http://www.cnblogs.com/huang0925/p/3160094.html

Filter(com.google.common.collect.Collections2.*filter*)

我们先创建一个简单的Person类。

```
public class Person {
    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;
    }

    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

如果要产生一个Person类的List，通常的写法可能是这样子。

```
List<Person> peopleList = new ArrayList<Person>();
        peopleList.add(new Person("bowen",27));
        peopleList.add(new Person("bob", 20));
        peopleList.add(new Person("Katy", 18));
        peopleList.add(new Person("Logon", 24));
```

而Guava提供了一个newArrayList的方法，其自带类型推演，并可以方便的生成一个List,并且通过参数传递初始化值。

```
List<Person> people = newArrayList(new Person("bowen", 27),
                new Person("bob", 20),
                new Person("Katy", 18),
```

```
                new Person("Logon", 24));
```

当然，这不算函数式编程的范畴，这是Guava给我们提供的一个实用的函数。
如果我们**选取其中年龄大于20的人**，通常的写法可能是这样子。

```
 List<Person> oldPeople = new ArrayList<Person>();
        for (Person person : peopleList) {
            if (person.getAge() >= 20) {
                oldPeople.add(person);
            }
        }
```

这就是典型的filter模式。filter即从一个集合中根据一个条件筛选元素。其中person.getAge() >=20就是这个条件。
Guava为这种模式提供了一个filter的方法。所以我们可以这样写。

```
List<Person> oldPeople = newArrayList(
    filter(peopleList, new Predicate<Person>() {
            public boolean apply(Person person) {
                return person.getAge() >= 20;
            }
        }));
```

And Predicate(com.google.common.base.Predicate)

如果要再实现一个方法来查找People列表中所有名字中包含b字母的列表，我们可以用Guava简单的实现。

```
 List<Person> namedPeople = newArrayList(filter(people, new
Predicate<Person>() {
            public boolean apply(Person person) {
                return person.getName().contains("b");
            }
        }));
```

一切是这么的简单。 那么新需求来了，如果现在需要找年龄>=20并且名称包含b的人，该如何实现那？ 可能你会这样写。

```
List<Person> filteredPeople = newArrayList(filter(people, new
Predicate<Person>() {
            public boolean apply(Person person) {
                return person.getName().contains("b") &&
person.getAge() >= 20;
            }
        }));
```

这样写的话就有一定的代码重复，因为之前我们已经写了两个Predicate来分别实现这两个条件判断，能不能重用之前的Predicate那？答案是能。 我们首先将之前生成年龄判断和名称判断的两个Predicate抽成方法。

```
private Predicate<Person> ageBiggerThan(final int age) {
        return new Predicate<Person>() {
            public boolean apply(Person person) {
                return person.getAge() >= age;
            }
        };
    }

private Predicate<Person> nameContains(final String str) {
        return new Predicate<Person>() {
            public boolean apply(Person person) {
                return person.getName().contains(str);
            }
        };
    }
```

而我们的结果其实就是这两个Predicate相与。Guava给我们提供了and方法，用于对一组Predicate求与。

```
List<Person> filteredPeople = newArrayList(filter(people,
and(ageBiggerThan(20), nameContains("b"))));
```

由于and接收一组Predicate，返回也是一个Predicate，所以可以直接作为filter的第二个参数。如果不熟悉函数式编程的人可能感觉有点怪异，但是习惯了就会觉得它的强大与简洁。 当然除了and，Guava还为我们提供了or，用于对一组Predicate求或。这里就不多讲了，大家可以自己练习下。

Map(transform)

列表操作还有另一个常见的模式，就是将数组中的所有元素映射为另一种元素的列表，这就是map pattern。举个例子，求People列表中的所有人名。程序员十有八九都会这样写。

```
List<String> names = new ArrayList<String>();
        for (Person person : people) {
            names.add(person.getName());
        }
```

Guava已经给我们提供了这种Pattern的结果办法，那就是使用transform方法。

```
 List<String> names = newArrayList(transform(people, new
Function<Person, String>() {
            public String apply( Person person) {
                return person.getName();
            }
        }));
```

也可以写成Lambda表达式：

```
List<String> names = newArrayList(transform(people, person ->
person.getName()));
                                                    //person:apply方
法的参数名;
                                                    
//person.getName():apply方法的返回值
```

reduce

除了filter与map模式外，列表操作还有一种reduce操作。比如求people列表中所有人年龄的和。Guava并未提供reduce方法。具体原因我们并不清楚。但是我们可以自己简单的实现一个reduce pattern。 先定义一个Func的接口。

```
public interface Func<F,T> {
         T apply(F currentElement, T origin);
     }
```

apply方法的第一个参数为列表中的当前元素，第二个参数为默认值，返回值类型为默认值类型。 然后我们定义个reduce的静态方法。

```
public class Reduce {
    private Reduce() {

    }

    public static <F,T> T reduce(final Iterable<F> iterable, final
Func<F, T> func, T origin) {

        for (Iterator iterator = iterable.iterator();
iterator.hasNext(); ) {
            origin = func.apply((F)(iterator.next()), origin);
        }

        return origin;
    }
```

```
}
```

reduce方法接收三个参数，第一个是需要进行reduce操作的列表，第二个是封装reduce操作的Func，第三个参数是初始值。
我们可以使用这个reduce来实现求people列表中所有人的年龄之和。

```
Integer ages = Reduce.reduce(people, new Func<Person, Integer>() {

            public Integer apply(Person person, Integer origin) {
                return person.getAge() + origin;
            }
        }, 0);
```

我们也可以轻松的写一个方法来得到年龄的最大值。

```
Integer maxAge = Reduce.reduce(people, new Func<Person, Integer>()
{

            public Integer apply(Person person, Integer origin) {
                return person.getAge() > origin ? person.getAge()
: origin;
            }
        }, 0);
```

Fluent pattern

现在新需求来了，需要找出年龄>=20岁的人的所有名称。该如何操作那？我们可以使用filter过滤出年龄>=20的人，然后使用transform得到剩下的所有人的人名。

```
 private Function<Person, String> getName() {
        return new Function<Person, String>() {
            public String apply( Person person) {
                return person.getName();
            }
        };
    }

    public void getPeopleNamesByAge() {

        List<String> names = newArrayList(transform(filter(people,
ageBiggerThan(20)), getName()));
    }
```

这样括号套括号的着实不好看。能不能改进一下那？Guava为我们提供了fluent模式

的API,我们可以这样来写。

```
List<String> names =
from(people).filter(ageBiggerThan(20)).transform(getName()).toList
();
```