示意 - 泛型代码

```
// 类的泛型
class Two<A,B> {
    public A first;
    public B second;
    public Two(A a, B b) {
        first = a;
        second = b;
    }
}
class Test {
    // 方法的泛型
    static <T> T get(T x) { return x; }
}
```

泛型是什么

上面代码中，Two这个类使用了泛型（在类名的后面指定了`<A,B>`），则在实例一个Two对象时可代入两个类型A和B。一般我们可以称Two为一个容器。使用如下：

```
Two<String, Integer> a = new Two<>("Test", 2);
a.first = "modify";
System.out.print(a.first);
```

泛型方法的声明，见Test类中的get方法，会根据代入的类型确定T：

```
System.out.println(Test.get(3)+Test.get("test..."));
```

有什么用

就可以方便的代入指定类型，达到模板复用的效果。

上方对Two类的使用中，虽然在类里first域是一个A类型，无意义。但在我调用a.first时，已经被IDE与编译器视为String类型了。

效果：只要你指明了类型，编译器会帮你处理好一切。

另外：

泛型类与泛型方法完全可以分开考虑，泛型类可以没有泛型方法，泛型方法可以不在泛型类中（可参考考虑Test类）

擦除与边界

如前面所见的Two类型，first与second貌似已经在新建中变成了String与Integer类型，在取出与存入时都可以将其视为对应类型。但实际上不是这样的。

first与second在Two类中，是没有其它属性的，它们只是一个Object。

那为什么在读写时我们可以使用对应的类型呢？

以下需要提两个概念：

擦除：可以在类Two中尝试打入`a.`会发现没有什么功能。因为编译器在类的内部处理中，将first与second的**属性擦除**了，只把它们视为Object。

边界：编译器唯一对泛型处理的地方是，当我们写入如`a.second = 3`时，编译器将3转为一个`Object`并赋给了`a.second`；当我们读取此域如`System.out.println(a.second)`时，编译器将`a.second`读出后转为了Integer类型。

即是说，**一切泛型的作用，都只发生在它读取的时候**。

如何保持它在类中功能

这涉及到extends方法，见如下代码，B是A的实现类或派生类：

```
interface A { int get(); }
class B implements A {
    public int get() { return 1; }
}
class Test1<T extends A> {
    T x;
    Test1(T x) { this.x = x; }
    void test() { System.out.println(x.get()); }
}

public class Demo
{
    @Test
    public void test() {
        Test1<B> t = new Test1<>(new B());
        t.test();
    }
}
```

使用此类写法时，内部直接将x看作A类型了。那么这样有何意义呢？

我认为唯一的意义还是在于**边界**处，即读写时，会将此成员转为B类型。

泛型的不足

以下不足可以在面试中吹吹牛皮用：

擦除的问题

由于擦除，Java不能获取泛型的具体信息。这就导致以下后果：

泛型在C++中对应的功能是模板类，在模板中可以使用T x的方法，比如x.f()，然后在实例此模板时编译器自动判断T为是否带有f()方法。而Java将T的一切都擦除了，导致T实际上几乎没有功能。虽然可以使用来补充擦除的边界，但这样还不如直接把T当成基类HasF来使用（即将类中的T去掉，将所有用到T的类型换成HasF），也能达到一样的效果。因此在平时直接使用T这样的泛型时，必须提醒自己，这是一个Object，泛型的效果仅在发生在代入与传出值时。

不能使用一些类型

`Test<T>`中的T不能代入int\float等Java默认基本类型（虽然已有对应的Integer等类了）

`Test<T>`中的T不能是Throwable的派生类，即是代入的T不能是会抛出异常的类。

不能同时拥有一个泛型接口的多种实现

```
class C<T> {}
class A implements C<Integer> {}
// error
class B extends A implements C<String> {}
```

这样编译是无法通过的，B继承A时已经拥有了`C<Integer>`就无法使用的。这种限定的好坏见仁见智吧。

不能重载

如上述`Two<A,B>`中如果实现一个函数void f(A x)就不能再写void f(B x)了，这不符合Java重载的机制。原因其实很好理解，因为A与B在类的内部统一被当作Object了。