方法接受的参数不一样的问题，你可能会觉得奇怪，毕竟一个参数是字符串，一个参数是Class对象，这样看上去有点不公平。

但，从需求上讲，既然我明摆着说了，需要知道某某类型的图书有多少本，那么其他人在调用我写的方法时，肯定是清楚这个图书类型的。所以，传字符串与传Class对象，本质上是一样的。

第二个方法不算大括号一共5行代码，这一切是因为isInstance方法，动态的执行判断。反射里头与之类似的方法还有不少，还是很屌的。

## 反射相关的类

Class类，通过这个类可以获取类类型的类名、修饰符、包信息、父类、实现的接口、构造器、方法、变量、注解

Constructor类，构造器，我习惯说构造函数，作用是new对象，不然还想怎的。

Field类，获取名称、类型、修改值。

Method类，获取名称，接受的参数以及返回的参数，而且能够调用方法。

java.lang.reflect.Array类，我对他一无所知。。。。

## 突破权限控制

这个特性，给我一种莫名的喜剧片既视感

“哦，我的上帝啊，我们造了一扇好棒的门”

“亲爱的乔治，它棒到我们自己都打不开呀”

…… ……

“看样子是没办法了，我的好朋友，再造把锤子吧”

private，default，protected，public，java的四大访问权限，现在被反射无视了。

class A {

private int i = 0;

public int getI() {

return i;

}

private void g() {

System.out.println("私有的");

}

}

public class Work {

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

A a = new A();

Class<A> aClass = A.class;

Field[] field = aClass.getDeclaredFields();

Field iField = field[0];

iField.setAccessible(true);

iField.set(a, 2);

System.out.println(a.getI());

Method[] method = aClass.getDeclaredMethods();

Method gMethod = method[1];

gMethod.setAccessible(true);

gMethod.invoke(a, null);

}

}

记住它，setAccessible。

总结起来就是那句老话，在编译期进行严格的权限控制，在运行期放开。

## 动态代理

**///////////////////////////////////////////////////////////////////////////////////**

# 泛型

## 为什么要泛型

如果一个方法的参数是String类型，那么这个方法便只接受字符串做参数，传StringBuffer或StringBuilder都是会报错的。但将方法的参数定义成CharSequence类型，那么即可以传String，又可以传StringBuffer还可以传StringBuilder。

概括起来，将方法的参数类型定义为基类，那么任意导出类都可以作为参数传递。这就是多态的好处，它可以让代码具有通用性。

再看上面那个例子，虽然定义成CharSequence可以随便传任意类型的字符串，但是依旧不能传int，不能传float，不能传double，不能传long。你先别急着思考为什么要传这么多类型的参数，反正会有这种日了狗的需求的。

因此，将参数定义成基类依旧是有限制性的，只是限制范围稍微宽了点。

这时候泛型就登场了，泛型的意思就是任意类型 Any ,有了它你真的是想怎么传就怎么传啊。

当然你要是像我一样钻牛角尖，你刚接触泛型多半也会陷进这个问题里。

Java不是有Object吗？

这个所有类的父类，我总觉得有它在就没泛型什么事。所以说还是当时动手能力弱啊，只会想不晓得敲两行暖暖手。像List这种东西，要是用Object做存入类型，到取数据的时候，写强转的括号能把人写疯。

我想可能也正是因为这个问题，导致java在1.5版本里才加入了泛型这个特性。

语言会进化，“省代码”这个三个字真是起了不可磨灭的推动作用。代码这东西是从“0,1”演变成汇编，演变成C，再演变成python这种看上去几乎就是英语的玩意。  
 不知道以后会变成什么样子。

## 泛型的第一种用法

Class Tuple<A, B>{

public final A a;

public final B b;

public Tuple(A a, B b) {

this.a = a;

this.b = b;

}

}

这样就创建了一个可以持有任意两种类型对象的类。

其实这东西我也解释不出太多东西，就直接背好了。泛型就是尖括号，然后大写英文字母代表某种类型。

**///////////////////////////////////////////////////////////////////////////////////**

# Comparable & Comparator

俩接口，作用都是排序java.lang.Comparable与java.util.Comparator，而且均为泛型。

Comparable接口只有一个方法 compareTo(T t) ，返回值为int。实现了comparable接口的类可以直接使用Collections.sort()与Arrays.sort()方法。

Comparator接口主要有两个方法，compare（T o1，T o2）与equals()。equals()方法可以不重载，它的作用就是保证两个comparator相等。将比较器作为Collections.sort()及Arrays.sort()的参数并调用。

直接调用内部方法与间接调用外部方法的区别。

comparator的升序与降序。

o1 < o2

如果return 1；那么这是升序。

如果return -1；那么这是降序。

**///////////////////////////////////////////////////////////////////////////////////**

# 接口与抽象类的异同点

相同点：

可以被继承，但都不能被实例化。

不同点：

从变量上讲，接口中一般不包含成员变量，如果有必须是static final 类型的；抽象类没有这种限制。

从方法上讲，接口中的方法自动是public abstract类型的；抽象类的可以存在普通方法访问类型也可以自己设置。

从继承上讲，接口可以实现多个，抽象类只能继承一个。

**///////////////////////////////////////////////////////////////////////////////////**

# Object

protected Object clone()

// protected我不是很懂，因为从访问权限上讲，任何类都自动继承Object，因此该方法也是可以随意访问的。

使用前有一点要注意，调用该方法的对象的类必须实现Cloneable接口，不然会抛出CloneNotSupportedException。

这个方法让我想起了C++的浅拷贝与深拷贝。类内可能还包含其他类的引用，直接调用clone()方法只会复制引用，而引用背后的对象是不会复制，这种浅拷贝存在很大的不稳定因素，比如修改。

要适当重载，虽说这个方法我根本就没用过。。。。

public boolean equals(Object obj)

我在这个方法上栽过跟头，不止一次。比较字符串之间是否相等就要用equals()不能用==，这个操作符只比较引用。另外默认的equals()方法只比较内存地址，String类的equals()方法是重载过的，它会完整比较两个字符串的字符。

protected void finalize()

并非释放内存，详情见垃圾回收。

////////////////////////////////////////////////////////////////////////////////////////////

***///////////////////////////////////////////////////////////////////////////////////***

# 操蛋的时间

yyyy-MM-dd HH:mm:ss   
年-月-日 时:分:秒  
大写是为了区分“**月**”与“**分**”

另外小写的h是12小时制，大写的H是24小时制。

书写格式和语言规定有关，上述写法是Windows系统中的我们常见的写法，包括日期设置于办公软件在内。在其他语言中有类似的但使用符号或格式不同的写法。

有的时候我们会看到这样的格式：yyyy-M-d H:m:s  
mm与m等，它们的区别为**是否有前导零**：H,m,s表示**非零开始**，HH,mm,ss表示**从零开始**。  
比如凌晨1点2分，HH:mm显示为**01:02**，H:m显示为**1:2**。

以2014年1月1日凌晨1点1分1秒（当天是星期三）为例子介绍一下其他的：  
**yyyy/yyy/yy/y** 显示为 **2014/2014/14/4**  
（3个y与4个y是一样的，为了便于理解多写成4个y）

**MMMM/MMM/MM/M** 显示为 **一月/一月/01/1**  
（4个M显示全称，3个M显示缩写，不过中文显示是一样的，英文就是January和Jan）

**dddd/ddd/dd/d** 显示为 **星期三/周三**(有的语言显示为“三”)**/01/1**  
（在英文中同M一样，4个d是全称，3个是简称；  
dddd/ddd表示星期几，dd/d表示几号）

**HH/H/hh/h** 显示为 **01/1/01 AM/1 AM**

剩下的**mm/m/ss/s**只是前导零的问题了。

yyyy/M/d/dddd H:mm:ss 就是 2014年1月1日星期三 1:01:01

# ASCII码

再来分享一个工作感悟，还是关于调试sip2指令的。

1. 指令的完结是由末尾的换行符决定的，比如这条登录指令：

93NNCNXXX|COXXX|CPXXX

这样对方服务器接收到后是不会有反应的，因为它找不到换行符，得改成这样

93NNCNXXX|COXXX|CPXXX\r\n

但比较麻烦的是，除\r\n外，\r、\n、\0这三个字符也能表示换行，即指令的末尾。

因此

93NNCNXXX|COXXX|CPXXX\r

93NNCNXXX|COXXX|CPXXX\n

93NNCNXXX|COXXX|CPXXX\0

都是可以的，至于到底选择哪一种，看厂商。

于是带来了第一个问题。厂商的文档里不会告诉你请求后带那种格式的换行符，我碰到的每一家厂商都是，这帮竞争者们在这件事上出乎其料的保持一致。

另外，不光是请求过去的指令需要换行符，返回过来的指令也有换行符，因此可以搓出16中组合方式。

理论上讲，本地写个main方法，用mina一个个试很快也就过去了。可他妈的，这次对接的厂商没有将服务器映射到外网，我本地试个奶子。

没办法，提交，发布，用正式环境去试，在不停的看日志。

要命。。。。

尤其是忙了一天，脑袋晕乎乎的，上上次试了哪种组合，一不小心就忘了。

再试出了请求指令的末尾换行符，准备继续试返回指令的末尾换行符时，突然间一道灵光击中大脑。

把服务器返回的指令用解析成ascii码，然后整个输出出来，看末尾到底是什么不完了！

for(char c : sb.toString().toCharArray()) {

System.out.println(Integer.valueOf(c));

}

# ecplise插件

http://marketplace.eclipse.org/content/mybatipse