Python Generated Code

This page describes exactly what Python definitions the protocol buffer compiler generates for any given protocol definition. You should read the language guide before reading this document.

The Python Protocol Buffers implementation is a little different from C++ and Java. In Python, the compiler only outputs code to build descriptors for the generated classes, and a Python metaclass does the real work. This document describes what you get *after* the metaclass has been applied.

Compiler Invocation

The protocol buffer compiler produces Python output when invoked with the `--python_out=` command-line flag. The parameter to the `--python_out=` option is the directory where you want the compiler to write your Python output. The compiler creates a `.py` file for each `.proto` file input. The names of the output files are computed by taking the name of the `.proto` file and making two changes:

- The extension (`.proto`) is replaced with `_pb2.py`.
- The proto path (specified with the `--proto_path=` or `-I` command-line flag) is replaced with the output path (specified with the `--python_out=` flag).

So, for example, let's say you invoke the compiler as follows:

```
protoc --proto_path=src --python_out=build/gen src/foo.proto src/bar/baz.proto
```

The compiler will read the files `src/foo.proto` and `src/bar/baz.proto` and produce two output files: `build/gen/foo_pb2.py` and `build/gen/bar/baz_pb2.py`. The compiler will automatically create the directory `build/gen/bar` if necessary, but it will *not* create `build` or `build/gen`; they must already exist.

Note that if the `.proto` file or its path contains any characters which cannot be used in Python module names (for example, hyphens), they will be replaced with underscores. So, the file `foo-bar.proto` becomes the Python file `foo_bar_pb2.py`.

> When outputting Python code, the protocol buffer compiler's ability to output directly to ZIP archives is particularly convenient, as the Python interpreter is able to read directly from these archives if placed in the `PYTHONPATH`. To output to a ZIP file, simply provide an output location ending in `.zip`.

> The number 2 in the extension `_pb2.py` designates version 2 of Protocol Buffers. Version 1 was used primarily inside Google, though you might be able to find parts of it included in other Python code that was released before Protocol Buffers. Since version 2 of Python Protocol Buffers has a completely different interface, and since Python does not have compile-time type checking to catch mistakes, we chose to make the version number be a prominent part of generated Python file names.

Packages

The Python code generated by the protocol buffer compiler is completely unaffected by the package name defined in the `.proto` file. Instead, Python packages are identified by directory structure.

Messages

Given a simple message declaration:

```
message Foo {}
```

The protocol buffer compiler generates a class called `Foo`, which subclasses `google.protobuf.Message`. The class is a concrete class; no abstract methods are left unimplemented. Unlike C++ and Java, Python generated code is unaffected by the `optimize_for` option in the `.proto` file; in effect, all Python code is optimized for code size.

You should *not* create your own `Foo` subclasses. Generated classes are not designed for subclassing and may lead to "fragile base class" problems. Besides, implementation inheritance is bad design.

Python message classes have no particular public members other than those defined by the `Message` interface and those generated for nested fields, messages, and enum types (described below). `Message` provides methods you can use to check, manipulate, read, or write the entire message, including parsing from and serializing to binary strings. In addition to these methods, the `Foo` class defines the following static methods:

- `FromString(s)`: Returns a new message instance deserialized from the given string.

Note that you can also use the `text_format` module to work with protocol messages in text format: for example, the `Merge()` method lets you merge an ASCII representation of a message into an existing message.

A message can be declared inside another message. For example: `message Foo { message Bar { } }`

In this case, the `Bar` class is declared as a static member of `Foo`, so you can refer to it as `Foo.Bar`.

Fields

For each field in a message type, the corresponding class has a member with the same name as the field. How you can manipulate the member depends on its type.

As well as accessor methods, the compiler generates an integer constant for each field containing its field number. The constant name is the field name converted to upper-case followed by `_FIELD_NUMBER`. For example, given the field `optional int32 foo_bar = 5;`, the compiler will generate the constant `FOO_BAR_FIELD_NUMBER = 5`.

Singular Fields

If you have a singular (optional or required) field `foo` of any non-message type, you can manipulate the field `foo` as if it were a regular field. For example, if `foo`'s type is `int32`, you can say:

```
message.foo = 123
print message.foo
```

Note that setting `foo` to a value of the wrong type will raise a `TypeError`.

If `foo` is read when it is not set, its value is the default value for that field. To check if `foo` is set, or to clear the value of `foo`, you must call the `HasField()` or `ClearField()` methods of the `Message` interface. For example:

```
assert not message.HasField("foo")
message.foo = 123
assert message.HasField("foo")
message.ClearField("foo")
assert not message.HasField("foo")
```

Singular Message Fields

Message types work slightly differently. You cannot assign a value to an embedded message field. Instead, assigning a value to any field within the child message implies setting the message field in the parent. So, for example, let's say you have the following `.proto` definition:

```
message Foo {
  optional Bar bar = 1;
}
message Bar {
  optional int32 i = 1;
}
```

You *cannot* do the following:

```
foo = Foo()
foo.bar = Bar()  # WRONG!
```

Instead, to set `bar`, you simply assign a value directly to a field within `bar`, and - presto! - `foo` has a `bar` field:

```
foo = Foo()
assert not foo.HasField("bar")
foo.bar.i = 1
assert foo.HasField("bar")
assert foo.bar.i == 1
foo.ClearField("bar")
assert not foo.HasField("bar")
assert foo.bar.i == 0  # Default value
```

Similarly, you can set `bar` using the Message interface's `CopyFrom()` method. This copies all the values from another message of the same type as `bar`.

```
foo.bar.CopyFrom(baz)
```

Note that simply reading a field inside `bar` does *not* set the field:

```
foo = Foo()
assert not foo.HasField("bar")
print foo.bar.i  # Print i's default value
assert not foo.HasField("bar")
```

Repeated Fields

Repeated fields are represented as an object that acts like a Python sequence. As with embedded messages, you cannot assign the field directly, but you can manipulate it. For example, given this message definition:

```
message Foo {
  repeated int32 nums = 1;
}
```

You can do the following:

```
foo = Foo()
foo.nums.append(15)        # Appends one value
foo.nums.extend([32, 47])  # Appends an entire list

assert len(foo.nums) == 3
assert foo.nums[0] == 15
assert foo.nums[1] == 32
assert foo.nums == [15, 32, 47]

foo.nums[1] = 56    # Reassigns a value
assert foo.nums[1] == 56
for i in foo.nums:  # Loops and print
  print i
del foo.nums[:]     # Clears list (works just like in a Python list)
```

The `ClearField()` method of the `Message` interface works as well in addition to using Python `del`.

Repeated Message Fields

Repeated messages works similar to repeated scalar fields, except the corresponding Python object does not have an `append()` function. Instead, it has an `add()` function that creates a new message object, appends it to the list, and returns it for the caller to fill in. It also has an `extend()` function that appends an entire list of messages, but makes a **copy** of every message in the list. This is done so that messages are always owned by the parent message to avoid circular references and other confusion that can happen when a mutable data structure has multiple owners.

For example, given this message definition:

```
message Foo {
  repeated Bar bars = 1;
}
message Bar {
  optional int32 i = 1;
  optional int32 j = 2;
}
```

You can do the following:

```
foo = Foo()
bar = foo.bars.add()        # Adds a Bar then modify
bar.i = 15
foo.bars.add().i = 32       # Adds and modify at the same time
new_bar = Bar()
new_bar.i = 47
foo.bars.extend([new_bar])  # Uses extend() to copy

assert len(foo.bars) = 3
assert foo.bars[0].i == 15
assert foo.bars[1].i == 32
assert foo.bars[2].i == 47
assert foo.bars[2] == new_bar      # The extended message is equal,
assert foo.bars[2] is not new_bar  # but it is a copy!

foo.bars[1].i = 56    # Modifies a single element
assert foo.bars[1].i == 56
for bar in foo.bars:  # Loops and print
  print bar.i
del foo.bars[:]       # Clears list

# add() also forwards keyword arguments to the concrete class.
# For example, you can do:

foo.bars.add(i = 12, j = 13)
```

Enumerations

In Python, enums are just integers. A set of integral constants are defined corresponding to the enum's defined values. For example, given:

```
message Foo {
  enum SomeEnum {
    VALUE_A = 1;
    VALUE_B = 5;
    VALUE_C = 1234;
  }
  optional SomeEnum bar = 1;
}
```

The constants `VALUE_A`, `VALUE_B`, and `VALUE_C` are defined with values 1, 5, and 1234, respectively. No type corresponding to `SomeEnum` is defined. If an enum is defined in the outer scope, the values are module constants; if it is defined within a message (like above), they become static members of that message class.

An enum field works just like a scalar field. It does **not** do any type checking in the setter or getter.

```
foo = Foo()
foo.bar = Foo.VALUE_A
assert foo.bar == 1
assert foo.bar == Foo.VALUE_A
```

Note that in C++ and Java, an enum field cannot contain a numeric value other than those defined for the enum type. If an unknown enum value is encountered while parsing, the field will be treated as if its tag number were unknown. Therefore, you should never assign an enum field to an undefined value in Python, either. A future version of the library may explicitly disallow this.

Extensions

Given a message with an extension range:

```
message Foo {
  extensions 100 to 199;
}
```

The Python class corresponding to `Foo` will have a member called `Extensions`, which is a dictionary mapping extension identifiers to their current values.

Given an extension definition:

```
extend Foo {
  optional int32 bar = 123;
}
```

The protocol buffer compiler generates an "extension identifier" called `bar`. The identifier acts as a key to the `Extensions` dictionary. The result of looking up a value in this dictionary is exactly the same as if you accessed a normal field of the same type. So, given the above example, you could do:

```
foo = Foo()
foo.Extensions[proto_file_pb2.bar] = 2
assert foo.Extensions[proto_file_pb2.bar] == 2
```

Note that you need to specify the extension identifier constant, not just a string name: this is because it's possible for multiple extensions with the same name to be specified in different scopes.

Analogous to normal fields, `Extensions[...]` returns a message object for singular messages and a sequence for repeated fields.

The `Message` interface's `HasField()` and `ClearField()` methods do not work with extensions; you must use `HasExtension()` and `ClearExtension()` instead.

Services

If the `.proto` file contains the following line:

```
option py_generic_services = true;
```

Then the protocol buffer compiler will generate code based on the service definitions found in the file as described in this section. However, the generated code may be undesirable as it is not tied to any particular RPC system, and thus requires more levels of indirection that code tailored to one system. If you do NOT want this code to be generated, add this line to the file:

```
option py_generic_services = false;
```

If neither of the above lines are given, the option defaults to `false`, as generic services are deprecated. (Note that prior to 2.4.0, the option defaults to `true`)

RPC systems based on `.proto`-language service definitions should provide plugins to generate code approriate for the system. These plugins are likely to require that abstract services are disabled, so that they can generate their own classes of the same names. Plugins are new in version 2.3.0 (January 2010).

The remainder of this section describes what the protocol buffer compiler generates when abstract services are enabled.

Interface

Given a service definition:

```
service Foo {
  rpc Bar(FooRequest) returns(FooResponse);
}
```

The protocol buffer compiler will generate a class `Foo` to represent this service. `Foo` will have a method for each method defined in the service definition. In this case, the method `Bar` is defined as:

```
def Bar(self, rpc_controller, request, done)
```

The parameters are equivalent to the parameters of `Service.CallMethod()`, except that the `method_descriptor` argument is implied.

These generated methods are intended to be overridden by subclasses. The default implementations simply call `controller.SetFailed()` with an error message indicating that the method is unimplemented, then invoke the `done` callback. When implementing your own service, you must subclass this generated service and implement its methods as appropriate.

`Foo` subclasses the `Service` interface. The protocol buffer compiler automatically generates implementations of the methods of `Service` as follows:

- `GetDescriptor`: Returns the service's `ServiceDescriptor`.
- `CallMethod`: Determines which method is being called based on the provided method descriptor and calls it directly.
- `GetRequestClass` and `GetResponseClass`: Returns the class of the request or response of the correct type for the given method.

Stub

The protocol buffer compiler also generates a "stub" implementation of every service interface, which is used by clients wishing to send requests to servers implementing the service. For the `Foo` service (above), the stub implementation `Foo_Stub` will be defined.

`Foo_Stub` is a subclass of `Foo`. Its constructor takes an `RpcChannel` as a parameter. The stub then implements each of the service's methods by calling the channel's `CallMethod()` method.

The Protocol Buffer library does not include an RPC implementation. However, it includes all of the tools you need to hook up a generated service class to any arbitrary RPC implementation of your choice. You need only provide implementations of `RpcChannel` and `RpcController`.

Plugin Insertion Points

Code generator plugins which want to extend the output of the Python code generator may insert code of the following types using the given insertion point names.

- `imports`: Import statements.
- `module_scope`: Top-level declarations.
- `class_scope:TYPENAME`: Member declarations that belong in a message class. `TYPENAME` is the full proto name, e.g. `package.MessageType`.

> Do not generate code which relies on private class members declared by the standard code generator, as these implementation details may change in future versions of Protocol Buffers.

C++ Implementation

There is also an experimental C++ implementation for Python messages via a Python extension for better performance. Implementation type is controlled by an environment variable `PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION` (valid values: "cpp" and "python"). The default value is currently "python" but will be changed to "cpp" in future release.

Note that the environment variable needs to be set before installing the protobuf library, in order to build and install the python extension. The C++ implementation also requires CPython platforms. See `python/INSTALL.txt` for detailed install instructions.

Last updated 三月 5, 2013.