类：

还是使用class关键字

构造函数：

在Kotlin中的一个类可以有一个主构造函数和一个或多个此构造函数。

主构造函数是类头的一部分：它跟在类名（和可选的类型参数）后：

**class** Person **constructor**(firstName: String) {

}

如果主构造函数没有任何注解或者可见性修饰符，可以省略constructor关键字

主构造函数不能包含任何的代码。初始化的代码可以放到以init关键字作为前缀的初始化块中：

class InitOrderDemo(name: String) {

val firstProperty = "First property: $name".also(::println)

init {

println("First initializer block that prints ${name}")

}

val secondProperty = "Second property: ${name.length}".also(::println)

init {

println("Second initializer block that prints ${name.length}")

}

}

在实例初始化期间，初始化块按照它们出现在类体中的顺序执行，与属性初始化器交织在一起

注意，主构造的参数可以在初始化块中使用。它们也可以在类体内声明的属性初始化器中使用：

**class** Customer(name: String) {

**val** customerKey = name.toUpperCase()

}

事实上，声明属性以及从主构造函数初始化属性，Kotlin 有简洁的语法：

**class** Person(**val** firstName: String, **val** lastName: String, **var** age: Int) {

// ……

}

与普通属性一样，主构造函数中声明的属性可以是可变的（var）或只读的（val）。

如果构造函数有注解或可见性修饰符，这个constructor关键字是必需的，并且在这些修饰符在它前面：

**class** Customer **public** @Inject **constructor**(name: String) { …… }

次构造器：

前缀有constructor的次构造器：

**class** Person {

**constructor**(parent: Person) {

parent.children.add(**this**)

}

}

如果类有一个主构造函数，每个次构造函数需要委托给主构造函数，可以直接委托或者通过别的次构造函数间接委托。委托到同一个类的另一个构造函数用this关键字即可

**class** Person {

**constructor**(parent: Person) {

parent.children.add(**this**)

}

}

注意：

初始化块中的代码实际上会成为主构造函数的一部分。委托给主构造函数会作为次构造函数的第一条语句，因此所有初始化块中的代码都会在此构造函数体之前执行。即使该类没有主构造函数，这种委托仍会隐式发生，并且仍会执行初始化块：

class Constructors {

init {

println("Init block")

}

constructor(i: Int) {

println("Constructor")

}

}

如果一个非抽象类没有声明任何（主或次）构造函数，它会有一个生成的不带参数的主构造器函数。构造函数的可见性是public。如果不希望有一个公有的构造函数，则需要声明一个带有非默认可见性的空的主构造函数：

**class** DontCreateMe **private** **constructor** () {

}

****注意****：在 JVM 上，如果主构造函数的所有的参数都有默认值，编译器会生成 一个额外的无参构造函数，它将使用默认值。这使得 Kotlin 更易于使用像 Jackson 或者 JPA 这样的通过无参构造函数创建类的实例的库。

**class** Customer(**val** customerName: String = "")

创建类的实例：

**val** invoice = Invoice()

**val** customer = Customer("Joe Smith")

注意：Kotlin并没有new关键字。

继承：

在Kotlin中所有类都有一个共同的超类Any

Any不是java.lang.Object，它除了equals（），hasCode（）和toString（）外没有任何成员

要声明一个显示的超类型，把类型放到类头的冒号之后：

**open** **class** Base(p: Int)

**class** Derived(p: Int) : Base(p)

如果该类有一个主构造函数，其基类可以（并且必须）用基类型的主构造函数参数就地初始化。

如果类没有主构造函数，那么每个次构造函数必须使用super关键字初始化其基类型，或委托给另一个构造函数做到这一地。注意，在这种情况下，不同的次构造函数可以调用基类型额不同的构造函数：

**class** MyView : View {

**constructor**(ctx: Context) : **super**(ctx)

**constructor**(ctx: Context, attrs: AttributeSet) : **super**(ctx, attrs)

}

Open标注与java中的final相反，它允许其他类从这个类继承。默认情况下在Kotlin中所有的类都是final

覆盖方法：

与java不同，Kotlin需要显式标注可覆盖的成员和覆盖后的成员：

**open** **class** Base {

**open** **fun** v() {}

**fun** nv() {}

}**class** Derived() : Base() {

**override** **fun** v() {}

}

如果函数没有标注open，则子类中不允许定义相同签名的函数，不论加不加override。在一个final类中，开放成员是禁止的。

标记为override的成员本身是开放的，也就是说，它可以在子类覆盖。如果想禁止再次覆盖，使用final关键字：

**open** **class** AnotherDerived() : Base() {

**final** **override** **fun** v() {

}

}

覆盖属相：

同样必须以override开头，并且它们必须具有兼容的类型。每个声明的属性可以由具有初始化器的属性或者具有getter方法的属性覆盖

**open** **class** Foo {

**open** **val** x: Int **get**() { …… }

}

**class** Bar1 : Foo() {

**override** **val** x: Int = ……

}

也可以用一个var属性覆盖一个val属性，但反之不可以。因为一个val属性本质上声明了一个getter方法，而将其覆盖为var只是在子类中额外声明一个setter方法。

可以在主构造函数中使用override关键字作为属性声明的一部分：

**interface** Foo {

**val** count: Int

}

**class** Bar1(**override** **val** count: Int) : Foo

**class** Bar2 : Foo {

**override** **var** count: Int = 0

}

调用超类的方法和成员变量，还是用super

在一个内部类中访问外部类的超类，可以通过由外部类名限定的 super 关键字来实现：super@Outer：

**class** Bar : Foo() {

**override** **fun** f() { /\* …… \*/ }

**override** **val** x: Int **get**() = 0

**inner** **class** Baz {

**fun** g() {

**super**@Bar.f() // 调用 Foo 实现的 f()

println(**super**@Bar.x) // 使用 Foo 实现的 x 的 getter

}

}

}

覆盖规则：

在kotlin中，如果一个类从它的直接超类继承相同成员的多个实现，它必须覆盖这个成员并提供自己的实现。为了表示采用从哪个超类型继承的实现，使用由尖括号中超类型名限定的super

**open** **class** A {

**open** **fun** f() {

print("A")

}

**fun** a() {

print("a")

}

}

**interface** B {

**fun** f() {

print("B")

} // 接口成员默认就是“open”的

**fun** b() {

print("b")

}

}

**class** C() : A(), B {

// 编译器要求覆盖 f()：

**override** **fun** f() {

**super**<A>.f() // 调用 A.f()

**super**<B>.f() // 调用 B.f()

}

}

抽象类：

类和其中的某些成员可以声明为abstract。不需要用open标注一个抽象类或者函数。

**open** **class** Base {

**open** **fun** f() {}

}

**abstract** **class** Derived : Base() {

**override** **abstract** **fun** f()

}

## 伴生对象：

在Kotlin中类没有静态方法。

如果在你的类内声明了一个[伴生对象](http://www.kotlincn.net/docs/reference/object-declarations.html" \l "伴生对象)， 你就可以使用像在 Java/C# 中调用静态方法相同的语法来调用其成员，只使用类名作为限定符。

属性和字段：

声明属性：

Kotlin的类可以有属性，属性可以用关键字var声明为可变的，val为只读

编译期常量：

用const修饰符标记为编译期常量

- 位于顶层或者是object的一个成员

- 用String或原生类型值初始化

延迟初始化属性与变量：

有时声明一个非空类型的属性必须在构造函数中初始化。但是有时候我不想这样做，我可能需要在另外一个方法中进行初始化，这个时候就可以用lateinit修饰符标记该属性：

**public** **class** MyTest {

**lateinit** **var** subject: TestSubject

@SetUp **fun** setup() {

subject = TestSubject()

}

@Test **fun** test() {

subject.method() // 直接解引用

}

}

该修饰符只能用在类体中的属性。

### 检测一个 lateinit var 是否已初始化（自 1.2 起）

要检测一个 lateinit var 是否已经初始化过，请在[该属性的引用](http://www.kotlincn.net/docs/reference/reflection.html" \l "属性引用)上使用 .isInitialized：

**if** (foo::bar.isInitialized) {

println(foo.bar)

}

此检测仅对可词法级访问的属性可用，即声明位于同一个类型内、位于其中一个外围类型中或者位于相同文件的顶层的属性。

接口：

还是使用interface来定义接口：

**interface** MyInterface {

**fun** bar()

**fun** foo() {

// 可选的方法体

}

}

## 实现接口

一个类或者对象可以实现一个或多个接口。

**class** Child : MyInterface {

**override** **fun** bar() {

// 方法体

}

}

## 接口中的属性

你可以在接口中定义属性。在接口中声明的属性要么是抽象的，要么提供访问器的实现。在接口中声明的属性不能有幕后字段（backing field），因此接口中声明的访问器不能引用它们。

**interface** MyInterface {

**val** prop: Int // 抽象的

**val** propertyWithImplementation: String

**get**() = "foo"

**fun** foo() {

print(prop)

}

}

**class** Child : MyInterface {

**override** **val** prop: Int = 29

}

解决覆盖冲突，仍然用super<>

可见性修饰符：

类，对象，接口，构造函数，方法，属性和它们的setter都可以有可见性修饰符（getter总是与属性有着相同的可见性）。

在Kotlin中有四个可见性修饰符：private，proteceted，internal，public

如果没有显示指定修饰符得出话，默认可见性是public

- 如果不指定任何可见性修饰符，默认为public，这意味着我的声明将随处可见

- 如果声明为private，它只会在声明它的文件内可见

- 如果声明为internal，它会在相同模块内随处可见

- protected 不适用于顶层声明

如果覆盖一个protected成员并且没有显示指定其可见性，该成员还是会是protected可见性

例子:

**open** **class** Outer {

**private** **val** a = 1

**protected** **open** **val** b = 2

**internal** **val** c = 3

**val** d = 4 // 默认 public

**protected** **class** Nested {

**public** **val** e: Int = 5

}

}

**class** Subclass : Outer() {

// a 不可见

// b、c、d 可见

// Nested 和 e 可见

**override** **val** b = 5 // “b”为 protected

}

**class** Unrelated(o: Outer) {

// o.a、o.b 不可见

// o.c 和 o.d 可见（相同模块）

// Outer.Nested 不可见，Nested::e 也不可见

}

模块：

可见性修饰符internal意味着该成员只在相同模块内可见。更具体地说，一个模块是编译在一起的一套Kotlin文件：

- 一个Intellij IDEA模块

- 一个Maven项目

- 一个Gradle源集

- 一个<kotlinc>Ant任务执行所编译的一套文件

扩展：

Kotlin能够扩展一个类的新功能而无需继承该类或使用像装饰者这样的任何类型的设计模式。这通过叫做扩展的特殊声明完成。

Kotlin支持扩展函数和扩展属性

扩展函数：

声明一个扩展函数，需要用一个接受者类型也就是被扩展的类型来作为他的前缀：

MutableList<Int> 添加一个swap 函数：

**fun** MutableList<Int>.swap(index1: Int, index2: Int) {

**val** tmp = **this**[index1] // “this”对应该列表

**this**[index1] = **this**[index2]

**this**[index2] = tmp

}

扩展视静态解析的：

扩展不能真正修改他们所扩展的类。通过定义一个扩展，并没有在一个类中插入新成员，仅仅是可以通过该类的变量用点表达式用调用这个新函数

如果一个类定义一个成员函数和一个扩展函数，而这两个函数又有相同的接受者类型，相同的名字并且都适用给定的参数，这种情况总是取成员函数：

**class** C {

**fun** foo() { println("member") }

}

**fun** C.foo() { println("extension") }

如果我们调用 C 类型 c的 c.foo()，它将输出“member”，而不是“extension”。

当然，扩展函数重载同样名字但不同签名成员函数也完全可以：

**class** C {

**fun** foo() { println("member") }

}

**fun** C.foo(i: Int) { println("extension") }

调用 C().foo(1) 将输出 "extension"。

可空接受者：

可以为可空接受者类型定义扩展，这样的扩展可以在对象变量上调用，即使值为null，并且可以在函数体内部检测this == null，这能让我们再没有检测null的时候调用Kotlin中的toString（）：检测发生在扩展函数内部。

**fun** Any?.toString(): String {

**if** (**this** == **null**) **return** "null" // 空检测之后，“this”会自动转换为非空类型，所以下面的 toString() // 解析为 Any 类的成员函数

**return** toString()

}

扩展的作用域：

大多数的时候在顶层一定扩展，即直接在包里

扩展声明为成员：

在一个类内部可以为另一个类声明扩展。在这样的扩展内部，有多个隐式接受者--其中的对象成员可以无需通过限定符访问。扩展声明所在的类的实例称为分发接受者，扩展方法调用所在的接受者类型的实例称为扩展接受者

**class** D {

**fun** bar() { …… }

}

**class** C {

**fun** baz() { …… }

**fun** D.foo() {

bar() // 调用 D.bar

baz() // 调用 C.baz

}

**fun** caller(d: D) {

d.foo() // 调用扩展函数

}

}

对于分发接收者和扩展接收者的成员名字冲突的情况，扩展接收者优先。要引用分发接收者的成员你可以使用 [限定的 this 语法](http://www.kotlincn.net/docs/reference/this-expressions.html" \l "限定的-this)。

**class** C {

**fun** D.foo() {

toString() // 调用 D.toString()

**this**@C.toString() // 调用 C.toString()

}

关于扩展还有待研究。。。。。。

数据类：

用来保存数据的类

在Kotlin中，数据类被标记为data：

**data** **class** User(**val** name: String, **val** age: Int)

编译器自动从主构造函数中声明所有属性导出以下成员：

- equals（）/hashcode（）

- toString（）格式是“User（name=John，age=42）”

- componentN（）按声明顺序对应所有属性

- copy（）

数据了必须满足一下要求：

- 主构造函数至少有一个参数

- 主构造函数的所有参数需要标记为val或var

- 数据类不能是抽象，开放，密封或者内部的

- （1.1之前）数据类只能实现接口

成员生成遵循关于成员继承：

- 如果在数据类体中有显示实现equals（），hashCode（）或者toString，或者这些函数在父类中有final实现，那么不会生成这些函数，而会使用现有函数

- 如果超类型具有open的componentN（）函数并且返回兼容的类型，那么会为数据类生成相应的函数，并覆盖超类的现实。如果超类型的这些函数由于签名不兼容或者是final而导致违法覆盖，那么会报错

- 从一个已具有copy（.......）函数且签名匹配的类型派生一个数据类在Kotlin1.2中已经弃用，在Kotlin1.3中禁用

- 不允许为componentN（）以及copy函数提供显示实现。

复制：

在很多情况下，需要复制一个对象改变它的一些属性，但是其余部分不变

**fun** copy(name: String = **this**.name, age: Int = **this**.age) = User(name, age)

这让我们可以写：

**val** jack = User(name = "Jack", age = 1)**val** olderJack = jack.copy(age = 2)

为数据类生成额component函数使它们可在解构声明中使用：

**val** jane = User("Jane", 35)

**val** (name, age) = jane

println("$name, $age years of age") // 输出 "Jane, 35 years of age"

密封类：

密封类用来表示受限的类继承结构：当一个值为有限集中的类型，而不能有任何其他类型时。在某种意义上，它们是枚举类的扩展：枚举类型的值集合也是受限的，但每个枚举常量只存在一个实例，而密封类的一个子类可以有包含状态的多个实例。

要声明一个密封类，需要在类名前面添加sealed修饰符。虽然密封类也可以有子类，但是所有子类都必须在密封类自身相同文件中声明。

**sealed** **class** Expr

**data** **class** Const(**val** number: Double) : Expr()

**data** **class** Sum(**val** e1: Expr, **val** e2: Expr) : Expr()

**object** NotANumber : Expr()

一个密封类是自身抽象的，呀不能直接实例化并可以有抽象成员

密封类不允许有非private构造函数（其构造函数默认为private）

请注意、；扩展密封类子类的类（间接继承者）可以放在任何位置，而无需在同一个文件中

使用密封类的关键好处是在使用when表达式的时候，如果能够验证语句覆盖了所有情况，就不需要为该语句再添加一个else子句了

**fun** eval(expr: Expr): Double = **when**(expr) {

**is** Const -> expr.number

**is** Sum -> eval(expr.e1) + eval(expr.e2)

NotANumber -> Double.NaN // 不再需要 `else` 子句，因为我们已经覆盖了所有的情况

}

泛型：

与java类似，Kotlin中的类也可以有类型参数：

**class** Box<T>(t: T) {

**var** value = t

}

一般来说，要创建这样类的实例，我们需要提供类型参数：

**val** box: Box<Int> = Box<Int>(1)

但是如果类型参数可以推断出来，例如从构造函数的参数或者从其他途径，允许省略类型参数：

**val** box = Box(1) // 1 具有类型 Int，所以编译器知道我们说的是 Box<Int>。

型变：

Kotlin中没有java中的通配符类型，它有两个其他的东西：声明处型变与类型投影

首先，java的泛型是不型变的，这意味着List<String>不是List<Object>的子类型。

声明处型变：

标注Source的类型参数T来确保它仅从Source<T>成员中返回，用out修饰符：

**interface** Source<**out** T> {

**fun** nextT(): T

}

**fun** demo(strs: Source<String>) {

**val** objects: Source<Any> = strs // 这个没问题，因为 T 是一个 out-参数 // ……

}

一般原则是：当一个类C的类型参数被声明为out时，它就只能出现在C的生源的输出位置。但回报是C<Base>可以安全地作为C<Derived>的超类

简而言之，类C是在参数T上是协变的，或者说T是一个协变的类型参数。

Out修饰符称为型变注解，并且由于它在类型参数声明处提供，所以叫声明处型变。

in修饰符使得一个类型参数逆变：只可以被消费而不可以被生产。

**interface** Comparable<**in** T> {

**operator** **fun** compareTo(other: T): Int

}

**fun** demo(x: Comparable<Number>) {

x.compareTo(1.0) // 1.0 拥有类型 Double，它是 Number 的子类型 // 因此，我们可以将 x 赋给类型为 Comparable <Double> 的变量

**val** y: Comparable<Double> = x // OK！

}

类型投影：

使用处型变：类型投影

将类型参数T声明为out非常方便，并且能避免使用处子类型化的麻烦，但是有些类实际上不能限制为只返回T

**class** Array<T>(**val** size: Int) {

**fun** **get**(index: Int): T { ///\* …… \*/ }

**fun** **set**(index: Int, value: T) { ///\* …… \*/ }

}

该类在 T 上既不能是协变的也不能是逆变的。这造成了一些不灵活性。考虑下述函数：

那么，我们唯一要确保的是 copy() 不会做任何坏事。我们想阻止它****写****到 from，我们可以：

**fun** copy(from: Array<**out** Any>, to: Array<Any>) {

// ……

}

这里发生的事情称为****类型投影****：我们说from不仅仅是一个数组，而是一个受限制的（****投影的****）数组：我们只可以调用返回类型为类型参数 T 的方法，如上，这意味着我们只能调用 get()。这就是我们的****使用处型变****的用法，并且是对应于 Java 的 Array<? extends Object>、 但使用更简单些的方式。

你也可以使用 ****in**** 投影一个类型：

**fun** fill(dest: Array<**in** String>, value: String) {

// ……

}

Array<in String> 对应于 Java 的 Array<? super String>，也就是说，你可以传递一个 CharSequence 数组或一个 Object 数组给 fill() 函数。

星投影：

- 对于Foo<out T>，其中T是一个具有上界TUpper的协变类型参数，Foo<\*>等价于Foo<out TUpper>。这意味着当T未知的时候，我们可以安全的从Foo<\*>读取TUpper的值

- 对于Foo<in T>其中T是一个逆变类型参数，Foo<\*>等价于Foo<in Nothing>，这意味着当T未知时，没有什么可以以安全的方式写入Foo<\*>

- 对于Foo<T>，其中T是一个具有上界TUpper的不型变类型参数，Foo<\*>对于读取时等价于Foo<out TUpper>而对于写值时等价于Foo<in Nothing>

上界：

最常见的约束类型是与 Java 的 extends 关键字对应的 ****上界****：

**fun** <T : Comparable<T>> sort(list: List<T>) {

// ……

}

冒号之后指定的类型是****上界****：只有 Comparable<T> 的子类型可以替代 T。 例如：

sort(listOf(1, 2, 3)) // OK。Int 是 Comparable<Int> 的子类型sort(listOf(HashMap<Int, String>())) // 错误：HashMap<Int, String> 不是 Comparable<HashMap<Int, String>> 的子类型

默认的上界（如果没有声明）是 Any?。在尖括号中只能指定一个上界。 如果同一类型参数需要多个上界，我们需要一个单独的 ****where****-子句：

**fun** <T> copyWhenGreater(list: List<T>, threshold: T): List<String>

**where** T : CharSequence,

T : Comparable<T> {

**return** list.filter { it > threshold }.map { it.toString() }

}

KOtlin为泛型声明用法执行的类型安全监测仅在编译器进行。运行时泛型类型的实例不保留关于其类型实参的任何信息。其类型信息称为擦除。

嵌套类与内部类：

类可以嵌套在其他类中：

**class** Outer {

**private** **val** bar: Int = 1

**class** Nested {

**fun** foo() = 2

}

}

**val** demo = Outer.Nested().foo() // == 2

类可以标记为inner以便能够访问外部类的成员。内部类会带有一个对外部类的对象的引用：

**class** Outer {

**private** **val** bar: Int = 1

**inner** **class** Inner {

**fun** foo() = bar

}

}

**val** demo = Outer().Inner().foo() // == 1

匿名内部类：

使用对象表达式创建匿名内部类实例：

window.addMouseListener(**object**: MouseAdapter() {

**override** **fun** mouseClicked(e: MouseEvent) {

// ……

}

**override** **fun** mouseEntered(e: MouseEvent) {

// ……

}

})

如果对象是函数式Java接口（即具有单个抽象方法的Java接口）的私立，我们可以使用带接口类型前缀的lambda表达式创建它：

**val** listener = ActionListener { println("clicked") }

对象表达式和对象声明：

有时候，我们需要创建一个对某个类做了轻微改动的类的对象，而不用为之显式声明新的子类。Java用匿名内部类处理这种情况

对象表达式：

要创建一个继承自某个（或某些）类型的匿名类的对象，会这样写：

window.addMouseListener(**object** : MouseAdapter() {

**override** **fun** mouseClicked(e: MouseEvent) {

// ……

}

**override** **fun** mouseEntered(e: MouseEvent) {

// ……

}

})

如果超类有一个构造函数则必须传递适当的构造函数参数给它。多个超类型可以由跟在冒号后面的逗号分隔的列表指定：

**open** **class** A(x: Int) {

**public** **open** **val** y: Int = x

}

**interface** B {……}

**val** ab: A = **object** : A(1), B {

**override** **val** y = 15

}

任何时候，如果我们只需要“一个对象而已”，并不需要特殊超类型，那么我们可以简单地写：

**fun** foo() {

**val** adHoc = **object** {

**var** x: Int = 0

**var** y: Int = 0

}

print(adHoc.x + adHoc.y)

}

注意：匿名对象可以用作只在本地和私有作用域中声明的类型。如果使用匿名对象作为公有函数的返回类型或用作公有熟悉感的类型，那么该函数或属性的实际类型会是匿名队形声明的超类型，如果没有声明任何超类型，就会是Any。在匿名对象中添加的成员将无法访问：

**class** C {

// 私有函数，所以其返回类型是匿名对象类型

**private** **fun** foo() = **object** {

**val** x: String = "x"

}

// 公有函数，所以其返回类型是 Any

**fun** publicFoo() = **object** {

**val** x: String = "x"

}

**fun** bar() {

**val** x1 = foo().x // 没问题

**val** x2 = publicFoo().x // 错误：未能解析的引用“x”

}

}

就像 Java 匿名内部类一样，对象表达式中的代码可以访问来自包含它的作用域的变量。 （与 Java 不同的是，这不仅限于 final 变量。）

对象声明：

**object** DataProviderManager {

**fun** registerDataProvider(provider: DataProvider) {

// ……

}

**val** allDataProviders: Collection<DataProvider>

**get**() = // ……

}

对象声明，它总是在object关键字后跟一个名称。就像变量声明一样，对象声明不是一个表达式，不能用在赋值语句的右边。

要引用该对象，直接使用名称即可：

DataProviderManager.registerDataProvider(……)

这些对象可以有超类型：

**object** DefaultListener : MouseAdapter() {

**override** **fun** mouseClicked(e: MouseEvent) {

// ……

}

**override** **fun** mouseEntered(e: MouseEvent) {

// ……

}

}

注意：对象声明不能再布局作用域，但是它们可以嵌套到其他对象声明或非内部类中。

伴生对象：

类内部的对象声明可以用companion关键字标记：

**class** MyClass {

**companion** **object** Factory {

**fun** create(): MyClass = MyClass()

}

}

该伴生对象的成员可通过只使用类名作为限定符来调用：

**val** instance = MyClass.create()

可以省略伴生对象的名称，在这种情况下将使用名称 Companion：

**class** MyClass {

**companion** **object** {

}

}

**val** x = MyClass.Companion

注意：即使伴生对象的成员看起来像其他语言的静态成员，在运行时它们仍然是真实对象的实例成员，还可以实现接口：

**interface** Factory<T> {

**fun** create(): T

}

**class** MyClass {

**companion** **object** : Factory<MyClass> {

**override** **fun** create(): MyClass = MyClass()

}

}

对象表达式和对象声明之间的语义差异

- 对象表达式是在使用它们的地方立即执行及初始化的

- 对象声明是在第一次被访问到时延迟初始化的

- 伴生对象的初始化是在相应的类被加载（解析）时

委托：

类委托：

类Derived可以继承一个接口Base，并将其所有共同方法委托给一个指定的对象：

**interface** Base {

**fun** print()

}

**class** BaseImpl(**val** x: Int) : Base {

**override** **fun** print() { print(x) }

}

**class** Derived(b: Base) : Base **by** b

**fun** main(args: Array<String>) {

**val** b = BaseImpl(10)

Derived(b).print() // 输出 10

}

Derived 的超类型列表中的by子句表示b将会在Derived中内部存储。并且编译器将生产转发给b所有的Base方法。

请注意，覆盖会以你所期望的方式工作：编译器会使用你的 override 实现取代委托对象中的实现。如果我们为 Derived 添加 override fun print() { print("abc") }，该程序会输出“abc”而不是“10”。

委托属性：

**class** Example {

**var** p: String **by** Delegate()

}

语法是： val/var <属性名>: <类型> by <表达式>。在by后面的表达式是该委托。因为属性对应的get（）和set（）会被委托给它的getValue（）setValue（）方法。属性的委托不必实现任何的接口，但是需要提供一个getValue（）（和setValue（）--对于var属性）

**class** Delegate {

**operator** **fun** getValue(thisRef: Any?, **property**: KProperty<\*>): String {

**return** "$thisRef, thank you for delegating '${property.name}' to me!"

}

**operator** **fun** setValue(thisRef: Any?, **property**: KProperty<\*>, value: String) {

println("$value has been assigned to '${property.name}' in $thisRef.")

}

}

当我们从委托到一个 Delegate实例的p读取时，将调动Delegate中的getValue（）函数，所以它的第一个参数读出p对象，第二个参数保存了对p自身的描述：

**val** e = Example()println(e.p)

输出结果：

Example@33a17727, thank you for delegating ‘p’ to me!

类似地，当我们给 p 赋值时，将调用 setValue() 函数。前两个参数相同，第三个参数保存将要被赋予的值：

e.p = "NEW"

输出结果：

NEW has been assigned to ‘p’ in Example@33a17727.

延迟属性Lazy：

Lazy（）是接受一个lambda并返回一个Lazy<T>实例的函数，返回的实例可以作为实现延迟属性的委托：第一次调用get（）会执行已传递给lazy（）的lambda表达式并记录结果，后续调用get（）只是返回记录的结果

**val** lazyValue: String **by** lazy {

println("computed!")

"Hello"

}

**fun** main(args: Array<String>) {

println(lazyValue)

println(lazyValue)

}

这个例子输出：

computed!

Hello

Hello

默认情况下，对于lazy属性的求值是同步锁的：该值只在一个线程中计算，并且所有线程会看到相同的值。如果初始化委托的同步锁不是必需的，这样多个线程可以同时执行，那么将LazyThreadSafetyMode.PUBLICATION 作为参数传递给 lazy() 函数。 而如果你确定初始化将总是发生在单个线程，那么你可以使用 LazyThreadSafetyMode.NONE 模式， 它不会有任何线程安全的保证和相关的开销。

### 可观察属性 Observable

[Delegates.observable()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.properties/-delegates/observable.html) 接受两个参数：初始值和修改时处理程序（handler）。 每当我们给属性赋值时会调用该处理程序（在赋值后执行）。它有三个参数：被赋值的属性、旧值和新值：

**import** kotlin.properties.Delegates

**class** User {

**var** name: String **by** Delegates.observable("<no name>") {

prop, old, new ->

println("$old -> $new")

}

}

**fun** main(args: Array<String>) {

**val** user = User()

user.name = "first"

user.name = "second"

}

这个例子输出：

<no name> -> first

first -> second

如果你想能够截获一个赋值并“否决”它，就使用 [vetoable()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.properties/-delegates/vetoable.html) 取代 observable()。 在属性被赋新值生效之前会调用传递给 vetoable 的处理程序。

## 把属性储存在映射中

一个常见的用例是在一个映射（map）里存储属性的值。 这经常出现在像解析 JSON 或者做其他“动态”事情的应用中。 在这种情况下，你可以使用映射实例自身作为委托来实现委托属性。

**class** User(**val** map: Map<String, Any?>) {

**val** name: String **by** map

**val** age: Int **by** map

}

在这个例子中，构造函数接受一个映射参数：

**val** user = User(mapOf(

"name" to "John Doe",

"age" to 25

))

委托属性会从这个映射中取值（通过字符串键——属性的名称）：

println(user.name) // Prints "John Doe"println(user.age) // Prints 25

这也适用于 var 属性，如果把只读的 Map 换成 MutableMap 的话：

**class** MutableUser(**val** map: MutableMap<String, Any?>) {

**var** name: String **by** map

**var** age: Int **by** map

}

## 局部委托属性（自 1.1 起）

你可以将局部变量声明为委托属性。 例如，你可以使一个局部变量惰性初始化：

**fun** example(computeFoo: () -> Foo) {

**val** memoizedFoo **by** lazy(computeFoo)

**if** (someCondition && memoizedFoo.isValid()) {

memoizedFoo.doSomething()

}

}

memoizedFoo 变量只会在第一次访问时计算。 如果 someCondition 失败，那么该变量根本不会计算。